conference
proceedings

2015 USENIX Annual Technical Conference

Santa Clara, CA, USA
July 8–10, 2015

Sponsored by
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Proceedings of the 2015 USENIX Annual Technical Conference
Santa Clara, CA, USA July 8–10, 2015

Thanks to Our USENIX ATC '15 Sponsors

Gold Sponsors






Silver Sponsor



Bronze Sponsors





Media Sponsors and Industry Partners

ACM *Queue*
ADMIN magazine
CRC Press
Datanami
Distributed Management Task Force (DMTF)
EnterpriseTech
FreeBSD Foundation
HPCwire
InfoSec News
Linux Journal
Linux Pro Magazine
No Starch Press
Raspberry Pi Geek
Ubuntu User
UserFriendly.org
Virus Bulletin

© 2015 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-931971-225

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons

Facebook Google Microsoft Research
NetApp VMware

USENIX Benefactors

Hewlett-Packard *Linux Pro Magazine* Symantec

USENIX and LISA SIG Partners

Booking.com Cambridge Computer
Can Stock Photo Fotosearch Google

USENIX Partners

Cisco Meraki EMC Huawei

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of USENIX ATC '15
2015 USENIX Annual Technical Conference

July 8–10, 2015
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

Shan Lu, *University of Chicago*
Erik Riedel, *EMC*

Program Committee

Keith Adams, *Facebook AI Research*
Jeremy Andrus, *Apple*
Edouard Bugnion, *École Polytechnique Fédérale de Lausanne (EPFL)*
Haibo Chen, *Shanghai Jiao Tong University*
Dilma Da Silva, *Texas A&M University*
Fred Douglis, *EMC*
Dawson Engler, *Stanford University*
Phillipa Gill, *Stony Brook University*
Dirk Grunwald, *University of Colorado, Boulder*
Ajay Gulati, *ZeroStack, Inc.*
Haryadi Gunawi, *University of Chicago*
Anthony D. Joseph, *University of California, Berkeley*
Orran Krieger, *Boston University*
Pradeep Padala, *VMware*
Liane Praza, *Oracle*
Feng Qin, *Ohio State University*
Liuba Shrira, *Brandeis University*
David Shue, *Google*
Emil Sit, *HubSpot*
Christopher Small, *Philo Inc.*
Theodore Ts'o, *Google*
Dan Tsafrir, *Technion—Israel Institute of Technology*
Andy Tucker, *Bracket Computing*
Meg Walraed-Sullivan, *Microsoft Research*
Xi Wang, *University of Washington*
Alec Wolman, *Microsoft Research*
Chia-Lin Yang, *National Taiwan University*
Ding Yuan, *University of Toronto*
Lin Zhong, *Rice University*

External Reviewers

Mohamed Alizadeh
David Andersen
Adam Belay
Yuan-Hao Chang
Che-Wei Chang
Yi-Jung Chen
Chen-Mou Cheng
Peter Desnoyers
Michael Factor
Sorin Faibish
Jonas Fietz
Tim Finley
Brian Ford
Cristiano Giuffrida
Daniel Goldberg
Sean Patrick Hogan
Anne Holler
Baris Kasikci
Geoff Kuenning
Cheng Li
Ren-Shuo Liu
Steve Muir
Erik Munson
Rishab Nithyanand
George Prekas
Mia Primorac
Srini Seetharaman
Zili Shao
Ioan Stefanovici
Doug Terry
Amin Tootoonchian
Mustafa Uysal
Carl Waldspurger
Grant Wallace
Jason Xue
Zhao Zhang

USENIX ATC '15
2015 USENIX Annual Technical Conference
July 8–10, 2015
Santa Clara, CA

Message from the Program Co-Chairs vii

Wednesday, July 8, 2015

Parallel & Distributed Systems

Spartan: A Distributed Array Framework with Smart Tiling 1
Chien-Chin Huang, *New York University;* Qi Chen, *Peking University;* Zhaoguo Wang and Russell Power, *New York University;* Jorge Ortiz, *IBM T.J. Watson Research Center;* Jinyang Li, *New York University;* Zhen Xiao, *Peking University*

Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code 17
Ryan Stutsman, *University of Utah;* Collin Lee and John Ousterhout, *Stanford University*

Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication 31
Asaf Cidon, *Stanford University;* Robert Escriva, *Cornell University;* Sachin Katti and Mendel Rosenblum, *Stanford University;* Emin Gün Sirer, *Cornell University*

Callisto-RTS: Fine-Grain Parallel Loops 45
Tim Harris, *Oracle Labs;* Stefan Kaestle, *ETH Zürich*

Cloud Storage

LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache 57
Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, and Yingwei Luo, *Peking University;* Chen Ding, *University of Rochester;* Song Jiang, *Wayne State University;* Zhenlin Wang, *Michigan Technological University*

LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data 71
Xingbo Wu and Yuehai Xu, *Wayne State University;* Zili Shao, *The Hong Kong Polytechnic University;* Song Jiang, *Wayne State University*

MetaSync: File Synchronization Across Multiple Untrusted Storage Services 83
Seungyeop Han and Haichen Shen, *University of Washington;* Taesoo Kim, *Georgia Institute of Technology;* Arvind Krishnamurthy, Thomas Anderson, and David Wetherall, *University of Washington*

Pyro: A Spatial-Temporal Big-Data Storage System 97
Shen Li and Shaohan Hu, *University of Illinois at Urbana-Champaign;* Raghu Ganti and Mudhakar Srivatsa, *IBM Research;* Tarek Abdelzaher, *University of Illinois at Urbana-Champaign*

CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal 111
Mingqiang Li, Chuan Qin, and Patrick P. C. Lee, *The Chinese University of Hong Kong*

Dependability

Surviving Peripheral Failures in Embedded Systems 125
Rebecca Smith and Scott Rixner, *Rice University*

***Log*^{2}: A Cost-Aware Logging Mechanism for Performance Diagnosis** 139
Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, and Qingwei Lin, *Microsoft Research;* Qiang Fu, *Microsoft;* Dongmei Zhang, *Microsoft Research;* Tao Xie, *University of Illinois at Urbana-Champaign*

(Wednesday, July 8, continues on the next page)

Identifying Trends in Enterprise Data Protection Systems. 151
George Amvrosiadis, *University of Toronto;* Medha Bhadkamkar, *Symantec Research Labs*

Systematically Exploring the Behavior of Control Programs. 165
Jason Croft, *University of Illinois at Urbana-Champaign;* Ratul Mahajan, *Microsoft Research;* Matthew Caesar, *University of Illinois at Urbana-Champaign;* Madan Musuvathi, *Microsoft Research*

Fence: Protecting Device Availability With Uniform Resource Control 177
Tao Li and Albert Rafetseder, *New York University;* Rodrigo Fonseca, *Brown University;* Justin Cappos, *New York University*

Thursday, July 9

File Systems & Flash

Request-Oriented Durable Write Caching for Application Performance 193
Sangwook Kim, *Sungkyunkwan University;* Hwanju Kim, *University of Cambridge;* Sang-Hoon Kim, *Korea Advanced Institute of Science and Technology (KAIST);* Joonwon Lee and Jinkyu Jeong, *Sungkyunkwan University*

NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store 207
Leonardo Marmol, *Florida International University;* Swaminathan Sundararaman and Nisha Talagala, *SanDisk;* Raju Rangaswami, *Florida International University*

Lightweight Application-Level Crash Consistency on Transactional Flash Storage 221
Changwoo Min, *Georgia Institute of Technology;* Woon-Hak Kang, *Sungkyunkwan University;* Taesoo Kim, *Georgia Institute of Technology;* Sang-Won Lee and Young Ik Eom, *Sungkyunkwan University*

WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. 235
Wongun Lee, Keonwoo Lee, and Hankeun Son, *Hanyang University;* Wook-Hee Kim and Beomseok Nam, *Ulsan National Institute of Science and Technology;* Youjip Won, *Hanyang University*

SpanFS: A Scalable File System on Fast Storage Devices. 249
Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai, *Beihang University*

Memory

Shoal: Smart Allocation and Replication of Memory For Parallel Programs 263
Stefan Kaestle, Reto Achermann, and Timothy Roscoe, *ETH Zürich;* Tim Harris, *Oracle Labs, Cambridge*

Thread and Memory Placement on NUMA Systems: Asymmetry Matters 277
Baptiste Lepers, *Simon Fraser University;* Vivien Quéma, *Grenoble INP;* Alexandra Fedorova, *Simon Fraser University*

Latency-Tolerant Software Distributed Shared Memory 291
Jacob Nelson, Brandon Holt, Brandon Myers, Preston Brigg, Luis Ceze, Simon Kahan, and Mark Oskin, *University of Washington*

NightWatch: Integrating Lightweight and Transparent Cache Pollution Control into Dynamic Memory Allocation Systems. 307
Rentong Guo, Xiaofei Liao, and Hai Jin, *Huazhong University of Science and Technology;* Jianhui Yue, *Auburn University;* Guang Tan, *Chinese Academy of Sciences*

Security

Secure Deduplication of General Computations 319
Yang Tang and Junfeng Yang, *Columbia University*

Lamassu: Storage-Efficient Host-Side Encryption 333
Peter Shah and Won So, *NetApp Inc.*

SecPod: a Framework for Virtualization-based Security Systems. 347
Xiaoguang Wang, *Xi'an Jiaotong University and Florida State University;* Yue Chen and Zhi Wang, *Florida State University;* Yong Qi, *Xi'an Jiaotong University;* Yajin Zhou, *Qihoo 360*

Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications 361
Jun Wang, *The Pennsylvania State University;* Xi Xiong, *Facebook Inc. and The Pennsylvania State University;* Peng Liu, *The Pennsylvania State University*

Graph Processing

GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning . . .375
Xiaowei Zhu, Wentao Han, and Wenguang Chen, *Tsinghua University*

GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC 387
Kai Wang and Guoqing Xu, *University of California, Irvine;* Zhendong Su, *University of California, Davis;* Yu David Liu, *SUNY at Binghamton*

Friday, July 10

Networking

Accurate Latency-based Congestion Feedback for Datacenters 403
Changhyun Lee and Chunjong Park, *Korea Advanced Institute of Science and Technology (KAIST);* Keon Jang, *Intel Labs;* Sue Moon and Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*

Mahimahi: Accurate Record-and-Replay for HTTP 417
Ravi Netravali, Anirudh Sivaraman, Somak Das, and Ameesh Goyal, *MIT CSAIL;* Keith Winstein, *Stanford University;* James Mickens, *Harvard University;* and Hari Balakrishnan, *MIT CSAIL*

Slipstream: Automatic Interprocess Communication Optimization 431
Will Dietz, Joshua Cranmer, Nathan Dautenhahn, and Vikram Adve, *University of Illinois at Urbana-Champaign*

FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency 445
Jihyung Lee, *Korea Advanced Institute of Science and Technology (KAIST);* Sungryoul Lee and Junghee Lee, *The Attached Institute of ETRI;* Yung Yi and KyoungSoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*

Scheduling at Large Scale

Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. 459
Andrey Goder, Alexey Spiridonov, and Yin Wang, *Facebook, Inc.*

Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing 473
Rohan Gandhi, Y. Charlie Hu and Cheng-kok Koh, *Purdue University;* Hongqiang (Harry) Liu and Ming Zhang, *Microsoft Research*

Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters 485
Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga, *Microsoft Corporation*

(Friday, July 10, continues on the next page)

Hawk: Hybrid Datacenter Scheduling 499
Pamela Delgado and Florin Dinu, *École Polytechnique Fédérale de Lausanne (EPFL);*
Anne-Marie Kermarrec, *Inria;* Willy Zwaenepoel, *École Polytechnique Fédérale de Lausanne (EPFL)*

OS & Hardware

Bolt: Faster Reconfiguration in Operating Systems 511
Sankaralingam Panneerselvam and Michael M. Swift, *University of Wisconsin—Madison*

Boosting GPU Virtualization Performance with Hybrid Shadow Page Tables. 517
Yaozu Dong and Mochi Xue, *Shanghai Jiao Tong University and Intel Corporation;* Xiao Zheng, *Intel Corporation*; Jiajun Wang, *Shanghai Jiao Tong University and Intel Corporation;* Zhengwei Qi and Haibing Guan, *Shanghai Jiao Tong University*

Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems 529
Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen, *University of Rochester*

Establishing a Base of Trust with Performance Counters for Enterprise Workloads 541
Andrzej Nowak, *CERN openlab and École Polytechnique Fédérale de Lausanne (EPFL);* Ahmad Yasin, *Intel;* Avi Mendelson, *Technion—Israel Institute of Technology;* Willy Zwaenepoel, *École Polytechnique Fédérale de Lausanne (EPFL)*

Utilizing the IOMMU Scalably 549
Omer Peleg and Adam Morrison, *Technion—Israel Institute of Technology;* Benjamin Serebrin, *Google;* Dan Tsafrir, *Technion—Israel Institute of Technology*

At Small Scale

Selectively Taming Background Android Apps to Improve Battery Lifetime 563
Marcelo Martins, *Brown University;* Justin Cappos, *New York University;* Rodrigo Fonseca, *Brown University*

U-root: A Go-based, Firmware Embeddable Root File System with On-demand Compilation 577
Ronald G. Minnich, *Google;* Andrey Mirtchovski, *Cisco*

LPD: Low Power Display Mechanism for Mobile and Wearable Devices. 587
MyungJoo Ham, Inki Dae, and Chanwoo Choi, *Samsung Electronics*

Memory-Centric Data Storage for Mobile Systems. 599
Jinglei Ren, *Tsinghua University;* Chieh-Jan Mike Liang, *Microsoft Research;* Yongwei Wu, *Tsinghua University;*Thomas Moscibroda, *Microsoft Research*

WearDrive: Fast and Energy-Efficient Storage for Wearables 613
Jian Huang, *Georgia Institute of Technology;* Anirudh Badam, Ranveer Chandra and Edmund B. Nightingale, *Microsoft Research*

Message from the USENIX ATC '15 Program Co-Chairs

Welcome to the 2015 USENIX Annual Technical Conference

This year's program committee has put together a program of 47 refereed papers and 15 practitioner talks. These papers and talks span a wide range of topics covering both novel research contributions and practical ideas in storage systems, networking, memory management, data analytics, parallel and distributed systems, mobile computing and consumer electronics, security, reliability, and virtualization.

We introduced a new track for industrial practitioners to submit talk proposals this year and received 17 talk proposals submitted by practitioners from 13 different industrial organizations. Of these, we selected 15 talks to present at the conference.

For the traditional refereed papers track, we received a near-record number of paper registrations and submissions this year. Authors registered 312 abstracts, of which 221 were submitted as complete papers. Of the submitted papers, 32 were short papers, which had to be at most five pages long plus references, and the other 189 were full-length papers, which had to be at most 11 pages long plus references.

Reviewing was single-blind, done by the program committee in two rounds, with a few external reviews. In the first round, each of the 221 submitted papers received two reviews. Papers receiving at least one "weak accept" or better (i.e., "accept" or "strong accept") review moved on to the second round. In total, 139 papers moved on, and 82 papers were tentatively rejected. In the second round, each paper received two more reviews on average. Altogether, 724 reviews were completed.

After two phases of review, an online discussion was conducted among reviewers, during which the program committee decided to accept 14 highly ranked papers and to further discuss 61 papers during the in-person program committee meeting. The meeting was held in April in Chicago. One PC member called in; the other members all attended in person. Over a period of nine hours, the committee decided to accept 35 papers, including the 14 papers that were accepted earlier. Among these 35 papers, two are short papers. In addition, a total of 15 papers entered the reject-resubmission process. These 15 papers appeared to contain interesting ideas but could not be accepted in their submitted form. Therefore, the authors were given the option to resubmit a revised version of the paper within one month. Each such paper was assigned a PC contact who helped the authors understand the committee concerns. For 14 out of these 15 papers, the authors took advantage of this option and resubmitted. The original reviewers read and re-reviewed each resubmitted version over a period of two weeks and decided to accept 12 out of 14 resubmissions. For most of these papers, all original reviewers were engaged in evaluating the resubmitted version.

The committee was comprised of 30 members, including the two co-chairs. Thirteen of them were affiliated with industrial organizations, and 17 were affiliated with academic organizations. Program committee members were allowed to submit papers. The two co-chairs did not submit any papers. We followed conventional rules for handling conflicts of interest: conflicted members (or co-chairs) left the room during discussion of conflicted papers.

In addition to the authors that submitted their work for consideration, the program committee, and the external reviewers, we would like to thank the USENIX staff that took care of all organizational details. Their help made our jobs a lot easier and allowed us to focus on reviewing papers and putting together the technical program.

We hope that you enjoy the conference, and thank you for participating in the USENIX ATC community.

Shan Lu, *University of Chicago*
Erik Riedel, *EMC*
USENIX ATC '15 Program Co-Chairs

Spartan: A Distributed Array Framework with Smart Tiling

Chien-Chin Huang[†], Qi Chen[*], Zhaoguo Wang[†], Russell Power[†], Jorge Ortiz[‡]
Jinyang Li[†], Zhen Xiao[*]

[†]*New York University,* [*]*Peking University,* [‡]*IBM T.J. Watson Research Center*

Abstract

Application programmers in domains like machine learning, scientific computing, and computational biology are accustomed to using powerful, high productivity array languages such as MatLab, R and NumPy. Distributed array frameworks aim to scale array programs across machines. However, maximizing the locality of access to distributed arrays is an unsolved problem; such locality is critical for high performance. This paper presents Spartan, a distributed array framework that automatically determines how to best partition (aka "tile") n-dimensional arrays and to co-locate data with computation to maximize locality. Spartan combines a lazy-evaluation based, optimizing frontend with a distributed tiled array backend. Central to Spartan's design is a small number of carefully chosen parallel *high-level operators*, which form the expression graph captured by Spartan's frontend during runtime. These operators simplify the programming of distributed applications. More importantly, their well-defined semantics allow Spartan's runtime to calculate the costs of different tiling strategies and pick the best one for evaluating the entire expression graph.

Using Spartan, we have implemented 12 applications from a variety of domains including machine learning and scientific computing. Our evaluations show that Spartan's automatic tiling mechanism leads to good and scalable performance while eliminating the need for manual tiling.

1 Introduction

High productivity array-languages, such as MATLAB [42], NumPy [51] and R [63], are the dominant toolkit for application programmers in areas like machine learning, scientific computing and computational finance. To help array programs scale across machines, there have been many proposals from both the HPC and the systems communities to develop a distributed array framework (discussed in §6). However, despite these efforts, an easy-to-use, high-performance distributed array framework has remained elusive. When distributing array programs, the open challenge is how to maximize the locality of access to array data spread out across the memory of many machines. To improve locality, one needs to both partition arrays smartly and co-locate computation with data. We refer to this as the "tiling" problem. Tiling is crucial for performance; programs that optimize for locality can be an order of magnitude faster than those that don't.

Existing distributed array frameworks do not adequately address the tiling problem. Most systems rely on users to manually specify array partitioning; examples include Pydron [46], Presto [64], MadLINQ [56], Global Arrays [20] and Petsc [9]. Although SciDB [62] can automatically choose a good chunk size to optimize loading arrays from disk, it still relies on a user-defined tiling strategy. Manual tiling can achieve good locality, but makes the resulting system much more tedious and complex to use than their single-machine counterpart. Ideally, a distributed array framework should support automatic tiling with minimal user input to achieve both ease-of-use and high performance.

This paper presents Spartan distributed array framework with smart tiling. Spartan provides the popular Numpy [51] array abstractions while achieving scalable high performance across machines. The key innovation of Spartan is its automatic tiling mechanism: when distributing an n-dimensional array across machines, the runtime of Spartan can automatically decide which axis(es) to cut each array along and to co-locate computation with data.

A major design of Spartan is the five high-level parallel operators, including map, fold, filter, scan and join_update. These high-level operators capture the parallel patterns of most array programs and we use them to distribute a myriad of built-in array functions as well as user programs. The semantics of these high-level operators lead to well-defined cost profiles. The cost profile of an operator gives an estimate of the communication cost for each potential tiling strategy (row-wised, column-wised, etc.) for its inputs. Therefore, it provides crucial information to enable the runtime to perform automatic tiling. As an example, the map operator applies a user-defined function element-wise to several input arrays with the same shape. Thus, this operator achieves the best locality (and zero communication cost) if all its input arrays are partitioned in the same way. Otherwise, the cost equals to the size of those input arrays with different tiling.

At runtime, Spartan splits program execution into a

series of frontend and backend steps. On the client machine, the frontend first turns a user program into an expression graph of high-level operators via lazy evaluation. It then runs a greedy search algorithm to find a good tiling for each node in the expression graph to reduce the overall communication cost. Finally, the frontend gives the tiled expression graph to the backend for execution. The backend creates distributed arrays according to the assigned tiling and evaluates each operator by scheduling parallel tasks among a collection of workers.

Spartan's automatic tiling is not without limitations. First, Spartan only aims to minimize network communication and does not consider other performance limiting factors such as how tiling impacts each machine's cache locality. Second, the default cost profile for join_update is not precise in some circumstances and require additional hints from users. While this imposes additional work from users, we have found the efforts to be reasonably low in practice. Third, the greedy search algorithm does not guarantee optimal tiling because the underlying optimization problem is NP-complete.

We have built Spartan to provide similar user interfaces as NumPy. It currently implements 50+ common Numpy functions. We have developed 12 applications on top of Spartan. All of them are simple to write using builtins or Spartan's high-level operators. Evaluations on a local cluster and the Amazon EC2 show that Spartan tiling algorithm can automatically find good tiling for arrays and achieve good scalability. Compared to an existing in-memory distributed array framework, Presto, Spartan applications achieve a speedup of $1.7\times$.

2 Automatic Tiling Overview

The Setup. The Spartan system is comprised of many worker machines in a high speed cluster. Spartan partitions each global array into several tiles (sub-arrays) and distributes each one to a potentially different worker. We refer to the partitioning strategy as $tiling$. There are several ways to "tile" an array. For example, Fig. 1 shows the three tiling choices for a 2D array (aka matrix).

In Spartan, an array is created by loading data from an external storage or as a result of some computation. Spartan decides the tiling choice for the array at its creation time. What is a good tiling choice? We consider the best tiling as one that incurs the minimum communication cost when the array is used in a computation – workers fetch and write as few remote tiles as possible. In this section, we examine what affects good tiling and give an overview of Spartan's approach to automatic tiling.

2.1 What Affects Good Tiling?

Several factors affect the tiling choice for an array. These include how the computation accesses the array, the runtime information of the array and how the array is used

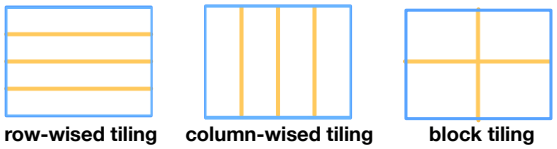


Figure 1: Three tiling methods for 2-dimensional arrays.

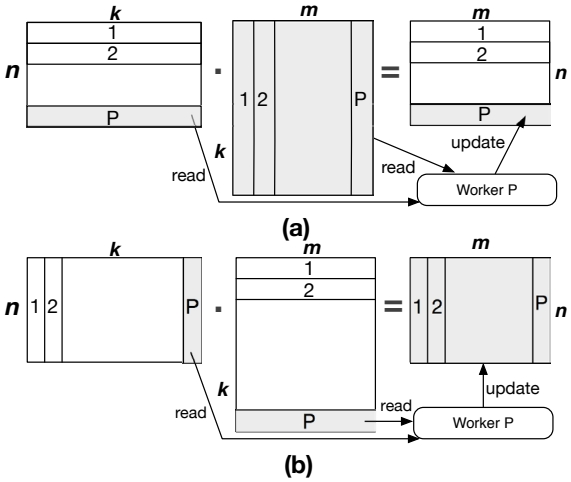


Figure 2: Two ways to implement matrix multiplication X·Y=Z, aka `dot` operation. Gray areas denote data read or updated by a single worker. In (a), each worker reads the entirety of Y across the network and performs local writes. Its per-worker communication cost is $k * m$. In (b), each worker performs local fetches and sends updates of size $n * m$ over the network. The per-worker communication cost is $n * m$.

across the program. Below, we illustrate how each of the factors affects tiling using concrete examples.

1) The access pattern of an array. Array computation tends to read or update an array along some particular axis. This access information is crucial for determining a good tiling. Fig. 2(a) shows the access pattern of a common implementation of matrix multiplication (aka `dot`). When computing $X \cdot Y = Z$, this implementation launches p parallel tasks each of which reads X row-wise and reads the entirety of Y. The task then performs a local dot and sends the result row-size to create Z. Consequently, it is best to tile both X and Z row-wise (it does not matter how Y is tiled). Other ways of tiling incur extra communication cost for fetching X and updating Z.

2) The shape and size of an array. The access pattern of an array often depends on the array's shape and size. Therefore, such runtime information affects the array's tiling choice. In addition to Fig. 2(a), there exists an alternative implementation of `dot`, shown as Fig. 2(b). In this alternative implementation, each of the p parallel tasks reads X column-wise and Y row-wise to perform a local matrix multiplication and update the entirety of Z. The final Z is created by aggregating updates from all p tasks. Consequently, it is best to tile X column-wise and Y row-wise.

Whether to use Fig. 2(a) or Fig. 2(b) to compute $X \cdot Y = Z$ is a runtime choice that depends on the array shapes. Suppose X is an $n \times k$ matrix and Y is a $k \times m$ matrix. Fig. 2(a) has a per task communication cost of $k * m$. This is because each task needs to fetch the entire Y across the network and can be scheduled to co-locate with the tile of X that it intends to read. By contrast, Fig. 2(b) has a per task communication cost of $n * m$. This is because each task needs to send its update of Z over the network and can be scheduled to co-locate with the tiles of X and Y that it intends to read. Therefore, the best tiling choice depends on the shape of X. If $n > k$, the cost of Fig. 2(a) is lower and the system computes `dot` using (a) whose preferred tiling for X is column-wise. If $n < k$, the cost of Fig. 2(b) is lower and the system computes `dot` using (b) whose preferred tiling for X is row-wise.

```
1  func ALS(A):
2    '''
3    Alternating Least Squares
4    Input: A is a n*k user-movie rating matrix.
5    Output: U and M are factor matrices.
6    '''
7    for i from 1 to max_iter
8        U = CalculateUsersFactor(A, M)
9        M = CalculateMoviesFactor(A, U)
10   endfor
11   return U, M
```

Figure 3: Pseudocode of Alternating Least Squares.

3) How an array is used throughout the program. An array can be read by multiple expressions. If these expressions access the array differently, we can reduce communication cost by creating multiple tilings for the array. In order to learn of an array's usage, the system cannot simply handle one expression at a time, but must "look ahead" in execution when determining an array's tiling. Consider the Alternating Least Squares (ALS) computation shown in Fig. 3. ALS solves the collaborative filtering problem by decomposing the given user-item rating matrix. Consider a movie recommendation system under ALS that makes use of two parameters: users and movies. In each iteration, ALS calculates the factor for each user, based on the rating matrix, A, and a movie factor matrix (line 5 in Fig. 3). Then, it calculates the factor for each movie based on the rating matrix, A, and users factor matrix (line 6 in Fig. 3). Thus, ALS needs to access A along both row (users) and column (movies) in one single iteration. If the system decides on A's tiling by line 8 only, it would tile A row-wise. Later, at line 9, the system incurs communication cost when reading A column-wise. This is far from optimal. If we unroll the for loop and look at all the expressions together, we can see that A is accessed by two expressions several times (max_iterations). Thus, the best tiling is to duplicate A and tile one along row and another along column.

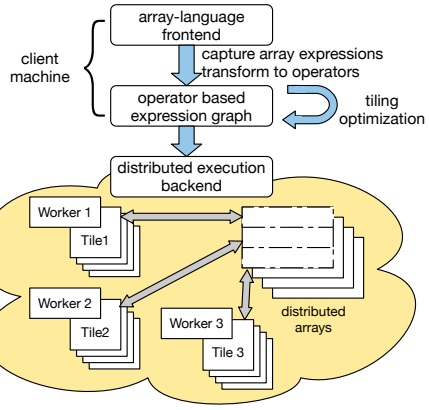


Figure 4: The layered design of Spartan. The frontend builds an expression graph and optimizes it. The backend executes the optimized graph on a cluster of machines. Each worker (3 workers in this figure) owns a portion of the global array.

2.2 Our Approach and Spartan Overview

Like NumPy and other popular array languages, users write applications in Spartan using a large number of built-in functions and array primitives (e.g. +,*,`dot`, `mean`, etc.). Spartan implements its built-in functions using a small number of *high-level parallel operators*. The high-level operators encapsulate common parallel patterns and can efficiently express most types of computation. Users may also directly program using these high-level operators if their computation cannot be expressed by existing builtins.

Spartan uses a layered approach which splits the execution into frontend and backend steps, shown in Fig. 4. The frontend, running on a client machine, captures user code and turns it into an expression graph whose nodes correspond to the high-level operators. Next, the frontend runs a tiling optimizer to determine good tiling for each node in the expression graph. Finally, the frontend sends the tiled expression graph to the backend. The backend provides high performance distributed implementations of high-level operators. For each operator, it schedules a collection of tasks running on many compute machines. The tasks create, fetch and update distributed in-memory arrays based on the tiling hint determined by the optimizer.

Spartan's high-level operators and its layered design help collect the necessary information for automatic tiling. First, by expressing various types of computation in a small set of high-level operators, the data access pattern is made explicit for analysis (§2.1 (1)). Second, the frontend dynamically captures the expression graph with runtime information about the shape of input and intermediate arrays (§2.1 (2)). Third, the expression graph represents a large execution context, thereby allowing the frontend to understand how an array is used by multiple

expressions. This is crucial for good tiling (§2.1 (3)).

3 Smart Tiling with High-level Operators

This section describes the design of Spartan, focusing on those parts crucial for automatic tiling. Specifically, we discuss high-level operators (§3.1), how Spartan's frontend turns an array program into a series of expression graphs (§3.2), the basic tiling algorithm (§3.3) and additional optimizations (§3.4).

3.1 High-level Operators

A high-level operator in Spartan is a parallel computation that can be parameterized by some user-defined function [1]. The operators are "functional" in nature: they take arrays or views of arrays as input and generate a new one without modifying existing arrays in place. Spartan supports views of arrays like NumPy. A view is an interface that allows users to manipulate arrays (e.g., swapping axes, slicing) without copying data. When reading a tile of a view, Spartan translates the shape and location from the view to those of the underlying array to fetch data.

High-level operators are crucial to Spartan's smart tiling, but what operators should we use? There are two considerations in choosing them. First, each operator should capture a general parallel pattern that can be used to implement many builtins. Second, each operator should have restricted semantics that correspond to a well-defined cost profile for different ways of tiling its input and output. This enables the captured expression graph to be analyzed to identify good tiling choices.

Spartan's current collection of five high-level operators is the result of many design iterations based on our experience of building various applications and builtins. Below, we describe each operator in turn and also discuss its (communication) cost w.r.t. different tiling choices.

- D=map$(f_{map}, S_1, S_2, \ldots)$ applies function f_{map} in parallel tile-wise over input arrays, $S_1, S_2, \ldots$, and generates output array D with the same shape. The total cost is zero if all inputs have the same tiling. Otherwise, the cost is the total size of all input arrays whose tiling differs from S_1.
  As an example usage of map, Fig. 5(line 4–7) shows the implementation of Spartan's built-in array addition function which simply uses map with f_{map} as Numpy's addition function.
- D=filter(f_{pred}, S) creates a view of S that excludes elements that do not satisfy the given predicate f_{pred}. Alternatively, filter can take a boolean array in place of f_{pred}. Since filter creates a view without copying actual data, the cost is zero.
- D=fold$(f_{accum}, S, axis)$ aggregates input array S using the commutative and associate function f_{accum} along the $axis$ dimension. For example, if S is a $m \times n$ matrix, then folding it along axis=0 creates a vector of n elements. Spartan performs the underlying folding in parallel using up to m tasks. The cost of fold is zero if S is tiled along the $axis$ dimension, otherwise, the cost is $S.size$.
- D=scan$(f_{accum}, S, axis)$ computes cumulative aggregates using f_{accum} over the $axis$ dimension of S. Unlike fold, its output D has the same shape as the input. The cost profile of scan is the same as fold.
- D=join_update$(f_{join}, f_{accum}, S_1, S_2, \ldots, axis_1, axis_2, \ldots, output_shape)$ is more complex than previous operators. This operator treats each input array S_i as a group of tiles along the $axis_i$, The shapes of the input arrays must satisfy the requirement that they have the same number of tiles along their respective $axis_i$. Spartan joins each tile among different groups and applies f_{join} in parallel. Function f_{join} generates some update to be written to output D at a specified location. Multiple workers running f_{join} may concurrently update to the same location of D; such conflicts are automatically resolved by applying f_{accum}.
  As an example of join_update, consider the matrix multiplication implementation in Fig. 2(b), where S_1 is a $n \times k$ matrix and S_2 is a $k \times m$ matrix. Fig. 5 (lines 20–22) uses join_update which divides S_1 into k column vectors and S_2 into k row vectors. The f_{join} (aka `dot_udf`) is called in parallel for each column vector of S_1 joined with the corresponding row vector of S_2. It performs a local dot product of the joined column and row to generate an $n \times m$ output tile. All updates are aggregated together using the addition accumulator to create the final output.
  A special case of join_update is when some input array S_i has $axis_i = -1$. In this case, the entire array S_i will be joined with each tile of other input arrays. Fig. 5 (lines 23-25) uses this special case of join_update to realize the alternative matrix implementation of Fig. 2(a).
  The cost of join_update consists of two parts, 1) the cost to read the input arrays. 2) the cost of updating the output array. If an input array S_i is partitioned along $axis_i$, the input cost for S_i is zero, otherwise, the cost is $S_i.size$. Since the size and shape of output array created by f_{join} is unknown to Spartan, it assumes a default update cost, $D.size$.

In addition to the five high-level operators, Spartan also provides several primitives to create distributed arrays or views of arrays.

[1] The user-defined function must be free of side-effects and deterministic.

- D=newarray$(shape, init_method)$ creates a distributed array with a given shape. The array can be initialized in several ways, 1) by loading data from an external storage, 2) by some computation, e.g. random, zeros.
- D=slice$(S, region)$ creates a view over a specified region in array S. The $region$ descriptor specifies the start and end of the sliced region along each dimension.
- D=swapaxis$(S, axis_1, axis_2)$ creates a view of array S by swapping the axes $axis_1$ and $axis_2$. The commonly used built-in transpose function is implemented using this operator. The output view D has a different tiling from S. For example, if S is a column-tiled matrix, then $D = swapaxis(S, 0, 1)$ is effectively a row-tiled matrix.

There is no cost for newarray, newarray and swapaxis (the cost of newarray reading from an external storage is unrelated to tiling).

```
1  import numpy
2  import spartan
3
4  # Spartan's parallel implementation of
5  # element-wise array addition
6  def add(a, b):
7      return spartan.map(a, b, f_map=numpy.add)
8
9  # User-defined f_join function
10 def dot_udf(input_tiles):
11   output_loc = spartan.location(0,0)
12   output_data = numpy.dot(input_tiles[0],
13                           input_tiles[1])
14   return output_loc, output_data
15
16 # Spartan's parallel implementation of
17 # matrix multiplication
18 def dot(a, b):
19   if a.shape[0] <= a.shape[1]:
20     return spartan.join_update(S=(a, b),
21                   axes=(1, 0), f_join=dot_udf,
22                   shape=..., f_accum=numpy.add)
23   else:
24     return spartan.join_update(S=(a, b),
25                                axes=(0, -1),..)
```

Figure 5: Implementations of add and dot in Spartan.

Based on the high-level operators, Spartan supports 50+ Numpy builtins. Fig. 5 shows two implementations of Spartan's builtins, add and dot.

Although Spartan's map and fold resemble the "map" and "reduce" primitives in the MapReduce world [21, 1, 67, 29], they are more restrictive. Spartan only allows f_{map} to write a tile in the same location of the output array as its input tile location and not some arbitrary location. Similarly, fold can only reduce along some $axis$ as opposed to over arbitrary keys in a key value collection. Such restriction is necessary for them to have a well-defined cost profile.

3.2 Expression Graph Capture

During a user program's execution, Spartan's frontend captures array expressions via lazy evaluation and turns

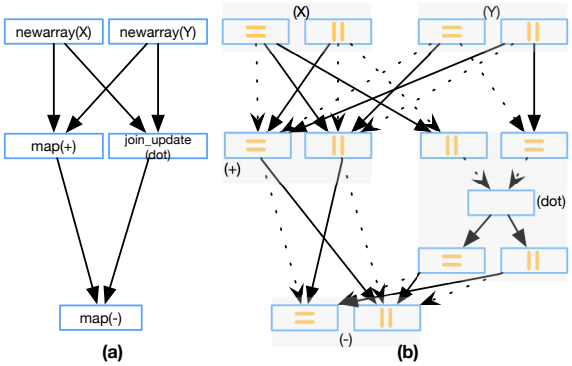


Figure 7: The expression graph and its corresponding tiling graph for $Z = X + Y - X \cdot Y$.

them into a series of expression graphs [15, 4]. In an expression graph, each node corresponds to a high-level operator and an edge from one node to another shows the data dependency between them. Fig. 7(a) shows an example expression graph. Expression graphs are acyclic because Spartan's high-level operators create immutable arrays.

The frontend stops growing an expression graph only when forced: this occurs in a few situations: (1) when a variable is used to determine the control flow, (2) when a variable is used for program output, (3) when a user explicitly requests evaluation. The use of lazy evaluation leads to an implicit form of loop unrolling: as long as there is no data dependent control flow, expression graph will continue growing until pre-configured limits.

3.3 Graph-based Tiling Optimizer

Spartan supports "rectangular" tiles: an n-dimensional array can be partitioned along any one dimension (e.g. row-wise, column-wise), or partitioned along two or more dimensions (e.g. block-wise tiling). Some existing work [28] explored other possible shapes that are more efficient for its applications.

Given an expression graph of high-level operators, the goal of the tiling optimizer is to choose a tiling for each operator node to minimize the overall cost. This optimization problem is NP-Complete (See appendix §A). It is also not practical to find the best tiling via brute force since the expression graph can be very large. Therefore, we propose a graph-based approximation algorithm to identify a good tiling quickly.

The algorithm works in two stages. First, it constructs a tiling graph based on the expression graph and the cost profile of each operator. Next, it uses a greedy strategy to search for a low cost tiling combination.

1) Constructing the tiling graph. The goal of the tiling graph is to *expose the tiling choices and cost in the expression graph*. For each operator in the expression graph, the optimizer transforms it into a *node group*, i.e.

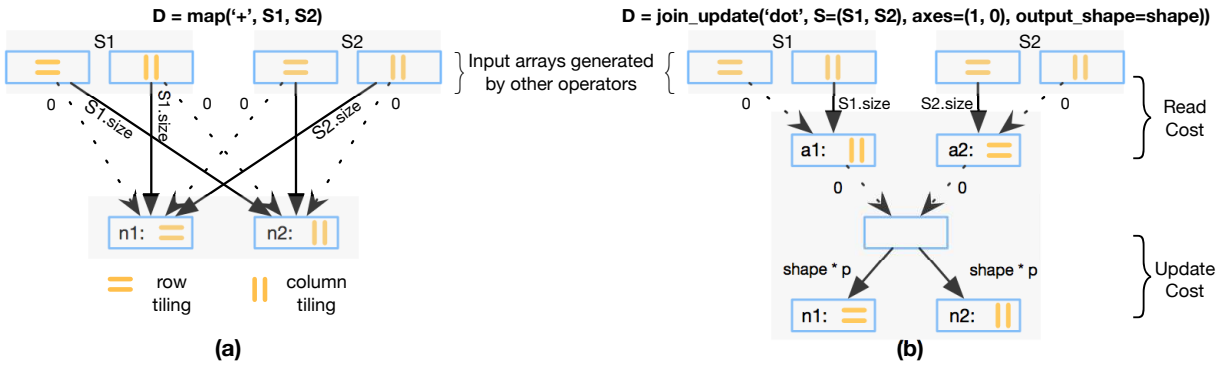


Figure 6: Two examples of building the tiling graph. (a) A plus expression, (S1 + S2), implemented by map operator (b) A dot expression, dot(S1, S2), implemented by join_update operator.

a cluster of several tiling nodes, each representing a specific choice to tile the operator's output or intermediate steps. The weight of each edge that connects two tiling nodes represents the underlying cost if the two operators are tiled according to the tiling nodes.

Fig. 6 shows how a map operator, corresponding to $D = S_1 + S_2$, is transformed. To keep the figure simple, we assume that all arrays are two dimensional with two tiling choices: row-based or column-based. And all dotted lines represent zero edge weights. As Fig. 6 shows, the map operator becomes two nodes in the tiling graph, each representing a different way to tile its output D. Similarly, each of the map operator's input arrays S_1 and S_2 (which are likely outputs from the previous operators) also correspond to two nodes. For map, there is a well-defined way to label the weights among nodes, as illustrated in Fig. 6. For example, if S_2 is tiled column-wise and D is tiled row-wise, the weight between the corresponding two nodes is S_2.size because workers have to read S_2 across the network to perform the map. fold and scan are treated similarly as map, but with edge weights labeled according to their own tiling cost profiles.

Next, we discuss the transformation of join_update. For this operator, we use some intermediate tiling nodes ($a_1, a_2 \ldots$ in Fig. 6(b)) to represent the reading cost during the join. A placeholder node is used to represent the join stage. We use another set of tiling nodes ($n1, n2$ in Fig. 6(b)) to capture the update cost to the output array. Unfortunately, Spartan can not know the precise update cost of join_update without executing the user-defined f_{join} function. Thus, we provide a default update cost according to the common update cost pattern observed in the applications implemented by join_update. If join_update is performed within a loop, the optimizer can adjust the edge cost of the tiling graph according to the actual cost observed during the previous execution of the join_update.

Fig. 6(b) shows the tiling graph used for the matrix multiplication function implemented in join_update. This implementation corresponds to the data access pattern shown in Fig. 2(b). As shown in Fig. 5, the join axes for the first and second arrays are column and row respectively. The edge weight for S_i is 0 if it matches the join axis and is $S_i.size$ otherwise. The cost is $S_i.size$ is because each worker needs to update the entirety of the result matrix. The edge weights for n_1 and n_2 are both $p * output_shape$.

Fig. 7 gives an example showing a specific array execution ($Z = X + Y - X \cdot Y$)) and its corresponding expression graph and tiling graph. We omitted the details of other edge weights to keep the graph readable.

2) Searching for a good tiling. Deciding a tiling choice for an operator corresponds to picking one node among the corresponding node group in the underlying tiling graph and different combinations of tiling nodes pose different costs. As a result, the next step for the tiling optimizer is to analyze the tiling graph and find a combination of tiling choices that minimizes the overall cost. The tiling optimizer adopts a greedy search algorithm. The heuristic is to decide the tiling for the node group with the maximum connectivity first. Here, connectivity of a node group is the number of its adjacent node groups. When deciding a tiling for a node group X, the algorithm chooses the one resulting in the minimum cost for X. Why does this heuristic work? The cost of a tiling for an operator depends on the tiling choices of its adjacent operators. Thus, an operator with more adjacent operators has a higher impact on overall cost. Consequently, the algorithm should first minimize the cost of node groups with higher connectivity[2].

Fig. 8 shows the pseudo code for the tiling algorithm. Given a tiling graph G, the algorithm processes node groups in the order of edge connectivity (Line 19–20). For each node group (x in Line 20), the algorithm calculates the cost of each tiling node and chooses the tiling

[2]Another natural heuristic is to search the node group with largest array size first. Unfortunately, this algorithm does not perform well according to our experiments.

node with the minimum cost (Line 23–29). After deciding the good tiling ($x.chosenTiling$ in Line 30) for node group x, the algorithm removes all edges connected to all other tiling nodes (Line 32). This implies that the algorithm can't freely choose tiling for adjacent node groups of x any more – it must consider the chosen tiling of x.

$FindCost$ obtains the cost of a tiling node (T in Line 1) by calculating the sum of the minimum edge weight between each adjacent node group and T (Line 4–14). If the adjacent node group is a view operator such as swapaxis, its tiling node will be decided by T. To get accurate cost affected by T, the algorithm should also consider the adjacent node groups for its view operators. As a result, $FindCost$ recursively finds the cost of the view node group (Line 5–6). The result corresponds to the best possible cost for tiling node T.

The complexity of the tiling algorithm is $O(E * N)$ where E is the number of edges in the tiling graph and N is the number of node groups. It is not guaranteed to find the optimal tiling. However, we find that the greedy strategy works well in practice (§5).

```
1  func FindCost(NodeGroup G, TileNode T)
2    # Find the cost for tiling node T of G
3    cost = 0
4    foreach NodeGroup g in G.connectedGroups():
5      if IsView(g, G):
6        cost += FindCost(g, g.viewTileNode(T))
7      else:
8        edgeCost = INFINITY
9        foreach Edge e in g <-> T
10         edgeCost = min(edgeCost, e.cost)
11       endfor
12       cost += edgeCost
13     endif
14   endfor
15   return cost
16
17 func FindTiling(TilingGraph G)
18   # Find good tiling for every operator in G.
19   GroupList = SortGroupByConnectivity(G)
20   foreach NodeGroup x in GroupList
21     minCost = INFINITY
22     goodTiling = NONE
23     foreach TileNode y in x
24       cost = FindCost(x, y)
25       if cost < minCost:
26          minCost = cost
27          goodTiling = y
28       endif
29     endfor
30     x.chosenTiling = goodTiling
31     # Other Group can only connect to goodTiling.
32     x.removeAllConnectedEdgesExcept(goodTiling)
33   endfor
34   return G
```

Figure 8: The maximum connectivity group first algorithm to find good tiling based on the tiling graph.

3.4 Additional Tiling Optimizations

Duplication of arrays. As the ALS example in Fig 3 shows, some arrays may be accessed along different axes several times. To reduce communication, Spartan supports duplication of arrays and tiles each replica along different dimensions. To support duplication in the tiling optimizer, we add a "duplication tile" node to each node group in the underlying tiling graph. As duplication of arrays increases memory consumption. Spartan allows users to specify the memory budget for duplicating arrays to limit memory usage. Whenever the optimizer chooses to "duplicate tile" which causes an operator's output to be duplicated, it deducts from the memory budget. The optimizer will not choose duplication tiling without enough memory budget.

Sparse arrays. Dense arrays and sparse arrays are different in several aspects. First, the size of a sparse array can't be known based on the shape. Smart tiling estimates the size by sampling before constructing the tiling graph. Second, the non-zero elements distribution of intermediate arrays may be different from those of the input arrays. Smart tiling addresses this problem by adjusting edge weights after executing operators. This technique is the same as how Spartan improves its initial imprecise cost estimate of join_update with successive execution. Finally, the distribution of a sparse array can be skewed. Smart tiling can use fine-grained tiles to help backend to perform work stealing [55].

4 Implementation

Since NumPy is wildly popular in machine learning and scientific computing, our implementation goal is to replicate the "feel" of NumPy as much as possible. Our prototype currently supports 50+ most commonly used Numpy builtins.

The Spartan frontend, written in Python, captures expression graph and performs tiling optimization (§3). The Spartan backend, consists of one designated master and many worker processes on a cluster of machines. Below, we provide more details on the major backend components:

Execution engine. The backend provides efficient implementations of all high-level operators. Given an expression graph, the master is responsible for coordinating the execution of one node (a high-level operator) at a time. To execute a node, the master first creates an output array with the given tiling hint and then schedules a set of tasks to run user-defined parameter functions in parallel according to the data locality. Locality here means the task is executed on the worker that stores its input source tile. If the node corresponds to a join_update, scan or fold, the backend also associates a user-defined accumulator function with the output array to aggregate updates from multiple workers.

User-defined parameter functions are written in Python NumPy and process one tile instead of one element at a time. Like MatLab, NumPy relies on high performance C-based linear algebra libraries like BLAS [35] or LAPACK [6]. As a result, the local execution of parameter functions in each worker is efficient.

Distributed, tiled arrays. Each distributed array is partitioned into a set of tiles according to its tiling hint and stored in workers' memory. To create an array, the master assigns each of its tile to a worker (e.g. in a round-robin fashion) and distributes the tile-to-worker mapping to all workers so everybody can access remote tiles without consulting the master. If two arrays of the same shape have identical hints, the master ensures that tiles corresponding to the same region in both arrays are co-located in the memory of the same worker.

Fault tolerance. To recover from worker failure in the middle of a long computation, the backend checkpoints in-memory arrays to durable storage. Our implementation currently adopts the simplest design: after finishing an entire operator, the master periodically instructs all workers to save their tiles and also saves its own state.

5 Evaluation

In this section, we measured the performance of our smart tiling algorithm. We also evaluated the scalability of applications and compared against other open-source distributed array frameworks.

5.1 Experimental setup

We evaluated the performance of Spartan on both our local cluster as well as Amazon EC2. The local cluster is a heterogeneous setup consisting of eleven machines: 6 machines have 8-core AMD Opterons with 16GB of RAM, and 5 machines have 4-core Intel Xeons with 8GB of RAM. The machines are connected by gigabit Ethernet. For the EC2 experiments, we use 128 spot instances of the older generation m2.xlarge. Each of these instances has 17.1GB memory and 2 virtual CPUs. The network performance is rated as "moderate", which is approximately 300Mbps according to our measurements.

Unless otherwise mentioned, we ran multiple worker processes on each machine, one associated with each CPU core. We use 12 applications as our benchmarks. They include algorithms from machine learning, data mining and computational finance.

5.2 Tiling

Smart Tiling Evaluation for Applications: We compared the running time of applications with the tiling generated by smart tiling against the best tiling – the tiling that incurs the minimum communication cost. The best tiling can be pre-calculated by using a brute-force algorithm to traverse the expression graph and search the minimum communication cost among all possible tiling choices. The experiment runs on 128 EC2 instances. Fig. 9 only shows 10 applications because the computational finance ones operate on one-dimensional arrays which can only be tiled along one axis. For applications which are not perfectly scalable such as ALS and Cholesky, we set the sample sizes up to 10 million. For others, the sample sizes are up to 1 billion due to the memory limitation.

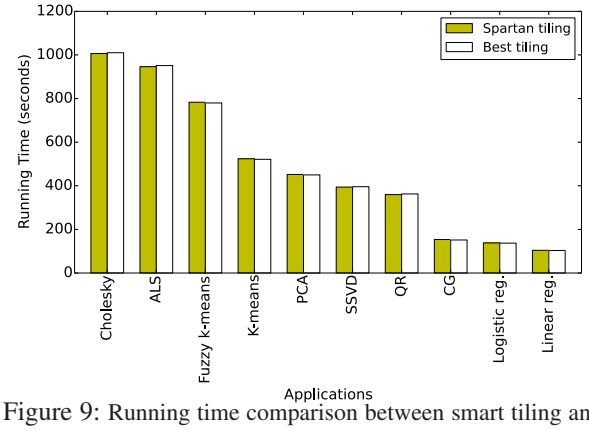


Figure 9: Running time comparison between smart tiling and the best tiling for 10 applications.

These applications show various kinds of tiling patterns. First, many applications contain expressions or operators that require runtime shape and axis information to best tile matrices, e.g. dot and join_update. Smart tiling analyzes the runtime information and gives the best tiling for the applications such as row-wise tiling for Regression and block tiling for Cholesky decomposition. Second, some program flows pass the intermediate matrices to expressions that change the view of tiling, e.g. swapaxis. Smart tiling identifies the best tiling through the global view of computation. Example applications include SSVD and PCA. Finally, some applications, like ALS, access matrices along different axes several times. As described in §2.1, the best tiling for these applications is duplication tiling.

Fig. 9 shows that Spartan's smart tiling is able to give the best tiling and improve the performance for all applications. Note that the application running time of the best tiling and Spartan's smart tiling are not the same; sometimes Spartan's smart tiling even outperforms the best tiling. The difference is caused by the instability of Amazon EC2. Spartan's optimizer makes the same choices as the best tiling for all applications.

A bad tiling can result in huge network transmission. For instance, if the tiling of the input arrays for logistic regression is partitioning along the smaller dimension, workers need to remotely fetch the matrix which is more than 512GB in the evaluation (4GB network transmission per instance in one iteration which result in approximately an extra 110 seconds in our environment). Another interesting example is ALS. Simply row-wise or column-wise tiling can result in 40% performance degradation compared to duplication tiling. Moreover, the running speed of smart tiling is fast. For example, the brute-force algorithm needs more than 500 seconds to analyze a 14-operators ALS while Spartan's smart tiling derives the same result in 0.06 seconds.

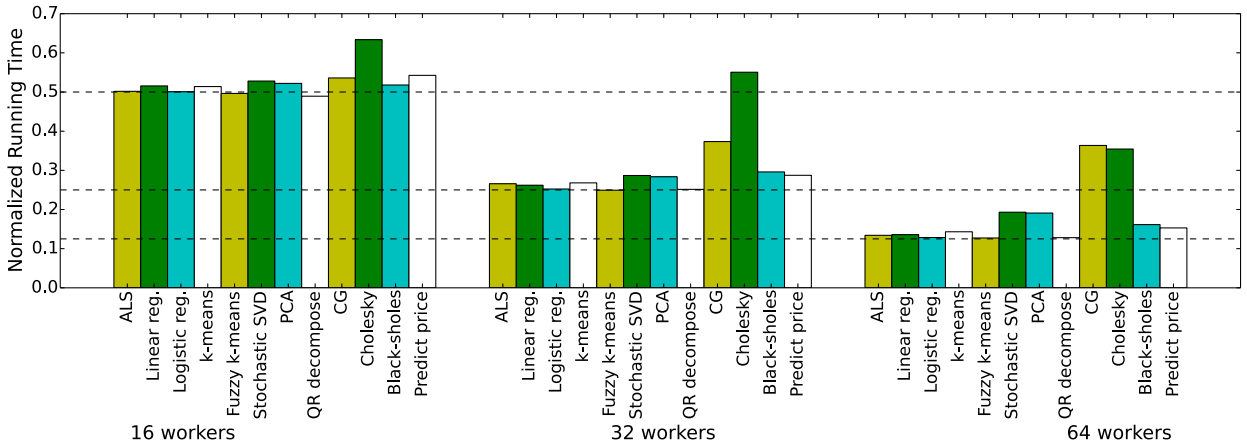


Figure 12: Fixed input size, varying number of workers. Normalized running time is calculated by dividing 8 worker running time on local cluster.

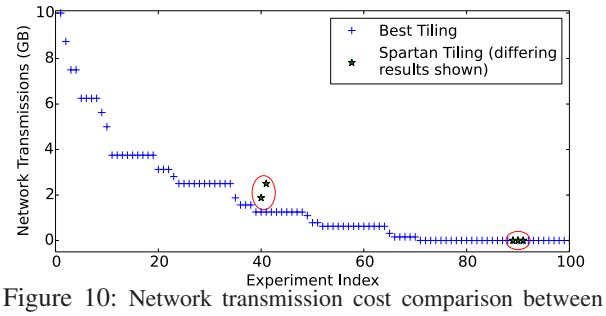


Figure 10: Network transmission cost comparison between smart tiling and the best tiling for 100 randomly generated programs. Sorted by network transmission for readability only (array sizes are randomly chosen from a set and there is no relation between experiment index and network transmission).

```
1  def sub_optimal_case_pattern(SIZE):
2      A = expr.rand((SIZE, SIZE))
3      B = expr.rand((SIZE, SIZE))
4      C = A + B
5      D = expr.transpose(A) + expr.transpose(B)
6      E = C + D
```

Figure 11: An example that smart tiling gives sub-optimal tiling.

Smart Tiling Evaluation for Randomly Generated Programs: Although smart tiling gives the best tiling for applications we implemented, there is no guarantee that smart tiling performs well for various kinds of applications. Therefore, we examined the performance of smart tiling for randomly generated programs. Each array dimension is randomly chosen from 128K to 512K. These programs contain various numbers and types of operators Spartan has supported. The number of operators per program ranges from 2 to 15.

Fig. 10 shows the network transmission cost of 100 randomly generated programs with the tiling given by smart tiling and the best tiling. The result shows that Spartan's smart tiling can give the best tiling for most programs. It is also fast compared to the brute-force algorithm. For all programs, smart tiling needs less than 0.1 seconds while the brute-force algorithm spends 1900 seconds when the program contains 15 operators.

Fig. 11 shows the pattern residing in those programs that smart tiling gives sub-optimal tiling. The best tiling for Fig. 11 is to tile D column-wise and other operators row-wise. However, smart tiling inspects the tiling cost for C first and then for D because of the maximum connectivity. It finds that row-wise tiling costs zero for both operators. Therefore, smart tiling partitions both C and D row-wise and thus gives sub-optimal tiling due to the conflict views (caused by transpose) of C and D.

Although smart tiling cannot give the best tiling for these programs, this sub-optimal case rarely happens. Smart tiling produces a conflict view only when a program exhibits two patterns simultaneously: 1) Two operators have different views of tiling from the same input arrays. 2) Both operators have more connectivity than their input arrays. As Fig. 10 shows, only 5 out of 100 random generated programs satisfy both requirements. For three of them, the best tiling needs zero network transmission while the smart tiling needs around 0.01 GB network transmission. The number is not large because these expressions include fold which reduces the size of matrices. For the other two instances, the best tiling requires 1.3 GB but the smart tiling consumes 1.9GB and 2.6GB respectively.

5.3 Scaling

We evaluated the scalability of all applications in two ways. First, the applications use fixed-size inputs and run across a varying number of workers. Second, the applications use inputs whose sizes are scaled linearly with the number of workers. All results are normalized by the 8 workers baseline cluster size to show the relative savings (comparing with 1 worker is not fair because there is no communication for only 1 worker). All inputs are synthetic data.

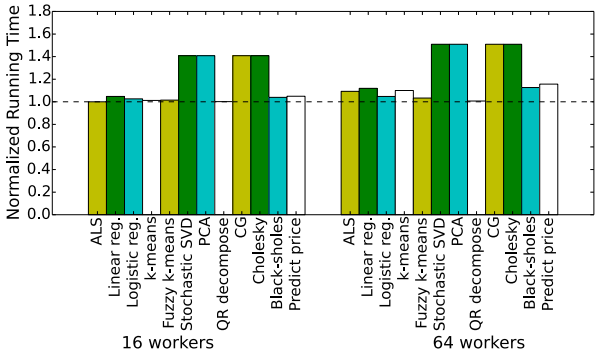


Figure 13: Scaling input size on local cluster.

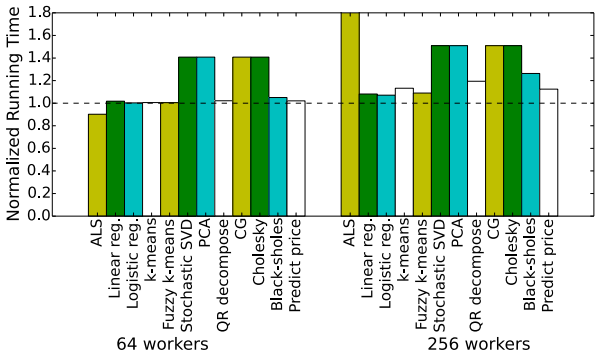


Figure 14: Scaling input size on 128 instances EC2.

Fixed input size. Fig. 12 shows the running time of 12 applications on the local cluster. The number of workers used in the experiments increases from 8 to 64. The dotted lines corresponding to $\frac{1}{2}$, $\frac{1}{4}$ or $\frac{1}{8}$ ratio represent the ideal scaling for 16, 32, and 64 workers.

The evaluation shows that the running time of many applications achieves perfect scaling. Some of them do not scale well due to the inefficiencies of the underlying algorithms. CG has many dependent folds that reduce to one value on one worker. Cholesky also has many dependent steps: the parallelism available in each step grows and shrinks, thus Cholesky cannot always utilize all workers.

Scaling input size. Fig. 13 shows the performance for 16 and 64 workers. Ideal scaling corresponds to a flat line of 1.0.To examine the scalability on a larger-scale system, we ran the experiment on EC2. Fig. 14 illustrates the experiment running up to 256 workers. The result is similar to that of Fig. 13 except for ALS. There are three matrices in ALS, rating matrix, sample matrix and item matrix. While Spartan's smart tiling can reduce the reading cost of rating matrix by duplication, ALS still needs to randomly fetch sample matrix and item matrix in each iteration and results in large communication. Thus, ALS is not scalable for large-scale datasets.

	Running Time (seconds)	Sample Size
Spartan	523.95s	1 billion
Presto	882.47s	1 billion
SciDB	2573.83s	10 million

Figure 15: K-Means performance comparison with Presto and SciDB on 128 instances EC2. The dataset for Spartan and Presto contains 1 billion points, 50 dimensions and 128 centers. The dataset for SciDB contains 10 million points.

5.4 Comparison with other systems

We compared the performance of Spartan's k-means with the implementation of Presto (also called Distributed R) and SciDB. The synthetic dataset contains 1 billion samples with 50 dimensions and 128 centers for Presto and Spartan while only 10 million samples for SciDB.

Fig. 15 shows that the performance of Spartan is 1.7x faster than Presto. Though both Spartan and Presto partition the arrays row-wise which is the best tiling, Presto requires users to explicitly assign the tiling while Spartan needs no user hints. Thus, the performance difference of Spartan and Presto comes from the backend library and implementation. We have verified this by running k-means only on a single worker.

Unlike Spartan and Presto, SciDB is not an in-memory distributed system and thus has much slower performance. The basic partition unit in SciDB is a chunk. It is important for SciDB to select the correct chunk size to reduce disk I/O. However, in Spartan, we focus on how to reduce the network communication.

6 Related Work

There is much prior work in the area of distributed array framework design and optimization.

Compiler-assisted data distribution. Prior work in this space proposes static, compile-time techniques for analysis. The first set of techniques focuses on partitioning [28] and the latter set on data co-location [33, 53, 45]. Prior work also has examined nested loops with affine array subscript patterns, using different structures (vector [28], matrix [58] or reference [30]) to model memory access patterns or polyhedral model [40] to perform localization analysis. Since static analysis deals poorly with ambiguities in source code [7], recent work proposes profile-guided methods [18] and memory-tracing [52] to capture memory access patterns. Simpler approaches focus on examining stencil code [52, 24, 26, 32, 25]. Spartan simplifies analysis significantly since high-level operator access patterns are well-defined.

Access patterns can be used to find a distribution of data that minimizes communication cost [28, 57, 10, 22, 27]. All approaches construct a weighted graph that captures possible layouts. Although searching the optimal solution is NP-Complete [31, 34, 36, 37], heuristics per-

form well in practice [37, 53]. Spartan adopts the idea of constructing a weighted graph. However, unlike prior work that requires language-specific compile-tile analysis, Spartan's high-level operators with know tiling costs provide enough information to analysis.

Parallel vector languages. ZPL [38], SISAL [43], NESL [13] and MatLab*P [17] share a common goal with Spartan. These languages expose distributed arrays and vector primitives and some provide a few core operators for parallel operations. Unlike Spartan, ZPL does not allow arbitrary indexing of distributed arrays and does not allow parallelization of indexable arrays. NESL relies on a *PRAM* model which assumes that a shared, distributed region of memory can be accessed with low latency. Spartan makes no such assumption. SISAL provides an explicit tiled model for arrays [23], however does not consider tiling strategies.

Distributed programming frameworks. Most distributed frameworks target primitives for key-value collections (e.g. MapReduce [21], Dryad [29], Piccolo [55], Spark [67], Ciel [48], Dandelion [59] and Naiad [47]). Some provide graph-centric primitives (e.g. GraphLab [39] and Pregel [41]). While one can encode arrays as key-value collections or graphs, doing so is much less efficient than Spartan's tile-based backend. It is possible to implement Spartan's backend by augmenting an in-memory framework, such as Spark or Piccolo. However, we built our prototype from scratch to allow for better integration with NumPy.

FlumeJava [15] provides programmers with a set of high-level operators. Its operators are transformed into MapReduce's [21] dataflow functions. FlumeJava is targeted at key-value collections instead of arrays. FlumeJava's operators look similar to Spartan's, but their underlying semantics are specific to key-value collections instead of arrays. Moreover, FlumeJava does not explicitly optimize for data locality because it is not designed for in-memory computation.

Relational queries are a natural layer on top of key-value centric distributed execution frameworks, as seen in systems like DryadLINQ [66], Shark [65], Dandelion [59] and Dremel [44]. Several efforts attempt to build an array interfaces on these. MadLINQ [56] adds support for distributed arrays and array-style computation to the dataflow model of DryadLINQ [66]. SciHadoop [14] is a plug-in for Hadoop to process array-formatted data. Google's R extensions [61], Presto [64] and SparkR [3] extend the R language to support distributed arrays. Julia [2] is a newly developed dynamic language designed for high performance and scientific computing. Julia provides primitives for users to parallel loops and distribute arrays. These extensions and languages rely on users to specify a tiling for each array, which burdens users with making non-trivial optimization that require deep familiarity which each operation and its data.

Distributed array libraries. Optimized, distributed linear algebra libraries, like LAPACK [6], ScaLAPACK [16], Elemental [54] Global Arrays Toolkit [49] and Petsc [8, 9] expose APIs specifically designed for large matrix operations. They focus on providing highly optimized implementations of specific operations. However, their speed depends on correct partitioning of arrays and their programming model is difficult to extend.

Global Address Spaces. Systems such as Unified Parallel C [19] and co-array Fortran [50] provide a global distributed address space for sharing arrays. They can be used to implement the backend for distributed array libraries. They do not directly provide a fully functional distributed array language.

Specialized application frameworks. There are a number of frameworks specifically targeted for distributed machine learning (e.g. MLBase [60], Apache Mahout [5], and Theano [12], for GPUs). Unlike these, Spartan targets a much wider audience and thus must address the complete set of challenges, including support for a number built-ins, minimizing the number of temporary copies and optimizing for locality.

Array Databases and Query Languages SciDB [62] and RasDaMan [11] are distributed databases with n-dimensional data storage and an array query language inspired by SQL. These represent the database community's answer to big numerical computation. The query language is flexible, but as the designers of SciDB have seen, application programmers often prefer expressing problems in more comprehensive array languages. SciDB-R is an attempt to win over R programmers by letting R scripts access data in SciDB and use SciDB to execute some R commands. SciDB's partition strategy is optimized for disk utilization. In contrast, Spartan focuses on in-memory data.

7 Conclusion

Spartan is a distributed array framework that provides a smart tiling algorithm to effectively partition distributed arrays. A set of carefully chosen high-level operators export well-defined communication cost and simplify the tiling process. User array code is captured by the frontend and turned into an expression graph whose nodes correspond to these high-level operators. With the expression graph, our smart tiling can estimate the communication cost across expressions and find good tilings for all the expressions.

Acknowledgments: We thank the anonymous reviewers and our shepherd David Shue. This work is supported in part by NSF (CSR-1065169) and a Google research award.

References

[1] Apache hadoop. http://hadoop.apache.org.

[2] Julia language. http://julialang.org.

[3] Sparkr: R frontend for spark. http://amplab-extras.github.io/SparkR-pkg.

[4] Dataflow program graphs. *IEEE Computer 15* (1982).

[5] Mahout: Scalable machine learning and data mining, 2012. http://mahout.apache.org.

[6] ANDERSON, E., BAI, Z., DONGARRA, J., GREENBAUM, A., MCKENNEY, A., DU CROZ, J., HAMMERLING, S., DEMMEL, J., BISCHOF, C., AND SORENSEN, D. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (1990), IEEE Computer Society Press, pp. 2–11.

[7] ANDERSON, P. Software engineering technology the use and limitations of static-analysis tools to improve software quality, 2008.

[8] BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., RUPP, K., SMITH, B. F., AND ZHANG, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[9] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[10] BAU, D., KODUKULA, I., KOTLYAR, V., PINGALI, K., AND STODGHILL, P. Solving alignment using elementary linear algebra. In *Languages and Compilers for Parallel Computing*. Springer, 1995, pp. 46–60.

[11] BAUMANN, P., DEHMEL, A., FURTADO, P., RITSCH, R., AND WIDMANN, N. The multidimensional database system RasDaMan. In *ACM SIGMOD Record* (1998), vol. 27, ACM, pp. 575–577.

[12] BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (2010).

[13] BLELLOCH, G. E. NESL: A nested data-parallel language.(version 3.1). Tech. rep., DTIC Document, 1995.

[14] BUCK, J. B., WATKINS, N., LEFEVRE, J., IOANNIDOU, K., MALTZAHN, C., POLYZOTIS, N., AND BRANDT, S. Scihadoop: array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011).

[15] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI - ACM SIGPLAN 2010* (2010).

[16] CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the* (1992), IEEE, pp. 120–127.

[17] CHOY, R., EDELMAN, A., AND OF, C. M. Parallel matlab: Doing it right. *Proceedings of the IEEE 93* (2005), 331–341.

[18] CHU, M., RAVINDRAN, R., AND MAHLKE, S. Data access partitioning for fine-grain parallelism on multicore architectures. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on* (2007), IEEE, pp. 369–380.

[19] CONSORTIUM, U. UPC language specifications, v1.2. Tech. rep., Lawrence Berkeley National Lab, 2005.

[20] DAILY, J., AND LEWIS, R. R. Using the global arrays toolkit to reimplement numpy for distributed computation. In *Proceedings of the 10th Python in Science Conference* (2011).

[21] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (2004).

[22] D'HOLLANDER, E. Partitioning and labeling of index sets in do loops with constant dependence vectors. In *1989 International Conference on Parallel Processing, University Park, PA* (1989).

[23] GAUDIOT, J.-L., BOHM, W., NAJJAR, W., DEBONI, T., FEO, J., AND MILLER, P. The sisal model of functional programming and its implementation. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium* (1997), IEEE, pp. 112–123.

[24] HE, J., SNAVELY, A. E., VAN DER WIJNGAART, R. F., AND FRUMKIN, M. A. Automatic recognition of performance idioms in scientific applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 118–127.

[25] HENRETTY, T., STOCK, K., POUCHET, L.-N., FRANCHETTI, F., RAMANUJAM, J., AND SADAYAPPAN, P. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction* (2011), Springer, pp. 225–245.

[26] HERNANDEZ, C. K. O. Open64-based regular stencil shape recognition in hercules.

[27] HUANG, C.-H., AND SADAYAPPAN, P. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing 19*, 2 (1993), 90–102.

[28] HUDAK, D. E., AND ABRAHAM, S. G. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *ACM SIGARCH Computer Architecture News* (1990), vol. 18, ACM, pp. 187–200.

[29] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)* (2007).

[30] JU, Y.-J., AND DIETZ, H. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing*. Springer, 1992, pp. 344–358.

[31] KENNEDY, K., AND KREMER, U. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS) 20*, 4 (1998), 869–916.

[32] KESSLER, C. W. Pattern-driven automatic parallelization. *Scientific Programming 5*, 3 (1996), 251–274.

[33] KNOBE, K., LUKAS, J. D., AND STEELE JR, G. L. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing 8*, 2 (1990), 102–118.

[34] KREMER, U. Np-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands* (1993).

[35] LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS) 5*, 3 (1979), 308–323.

[36] LI, J., AND CHEN, M. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the* (1990), IEEE, pp. 424–433.

[37] LI, J., AND CHEN, M. The data alignment phase in compiling programs for distributed-memory machines. *Journal of parallel and distributed computing 13*, 2 (1991), 213–221.

[38] LIN, C., AND SNYDER, L. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*. Springer, 1994, pp. 96–114.

[39] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)* (2012).

[40] LU, Q., ALIAS, C., BONDHUGULA, U., HENRETTY, T., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., SADAYAPPAN, P., CHEN, Y., LIN, H., ET AL. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on* (2009), IEEE, pp. 348–357.

[41] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), ACM, pp. 135–146.

[42] MATHWORKS. MATLAB software.

[43] MCGRAW, J., SKEDZIELEWSKI, S., ALLAN, S., OLDEHOEFT, R., GLAUERT, J., KIRKHAM, C., NOYCE, B., AND THOMAS, R. *SISAL: streams and iteration in a single assignment language. Language Reference Manual*. 1985.

[44] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *VLDB* (2010).

[45] MILOSAVLJEVIC, I. Z., AND JABRI, M. A. Automatic array alignment in parallel matlab scripts. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings* (1999), IEEE, pp. 285–289.

[46] MÜLLER, S. C., ALONSO, G., AMARA, A., AND CSILLAGHY, A. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 645–659.

[47] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[48] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: a universal execution engine for distributed data-flow computing. NSDI.

[49] NIEPLOCHA, J., HARRISON, R. J., AND LITTLEFIELD, R. J. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing 10*, 2 (1996), 169–189.

[50] NUMRICH, R. W., AND REID, J. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum 17* (1998).

[51] OLIPHANT, T., ET AL. NumPy, a Python library for numerical computations.

[52] PARK, E., KARTSAKLIS, C., JANJUSIC, T., AND CAVAZOS, J. Trace-driven memory access pattern recognition in computational kernels. In *Proceedings of the Second Workshop on Optimizing Stencil Computations* (2014), ACM, pp. 25–32.

[53] PHILIPPSEN, M. *Automatic alignment of array data and processes to reduce communication time on DMPPs*, vol. 30. ACM, 1995.

[54] POULSON, J., MARKER, B., VAN DE GEIJN, R. A., HAMMOND, J. R., AND ROMERO, N. A. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw. 39*, 2 (feb 2013), 13:1–13:24.

[55] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Symposium on Operating System Design and Implementation (OSDI)* (2010), pp. 293–306.

[56] QIAN, Z., CHEN, X., KANG, N., CHEN, M., YU, Y., MOSCIBRODA, T., AND ZHANG, Z. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), EuroSys '12.

[57] RAMANUJAM, J., AND SADAYAPPAN, P. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (1989), ACM, pp. 637–646.

[58] RAMANUJAM, J., AND SADAYAPPAN, P. Compile-time techniques for data distribution in distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on 2*, 4 (1991), 472–482.

[59] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 49–68.

[60] SPARKS, E. R., TALWALKAR, A., SMITH, V., KOTTALAMA, J., PAN, X., GONZALEZA, J., FRANKLIN, M., JORDANA, M., AND KRASKAB, T. MLI: An API for distributed machine learning. In *arXiv:1310.5426* (2013).

[61] STOKELY, M., ROHANI, F., AND TASSONE, E. Large-scale parallel statistical forecasting computations in r. In *JSM Proceedings, Section on Physical and Engineering Sciences* (Alexandria, VA, 2011).

[62] STONEBRAKER, M., BROWN, P., BECLA, J., AND ZHANG, D. Scidb: A new dbms for science and other applications with complex analytics.

[63] TEAM, R. D. R: A language and environment for statistical computing.

[64] VENKATARAMAN, S., BODZSAR, E., ROY, I., AUYOUNG, A., AND SCHREIBER, R. S. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (Eurosys)* (2013).

[65] XIN, R. S., ROSEN, J., ZAHARIA, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Shark: Sql and rich analytics at scale. In *SIGMOD* (2013).

[66] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)* (2008).

[67] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.

A NP-Completeness Proof

A.1 Problem Definition

To simplify the proof, we consider only $newarray$, map and $swapaxis$ operators. The general case is discussed in section A.4. This problem contains several operators in a program and each one can be the input of others. The first step is to build an expression graph for this problem as shown in section 3.3. Next is to convert the expression graph to the tiling graph. We define a *tiling graph* as following:

1. A node group represents an operator and contains several partition nodes.
2. If an operator A is an input of an operator B in the expression graph, there are some edges between node group A and group B in the tiling graph. How node group A connects to node group B depends on the type of operator B.
3. The cost of an edge $A.tiling_I \rightarrow B.tiling_K$ is the network transmission cost to do operator B when A is tiled as $tiling_I$ and B is tiled as $tiling_K$.

Figure 16 shows three operators that will be used in the proof. There are two kinds of tilings, row and column, for each operator. There is no input for a $newarray$. As for map, there is at least one input array. The tiling nodes of an input node group are fully connected to the tiling nodes of map. If two tiling nodes represent the same tilings, there is no cost for the edge between them. Otherwise, the cost is the size of the array, N. The last operator is $swapaxis$. There is one input array for $swapaxis$ and each tiling node of the input array connects to the tiling node of $swapaxis$ representing the swapped tiling. The cost for both edges are zero.

The problem is to choose a unique tiling node for each node group without conflict and achieve the minimum overall cost (summation of cost of all edges adjacent to two chosen tiling nodes). Conflict means that if there are edges between node group A and node group B, the chosen nodes must bear the same relationship. For example, if the chosen tiling node for the input of $swapaxis$ means row tiling, the chosen tiling node for $swapaxis$ can only be column tiling to avoid conflict.

Instead of directly proving the problem, we prove the corresponding verify problem which is to find out if there is a choice with the cost less than or equal to K where K is an integer. We denote the verify problem as $TILING(K)$.

A.2 NP Proof

To show that $TILING$ is in NP, we need to prove that a given choice can be verified in polynomial time. Suppose N is the number of node groups. Given a solution, we can verify the solution by adding up the cost for all edges connected to each chosen tiling node. There are at most $N-1$ edges connected to a tiling node and N chosen tiling nodes, we can get the total cost in $O(n^2)$. Therefore, $TILING(K)$ is in NP.

A.3 NP-Completeness Proof

To show $TILING(K)$ is NP-Complete, we prove that $NAE-3SAT(N)$ can be reduced to $TILING(K)$. $NAE-3SAT$ is similar to $3SAT$ except that each clause must have at least one $true$ and one $false$. Therefore, it rules out TTT and FFF while $3SAT$ only excludes FFF.

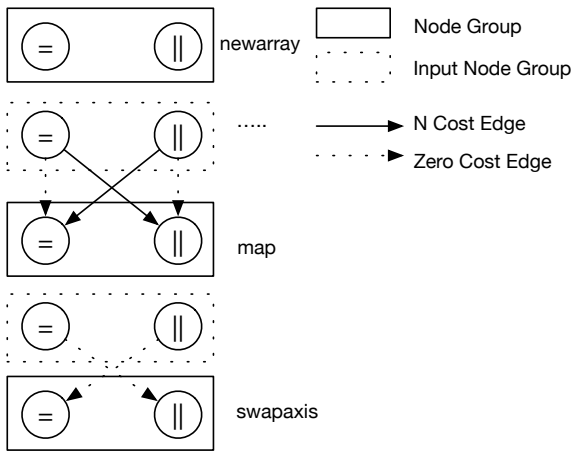


Figure 16: Three node groups and edge relationship with their input(s).

Assume that there are N literals and M clauses in the given question. M is polynomial to N. We prove that $NAE-3SAT(N)$ can be reduced to $TILING(K)$ where $K = M*2$.

1. **Construction Function,** $C(I)$:
   (a) For $C(I)$, $True$ is viewed as row tiling and $false$ is viewed as column tiling.
   (b) Each literal in $NAE-3SAT$ is an $array$ in $TILING(K)$. A negation literal is viewed as a $swapaxis$ of the original $array$.
   (c) For each clause $c_i = (L_1 \vee L_2 \vee L_3)$, $C(I)$ creates six expressions:

$$E_1 = map(swapaxis(L_1,0,1),L_2)$$
$$E_2 = map(swapaxis(L_1,0,1),L_3)$$
$$E_3 = map(swapaxis(L_2,0,1),L_1)$$
$$E_4 = map(swapaxis(L_2,0,1),L_3)$$
$$E_5 = map(swapaxis(L_3,0,1),L_1)$$
$$E_6 = map(swapaxis(L_3,0,1),L_2)$$

   For a negation literal, L, $swapaxis(L)$ represent the original $array$. For example, $C(I)$ creates six expressions for $c_j = (\neg L_1 \vee L_2 \vee L_3)$:

$$E_1 = map(L_1,L_2)$$
$$E_2 = map(L_1,L_3)$$
$$E_3 = map(swapaxis(L_2,0,1),swapaxis(L_1,0,1))$$
$$E_4 = map(swapaxis(L_2,0,1),L_3)$$
$$E_5 = map(swapaxis(L_3,0,1),swapaxis(L_1,0,1))$$
$$E_6 = map(swapaxis(L_3,0,1),L_2)$$

   For explanation purpose, we call the six expressions created by $C(I)$ a clause group.
   (d) After converting all clauses to clause groups, $C(I)$ create a cost graph according to the definition. Without loss of generality, we assume that the array size is 1. Therefore, the cost for an edge is either 0 or 1.

   For a clause group, if three literal have the same symbols, $true$ or $false$, the minimum cost is 6. For example, if three literals are all $true$ or all $false$ for $c_i = (L_1 \vee L_2 \vee L_3)$, the two inputs for each map of the clause group must have different tilings because of $swapaxis$. Thus the cost for map node group can only be 1. Since there are six maps for a clause group, the minimum cost is 6.

   For other cases, the minimum cost of a clause group is 2. For example, if L_1 is the only $true$ for $c_i = (L_1 \vee L_2 \vee L_3)$, only the input tilings of maps for $E4$ and $E6$ are different. Since all maps are not referenced by other operators, we can freely choose their tilings based only on the input tilings. Thus the cost for this case is 2. Other combinations are just symmetries of the above case and have the same cost.

   The time complexity for $C(I)$ is $O(N^2)$.

2. $C(B)$ **belongs to** $TILING(K)$ **if** B **belongs to** $TILING(K)$:
   If S is a solution for B, every clause in S has at least one $true$ and one $false$. This implies that at least one row tiling input and column tiling input for each clause group of $C(S)$. Therefore, the cost for $C(S)$ is $M*2$ which is equal to K.

3. B **belongs to** $NAE-3AT$ **if** $C(B)$ **belongs to** $TILING(K)$:
   If S is a solution for $C(B)$, there are at least one row tiling and one column tiling for each clause group. In other words, if one clause group has all row tiling inputs or all column tiling inputs, the total cost for the tiling graph will be at least $2*(M-1)+6 > K$. As a result, no clause group has all row tiling or column tiling input. Therefore, S is a solution for B.

Step 2 and step 3 prove that $NAE-3SAT$ can be reduced to $TILING(K)$.

A.4 General Graph

The previous proof only considers the tiling graph with $Array$, map and $swapaxis$. However, we argue that even though the tiling graph contains more different operators, it is still an NP-Complete problem to find out the solution. For any $TILING(K)$ which contains only the three operators, we add some other operators and expression which are independent from the original ones. Thus the new tiling graph contains two sub tiling graphs, the original tiling graph and the tiling graph representing the newly added operators. Moreover, two sub tiling graphs are not connected. Thus, to solve new $TILING(K')$ must first solve the $TILING(K)$ which is NP-Complete. Thus, we can also reduce $TILING(K)$ to the general graph.

Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code

Ryan Stutsman[1], Collin Lee[2], and John Ousterhout[2]
University of Utah[1], Stanford University[2]

Abstract

This paper describes how a rules-based approach allowed us to solve a broad class of challenging distributed system problems in the RAMCloud storage system. In the rules-based approach, behavior is described with small sections of code that trigger independently based on system state; this provides a clean separation between the deterministic and nondeterministic parts of an algorithm. To simplify the implementation of rules-based modules, we developed a task abstraction for information hiding and complexity management, pools for grouping tasks and minimizing the cost of rule evaluation, and a polling-based asynchronous RPC system. The rules-based approach is a special case of an event-based state machine, but it encourages a cleaner factoring of code.

1 Introduction

Over the last decade more and more systems programmers have begun working on new and challenging software subsystems that manage distributed resources in a concurrent and fault-tolerant fashion. We call these subsystems *DCFT modules* (Distributed, Concurrent, Fault-Tolerant). DCFT modules are most common in systems that provide infrastructure for large-scale applications, such as Bigtable [8], Chubby [6], Hadoop [2], HDFS [25], RAMCloud [22], Sparrow [23], and ZooKeeper [15]. A DCFT module typically manages a collection of distributed servers, such as workers in a MapReduce application or replicas for a chunk of data; it issues remote requests in parallel to maximize performance, and it recovers from failures so that higher layers of software need not deal with them.

DCFT code is different from most systems code because it must describe behavior that is highly nondeterministic. As a result, DCFT modules are painfully difficult to implement. For example, the Chubby developers reported:

> Fault-tolerant algorithms are notoriously hard to express correctly, even as pseudo-code. This problem is worse when the code for such an algorithm is intermingled with all the other code that goes into building a complete system. [7]

The current state of development for DCFT modules resembles the situation in the mid-1960s for synchronizing concurrent processes. In both cases, a new and challenging style of programming was becoming more common; there were no widely accepted design patterns for implementing these modules, so each team developed its own set of ad hoc implementation techniques. In the case of synchronization, many different approaches were tried over a period of more than a decade, and there was considerable discussion about which approach was best. By the early 1980s the systems community had mostly settled on locks and condition variables, and this approach has been the dominant one for managing small-scale concurrency over the last three decades. We hope that this paper will provide useful data to fuel the discussion of DCFT modules, and that agreement will eventually emerge that makes it easier to implement these challenging systems.

This paper describes our experiences implementing several DCFT modules in the RAMCloud storage system [22, 24]. After struggling with our first implementations, we noticed that each of the DCFT modules ended up organized around a collection of rules. In this *rules-based approach*, the behavior of a module is described with a small set of code snippets that trigger independently based on the module's state. The order of execution is not determined a priori, but rather by the evolution of the module's state in response to events in the distributed system.

Since discovering this commonality, we have used the rules-based approach explicitly in more recent DCFT modules; these modules have been considerably easier to develop than the early DCFT modules. Rules provide a clean mechanism for expressing the nondeterminism of a DCFT module while allowing the vast majority of code to be written in a traditional imperative style.

This paper makes three contributions. First and foremost, it provides the first in-depth discussion of how to implement DCFT modules in a practical large-scale system. The paper introduces two of the DCFT modules in RAMCloud, describes why they were hard to implement, and discusses the design choices we made for RAMCloud along with their implications. We believe that the problems and solutions for RAMCloud are general enough to be relevant for a variety of other systems.

Second, the paper describes the implications of a rules-based approach on system structure. We found several abstractions useful in structuring rules and implementing efficient rules-based subsystems:

- A *task* structure combines a set of related rules with a collection of state variables. Each task uses its rules and state variables to achieve a particular goal, such as replicating an object. Tasks make it easier to manage rules and understand their behavior.
- A *pool* is a simple scheduler that improves the efficiency of rule evaluation. Each pool manages a collection of related tasks; it separates inactive tasks (those that are in their goal state) from active tasks and evaluates rules only for the active tasks.
- A polling-based asynchronous mechanism for remote procedure calls (RPCs) provides an efficient and convenient way to incorporate remote communication into a rules-based module. Asynchronous RPCs provide a better factoring than messages because they allow many error conditions to be handled entirely within the RPC system.

These facilities allowed rules to be incorporated simply and naturally into the RAMCloud system.

The paper's third contribution is to demonstrate the value of the rules-based approach. Rules allowed us to solve a wide range of problems in RAMCloud using a small amount of code (only 30-300 lines of rules-based code for each DCFT module). Rules are also efficient: when used in the critical path of RAMCloud's write operations, rules overheads account for only about 200-300 ns out of the total write time of 13.5 µs. The rules-based approach is a specialized form of an event-driven state machine, but it results in cleaner factoring and simpler code than the traditional approach to state machines. We reimplemented the scheduler for Hadoop MapReduce (which uses the traditional approach) using rules; our rules-based implementation replaced 163 state transitions with only 19 rules.

2 DCFT modules

A DCFT module is a piece of code that runs on a single machine but coordinates a collection of distributed resources. For example, the resources might be a group of worker machines, each of which will process a subset of the data in a scalable computation. Or, the resources might be storage servers, out of which a subset will be chosen to store replicas for a chunk of data. In many cases the management complexity is concentrated in a DCFT module on a single machine. The other machines are simply slaves that respond to requests; the slaves are simple enough that they do not require the DCFT approaches discussed in this paper. In other cases, such as consensus protocols, each machine runs an independent DCFT module.

Most of the work of a DCFT module involves communicating with other machines, and this introduces two challenges. First, communication is expensive enough that DCFT modules typically issue concurrent requests to improve performance. Second, distributed resources may fail. For example, a worker may crash before completing its computation, or a storage server may crash and lose all of its replicas. A DCFT module must detect failures and take recovery actions such as restarting a computation on a different worker or creating new replicas to replace the lost ones. Ideally, the complexities of distribution, concurrency, and fault tolerance are encapsulated within the DCFT module, so that it provides a simple and fault-free API for its clients.

The rest of this section describes two DCFT modules from the RAMCloud storage system, which will be used as examples in the remainder of the paper. RAMCloud contains several other DCFT modules besides these two; they are described in Table 2 in Section 6.

2.1 Membership notifier

RAMCloud's cluster membership notifier is a relatively simple DCFT module. Each server in a RAMCloud cluster needs to know about all of the other servers currently in the cluster. A special server called the *cluster coordinator* maintains the master copy of cluster membership information, called the *server list*, and it must notify all of the other servers whenever a server enters or leaves the cluster. The membership notifier runs on the coordinator and is responsible for propagating server list changes to the rest of the cluster.

The notifier uses RPCs to send updates to other servers. In order to update the cluster quickly, it sends updates to multiple servers concurrently. Additional server list updates may occur while the notifier is working; when this happens, the notifier batches multiple updates in future RPCs in order to minimize the total number of RPCs. The notifier must ensure that each server eventually receives all updates, and that all servers observe server list changes in the same order.

The membership notifier must handle a variety of faults. For example, a server may crash while a notification RPC to it is underway. If some servers are slow to respond to RPCs, this must not prevent other servers from receiving timely updates. Temporary network outages may cause update RPCs to fail; these RPCs must be retried.

2.2 Replica manager

The most complex DCFT module in RAMCloud, and the one that motivated much of our thinking about DCFT code, is the replica manager. The replica manager handles log replication for storage servers. Each RAMCloud storage server, called a *master*, organizes its DRAM as an append-only log of data, which is divided into tens

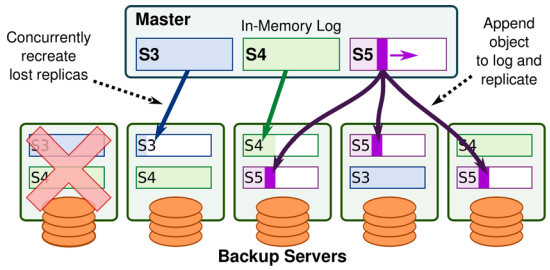


Figure 1: The replica manager is responsible for ensuring that each of a master's log segments is properly replicated on backups. In this example new data has recently been appended to the log head (S5), so it is being replicated. In addition, a backup has crashed, so the replica manager is replacing lost replicas for segments S3 and S4.

of thousands of 8 MB *segments* (see Figure 1). Each segment must be replicated on the secondary storage of several other servers, called *backups*. An independent replica manager module runs on each master; its job is to ensure that the segments on that master are properly replicated. When new data is appended to the head segment, the replica manager must update the replicas for that segment. If a backup server crashes, the replica manager must create replacements for any replicas stored on that server.

In addition to the requirements above, the replica manager must also enforce constraints between segments. For example, in order to ensure that the log head can be identified unambiguously during crash recovery, an initial header must be written to a new head segment before the previous head is closed by writing a footer to it, and the footer must be written before any data can be written to the new head segment. See [26] for details on these constraints.

The replica manager is under particularly stringent timing constraints, since it is on the critical path for basic write operations. A master cannot respond to a write request from a client until the new data has been fully replicated. In order to minimize write latency (currently about 13.5 µs end-to-end for small objects) the replica manager must issue update requests in parallel for all of the replicas of the head segment.

3 How we ended up with rules

We did not consciously choose a rules-based approach for RAMCloud's DCFT modules: the code gradually assumed this form as we added functionality over a series of refactorings. When we built the first DCFT modules in RAMCloud, such as the ones described in the previous section, we had no particular point of view on how to write such code, and we did not know that DCFT modules would require an unusual approach. Thus, we initially tried to write each module as a monolithic piece of code that solved a problem from start to finish using a traditional imperative approach. However, this approach broke down almost immediately because of nondeterminism caused by concurrency and faults. To handle the nondeterminism, the code disintegrated into fragments that needed to execute relatively independently. None of our DCFT modules has reached anywhere near complete functionality with an imperative implementation.

For example, in the replica manager, the disintegration was initially caused by the desire to replicate segments concurrently. Different replicas could be in different states and could progress at different rates, so it didn't make sense to manage the replicas with a deterministic global algorithm. The most natural approach was to treat each replica independently.

Fault tolerance caused additional disintegration of the code. Failures have the effect of undoing work that was previously completed, thereby requiring earlier steps to be redone. For example, in the replica manager, if a backup crashes while receiving a replica, the replica manager must redo the process of selecting a server to store the replica. Failures can occur at many points, and different failures may require different amounts of work to be redone. As a result, it isn't possible to code an algorithm from start to finish. It makes more sense to think about the algorithm in terms of steps that make incremental progress, such as selecting a backup server or transmitting the segment header to the server for a particular replica. The execution order of the steps is nondeterministic, based on concurrency and failures.

Given a large set of relatively independent code fragments, we faced the question of how to manage their execution. We considered a fine-grained threaded approach, but quickly rejected this possibility. The replica manager must manage thousands of segments, with several replicas per segment, so using a separate thread per replica, or even per segment, would have been too inefficient [27]. Furthermore, multi-threading would only have handled the code disintegration caused by concurrency; it would not have addressed the disintegration caused by fault tolerance. In addition, threads were not needed from a performance standpoint: most of the work of a DCFT module consists of managing RPCs to other servers, with only a few RPCs typically outstanding at a time.

We also considered a coarse-grained approach to threading like SEDA [27], where tasks pass through a series of stages with each stage served by one or a few threads. However, the inter-thread communication required for this would have been unacceptable given our requirements for low latency (for example, using a condition variable to wake a thread takes about 2 µs; RAMCloud servers process simple requests in about 1 µs).

Thus, we decided to manage all of the code fragments for each DCFT module in a single thread. This left the

problem of deciding the order in which fragments should execute in the thread. Writing an intelligent dispatcher that always knew what to do next was infeasible; the order depended on nondeterministic events such as RPC completions and failures, and there were complex dependencies between fragments (for example, the footer cannot be replicated for one segment until the header has been replicated for the following segment). As a result, we decided to let the fragments schedule themselves. Each fragment has an associated *condition*, which tests state variables to determine when it is appropriate for the fragment to run. The DCFT module operates by repeatedly testing conditions and executing the fragments for the conditions that are satisfied. Although this may appear to be expensive, we developed a few simple techniques that make this approach efficient (see Section 5).

Over time, we noticed that all of our DCFT modules disintegrated in the same way and ended up with similar features. Furthermore, these features resembled rules-based programming. Since then we have adopted the rules-based approach for all new DCFT modules, and we have developed infrastructure in RAMCloud to make the rules-based approach efficient and easy to use. Section 4 describes the approach at a conceptual level, and Section 5 describes how we implemented it in RAMCloud, and the supporting infrastructure that we developed.

In retrospect, we realized that DCFT modules require a highly unconstrained execution order; structures that restrict the order are likely to cause problems. For example, our first implementation of the membership notifier ran synchronously: each time the coordinator modified its server list to indicate the entry or exit of a server, the rest of the cluster was notified before returning from the modification. During this time, the server list lock was held. However, if a server crashed during the notification process, deadlock could result; the notifier couldn't complete without knowing about the crashed server (otherwise it would keep attempting to update that server), and it couldn't mark the server crashed without acquiring the lock. The rules-based version of this module never performs synchronous operations, which allows it to adapt to server list changes. We now use deadlock as a "canary in the coal mine:" if a module experiences deadlock, it may be a sign that its execution order is overly constrained and that we need to convert it to the rules-based approach.

As another example, we considered using the C++ exception mechanism to handle errors, but it is too restrictive to handle all of the error cases. Specifically, an exception handler can only catch exceptions in the calls nested beneath it, but the steps in a DCFT module do not follow a sequential or nested pattern. For example, the replica manager must replace segment replicas that are lost when a backup crashes. If a lost replica is for an older segment that is closed and inactive, then no thread is replicating it, and there is nowhere to deliver an exception.

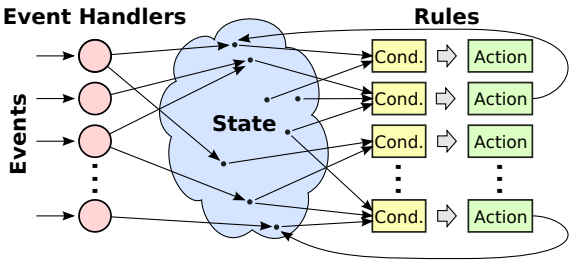


Figure 2: In a rules-based approach, code is divided into rules, each of which has a condition that tests state variables, and an action that is executed if the condition is satisfied. Rules can trigger in any order permitted by their conditions. The state is modified by handlers for external events and by the actions of some rules, which then allows new rules to trigger.

4 The rules-based approach

We use the term *rules-based* to describe a style of programming where there is not a crisp algorithm that progresses monotonically from beginning to end. Instead, the top-level controlling code of the module is divided into small chunks, called *actions*, which can potentially execute in any order (see Figure 2). Each action has an associated *condition* that determines when the action can execute; the condition is expressed as a predicate on the module's state variables. Together, an action and its associated condition constitute a *rule*.

A rules-based module operates by repeatedly selecting a rule whose condition is satisfied and then executing that rule's action. Each action makes incremental progress towards some *goal* (such as proper replication of a segment); the module executes rules repeatedly until it reaches the goal state. The goal is also described as a predicate on the module's state variables.

Actions can modify the state of the module or initiate external operations such as RPCs to other servers. Each action is nonblocking, and faults and external events have no effect on an action once it starts executing. If an action turns out to involve blocking or must handle nondeterminism due to faults, then it must be split into multiple actions in different rules. For example, an action cannot both initiate an RPC *and* wait for it to complete, since that would require the action to block and would expose it to nondeterministic failures of the RPC.

Nondeterminism manifests itself between actions, in the form of *events*. An event is an occurrence outside the direct control of the DCFT module that affects its behavior, such as:

Rules

Rule	Condition	Action
R1	No backup server selected.	Choose an available server to store replica.
R2	Header not committed, no RPC outstanding.	Start RPC containing segment header.
R3	Header RPC completed.	If backup rejected request, clear server assignment for replica. Otherwise, mark header committed and mark prior segment to allow footer replication.
R4	Uncommitted data, no RPC outstanding, prior footer is committed.	Start write RPC containing up to 1 MB of uncommitted data.
R5	RPC containing data completed.	Mark sent data as committed.
R6	Segment finalized, following header committed, footer not sent, no RPC outstanding.	Start RPC containing footer.
R7	Segment footer RPC completed.	Mark footer as committed and mark following segment to allow data replication.

Events

Event	What Happened	Handler
E1	RPC completed (or failed).	Update RPC object to indicate completion.
E2	New data added to segment.	Increment count of uncommitted bytes ready for replication.
E3	Backup server failed.	Cancel any RPCs outstanding to server. For all replicas stored on the failed server: cancel server selection; reset replica header, footer, and data to unsent and uncommitted.

Table 1: A partial list of the rules and events for managing one replica of a particular log segment. In the normal case, rules execute in order from R1 to R7 (R4 and R5 may trigger many times). Some rules test (R4 and R6) or modify (R3 and R7) state from multiple segments. If an RPC fails, no actions are taken other than to mark the state "no RPC outstanding"; the rules will automatically retry it. The handler for E1 is implemented by the RPC system.

- The completion of an RPC.
- The failure of a server.
- A new server joining the cluster (for the membership updater).
- The addition of new data to the head segment (for the replica manager).

When an event occurs, a handler updates state variables as shown in Figure 2; these state changes then allow new rules to trigger. For example, when an RPC completes, the RPC subsystem sets a state variable associated with the RPC.

The rules-based approach is similar in many ways to event-based programming. However, in event-based programming an event typically triggers actions directly. In the rules-based approach an event handler merely updates state variables; actions are then triggered based on the new state. In our experience, this two-step approach results in a cleaner factoring of code than the traditional event-based approach (see Section 8).

As an example, Table 1 shows some of the rules and events for managing a single segment replica in the replica manager described in Section 2. Rule R4 specifies the following predicate on a segment replica:

- some data appended to the segment has not been transmitted to the backup storing the replica, and
- no replication RPC is outstanding to the backup, and
- the preceding segment in the log has already committed its footer (so is safe to write to this replica).

If this condition is met, then the replica manager starts an RPC to send uncommitted data to the backup storing the replica. If the RPC completes successfully, a state variable is set, which allows R5 to execute. If a backup fails, event E3 executes: it cancels any RPCs outstanding to the failed backup, then iterates over the full list of segments in the log, resetting the replication state for any replica assigned to the failed backup. After the state is reset, recreation of the replicas happens automatically, just as it does during normal operation, starting with rule R1.

We found it natural to program with rules because they reflect the inherently nondeterministic structure of the problems being solved. Rules separate the deterministic parts of a module (actions) from the nondeterministic parts (events). Each action implements one of the basic steps of the module. In this problem domain it is difficult to describe all of the control flows from one step to another, so the rules-based approach does not even try. Instead, it describes the control flow in terms of the conditions that determine when each action executes, independently of how that state was reached. This results in a clean code factoring.

5 Implementing Rules

This section describes how we implemented the rules-based approach in RAMCloud, with emphasis on two issues: (a) achieving a clean code factoring, and (b) integrating rules-based DCFT modules cleanly and effi-

ciently with the rest of RAMCloud. We introduced two new abstractions to manage rules: *tasks*, which provide modularity by associating a set of rules with a collection of state variables, and *pools*, which reduce the cost of rule evaluation by separating tasks into *active* and *inactive* groups. The rules-based approach requires an asynchronous communication mechanism; we chose to implement an asynchronous RPC system, which provides a cleaner factoring than a message-based approach. In addition to discussing these abstractions, this section also describes how events are handled and the role of threads in processing rules.

5.1 Tasks

The primary abstraction for implementing rules in RAMCloud is a *task*. Tasks provide modularity for rule sets, and they make it easy to use rules within a system mostly programmed in an imperative style. A task consists of three elements: a collection of state variables, a set of rules, and a goal. The collection of state variables is implemented as an instance of a class, and the rules are implemented by an `applyRules` method on the class. Each invocation of `applyRules` makes one pass over all the rules for that task, testing conditions and invoking actions for any conditions that are satisfied. In its simplest form, the body of `applyRules` consists of a sequence of `if` statements, one for each rule.

The goal of a task represents the outcome that the task is trying to achieve, such as ensuring proper replication of a single segment or updating all of the server lists in the cluster to reflect a change on the coordinator. A goal can be expressed as a predicate on the task's state variables. Goals are reminiscent of invariants, but we chose to use a different term because goals are unmet during much of the operation of a DCFT module, whereas invariants are almost always true.

A DCFT module contains one or more tasks. It operates by repeatedly calling the `applyRules` methods on its tasks until all tasks have achieved their goals. Events may cause a task to fall out of its goal state (for example, a server may crash, or new data may arrive that requires replication). If this happens, the DCFT module resumes processing rules until all tasks have once again reached their goal states. RAMCloud uses a polling approach, continually testing rules for tasks not in their goal state. Section 7 describes how to use rules in environments where sleeping is preferable to polling.

Many DCFT modules contain only a single task; the RAMCloud membership notifier is one example. Its state includes the coordinator's server list (including its version number), the version number of the server list stored on each server in the cluster, a list of recent updates, where each update mutates a server list from one version to the next, and a list of outstanding RPCs. The task contains three rules:

- **Condition**: there exists a server whose server list is out of date with respect to the coordinator's list, and for which there is currently no outstanding RPC.
  Action: initiate an RPC to that server, containing the updates not yet received by that server.
- **Condition**: there are updates that are no longer needed (they have been received by all servers in the cluster).
  Action: delete those updates.
- **Condition**: an outstanding RPC has completed.
  Action: if the RPC succeeded, update the version number stored for that server to reflect the updates it just received.

The goal of the membership notifier is to reach a state where the update list is empty. The notifier is implemented as a thread that repeatedly invokes the `applyRules` method until the goal is achieved, then sleeps until the coordinator's server list changes.

We try to structure our `applyRules` methods to make it easier for the programmer to reason about the overall behavior of the task. For example, we tend to order the rules in an `applyRules` method to match the order in which they will occur in the normal case without errors. This preserves enough ordering in the code for the developer to understand how the code is intended to progress.

For tasks with a complex state space, it can sometimes be difficult to ensure that the rules cover all possible states. In these cases, we write the task's rules using a set of nested `if` statements, each of which always has an unconditional `else` clause. This approach ensures that all possible states have been considered; exactly one action executes each time `applyRules` is called. Some of these cases may not contain code, which means that state is waiting for an event; the empty block serves as documentation that the state was considered.

5.2 Handling Events

In addition to invoking rules, a DCFT module must also handle events, which are occurrences outside the module that modify its state. Events are asynchronous with respect to the DCFT module, so they must synchronize with the DCFT module's rules engine in order to update state. In RAMCloud this is typically done with a traditional locking approach. For example, in the membership notifier, the only events other than RPC completions are modifications to the coordinator's server list, which also create new entries in the update list. A lock synchronizes these modifications with the execution of rules, and a condition variable is used to wake up the rules engine if it is sleeping when the server list is modified.

RPC completion events are handled specially using state variables; this mechanism is described in Section 5.5.

5.3 Threading

There are several ways we could have chosen to use threads in implementing rules. One possibility would be to employ threading on a fine-grain basis, with one thread for each task or perhaps even one thread for each rule; however, this is unnecessary and inefficient. For RAMCloud we chose a coarse-grain approach where each DCFT module executes in a single thread and evaluates its rules sequentially. There is little incentive to use multiple threads within a DCFT module because DCFT modules spend most of their time waiting for RPCs to complete. The concurrency of a DCFT module comes from concurrent execution of RPCs to other servers, not from concurrent execution of the module's internal code. If a DCFT module's functions include significant local processing then multiple threads might make sense for that module, and Section 7 describes how this can be implemented, but we have not yet encountered any modules where this is the case.

Using a single thread per DCFT module eliminates the need for most locks within a DCFT module, which makes the module both simpler and faster. For example, rules from one task can safely test and modify state variables from other tasks without synchronization (see rules R4 and R7 in Table 1). However, most DCFT modules must respond to some events generated outside the module, and locks are needed to synchronize these event handlers with the DCFT rules. This is typically implemented with a lock around each call to `applyRules`.

Different DCFT modules can execute concurrently in RAMCloud, since they use different threads.

5.4 Pools

Many of RAMCloud's DCFT modules are simple ones with only one task, but other modules have multiple tasks. For example, the replica manager uses one task for each segment stored on the server (typically tens of thousands). Evaluating all of the rules for all of these tasks is prohibitively expensive, so we introduced a *pool* abstraction to make rule evaluation efficient. A pool is a simple scheduler for a collection of related tasks. Pools reduce the overhead of rule application by dividing tasks into two groups: *active* tasks, whose rules must be evaluated, and *inactive* tasks, whose rules can be skipped. A task stays active until it achieves its goal, at which point it becomes inactive. Typically, only a small subset of tasks are active at a time, so testing rules is efficient. Over its life, a single task may be activated and deactivated many times, since failures and other state changes can return a task to a state where its goal is no longer met.

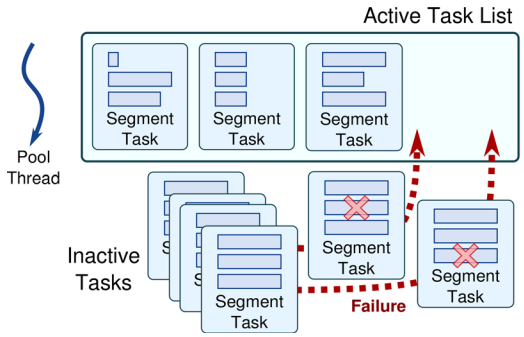


Figure 3: The replica manager's pool and its associated tasks. Each task manages the replication of a single segment. The pool thread continually evaluates rules for active tasks (those that have not yet met their replication goal). When a task meets its replication goal it becomes inactive and its rules are no longer evaluated. Server failures cause the state of some tasks to be reset, and the affected tasks are reactivated. Testing rules is efficient, since there is rarely more than one segment (the head) under active replication at a time.

Each pool is implemented as a thread associated with a list of active tasks. Whenever the list contains at least one task, the thread cycles through the tasks, invoking their `applyRules` methods. When the list is empty the thread sleeps.

For example, the replica manager contains a pool with one task for each segment; the task's goal is to maintain three complete replicas of the data in the segment. When new data is added to the head segment, the corresponding task is activated (see Figure 3). When the task catches up in replicating its data, its goal is reached, so it is deactivated. Once a segment is completely filled by the server and its task finishes replicating it, then its task is deactivated forever unless one of its replicas is lost due to a failure. In this case the task is reactivated to begin recreating the lost data. At any given time, most segments are fully replicated, so the replica manager pool usually only has to test rules for a few segment tasks at a time.

5.5 Asynchronous RPCs

The rules-based approach requires an asynchronous communication mechanism because actions that initiate remote requests cannot wait for completion (blocking would prevent other rules from firing; it would also expose actions to nondeterminism, since remote communication can fail). To meet this requirement, we implemented an asynchronous RPC mechanism based on polling. This section describes how the RAMCloud RPC system simplifies the implementation of rules, and why it results in a cleaner code factoring than alternatives such as callback-based RPCs or message-based programming.

In RAMCloud all RPCs are inherently asynchronous. Each RPC is represented with a C++ object; the constructor for the object forms the request message and initiates transmission of that message. The RPC object contains a state variable that can be tested to determine whether the RPC has completed. If the state variable indicates completion, another method may be invoked on the RPC object to retrieve results or failure information. Each RPC object also supports a synchronous `wait` method that polls the state variable until the RPC has finished.

This approach fits naturally with rules-based programming: DCFT modules keep RPC objects as part of their state, and the conditions for rules test the RPC objects for completion (see Table 1 for an example). An alternative approach for asynchronous RPCs is to invoke a callback function when an RPC completes. However, callbacks are awkward because they must synchronize their execution with rules that might be executing concurrently. The state variable provides a simpler form of synchronization between the completion of the RPC and the rules engine.

RAMCloud uses polling not only for asynchronous RPCs in DCFT rules engines, but also for synchronous RPCs invoked outside DCFT modules. Polling works well in RAMCloud because the expected completion time for RPCs is only a few microseconds. Blocking a thread to wait for an RPC serves little purpose: by the time the CPU could switch to another task, the RPC will probably have completed, and the polling approach eliminates the latency overhead for waking the blocked thread when the RPC completes.

An alternative to asynchronous RPCs would have been to use a messaging approach, with separate request and response messages. However, we found that RPCs produce a cleaner code factoring by allowing more functionality to be implemented transparently in the RPC mechanism; this simplifies the code in DCFT modules. Remote procedure calls automatically associate each request message with the corresponding response message. In a pure message-based approach, higher level software must make this association, which increases its complexity. Furthermore, the RPC approach allows some errors to be detected and handled transparently in the RPC system, whereas a message-based approach must expose these errors to higher-level software. RAMCloud's RPC system allows the creation of customized modules that recover automatically from many errors. For example, if a network connection fails, a recovery module will automatically open a new connection and retry; or, if an RPC fails with an error indicating that the target server no longer stores the desired object, a recovery module will automatically find the correct server and retry the request with that server. As a result, many of RAMCloud's RPCs return no errors except those caused by bad arguments: all system errors are handled internally by the RPC system. In a message-based approach, these problems must be handled by higher-level software.

6 Evaluation

This section discusses the strengths and weaknesses of rules, based on our experiences in RAMCloud.

6.1 Benefits

Thinking in terms of rules has allowed us to produce new DCFT modules more quickly, with fewer refactorings before reaching satisfactory solutions. Specifically:

- The task and pool abstractions simplify the development of DCFT modules. Tasks serve a purpose similar to that of monitors [19]: a monitor helps to modularize synchronization code by encapsulating a lock with a collection of state variables and a set of methods that manipulate those variables; a task helps to modularize DCFT code by encapsulating a collection of state variables with rules and events that manipulate those variables to achieve a goal. Both of these structures provide a framework that reduces the number of decisions a developer must make to produce a working module.
- The `applyRules` methods bring all of the rules for a task together in a few pages of code, making it easier to understand the task's behavior.
- It is relatively easy to add rules to an existing DCFT module when new issues are discovered.
- It is straightforward to integrate rules-based modules into the RAMCloud system. We use rules-based code surgically in only a few `applyRules` methods, while the vast majority of the system is programmed in a traditional imperative fashion.

Table 2 summarizes each of the seven DCFT modules in RAMCloud, with two overall conclusions. First, the table shows that the rules-based approach can be used to implement a variety of tasks, including different approaches to replication, coordinating workers executing in parallel, and crash recovery. Second, the rules-based approach allows the nondeterministic parts of the system to be concentrated in a small amount of code, so that the vast majority of the system can be written using a simpler imperative style. The `applyRules` methods in RAMCloud range in size from 30-300 lines, which is only a small fraction of the overall DCFT modules. All of the rules-based code in RAMCloud amounts to only about 1,100 lines, out of a total system size of more than 50,000 lines.

One potential problem with the rules-based approach is the cost of testing rule conditions, which happens repeatedly. We measured this cost for the RAMCloud replica manager, which is the most time-sensitive DCFT

DCFT Module	Functionality	Task Types	Rules	Events	`applyRules` code
Membership notifier	Notifies all servers of changes to coordinator's server list	1	3	3	36
Replica manager	Maintains a specified number of replicas of each segment on a master	1	23	3	258
Recovery manager	Executes on coordinator to recover crashed master: locates complete copy of log, splits the master's tablets, coordinates many masters to replay log and recover partitions	4	12	2	299
Recovery master replay	Executes on recovery masters during crash recovery: reads log segment replicas from backups, replays entries, replicates new data	1	3	0	230
Backup replica recovery	Executes on backups during crash recovery: reads segment replicas, divides log entries into buckets for different recovery masters	1	4	2	31
Multi-read	Executes on clients: reads many objects concurrently using batched requests to multiple servers	1	2	2	75
Indexed read	Executes on clients: retrieves index entries from one or more secondary index servers, then reads the corresponding objects from other servers	1	14	2	132

Table 2: Summary of DCFT modules in RAMCloud. "Task Types" counts the number of different kinds of task (not instances) in the module. "Rules" counts the total number of rules in all task types. "Events" counts only module-specific events (it excludes RPC completion events, which are handled automatically by the RPC subsystem). "`ApplyRules` code" counts lines of code (not including comments) in all `applyRules` methods. Some of the line counts include additional code not directly related to processing rules.

module in RAMCloud (it is on the critical path for all write operations). The replica manager also has the largest rule set of all the RAMCloud DCFT modules. As shown in Figure 4, only a few hundred nanoseconds are needed for evaluating conditions in each call to `applyRules`. When `applyRules` takes a significant amount of time to execute, it is because of actions that initiate RPCs and handle completions. Furthermore, only two invocations of `applyRules` are on the critical path for each write: the first (which issues replication RPCs) and the last (which receives the results from the last replication RPC). Based on Figure 4, we estimate that testing conditions accounts for 200-300 ns out of a total time of about 13.5 µs for writes.

6.2 Challenges

It is not always easy to identify modules that require the rules-based approach. The natural tendency is to code any new module in an imperative style (especially for programmers not already familiar with DCFT modules and rules), and it is easy to underestimate the implications of fault tolerance. Thus, we sometimes find ourselves attempting to implement new DCFT modules without rules. When this happens, corner cases result in refactorings that gradually break up the code flow, until eventually we realize that we need to switch to a rules-based approach. The introduction of rules usually simplifies the code, and seemingly intractable problems suddenly become tractable. For example, the introduction of rules in the membership notifier eliminated deadlocks that had plagued several previous versions of the code.

The greatest challenge in using rules is to get out of the traditional mental model where an algorithm is defined monolithically. Instead, the algorithm must be defined as a collection of independent small pieces, each of which makes incremental progress towards a goal. These pieces become the actions of rules, and conditions and event handlers are added to invoke the actions appropriately. Our experience is that once a developer adopts this mental model, the actual rule set follows fairly quickly, and it is straightforward to incorporate the rules into the overall system.

It can be difficult to visualize the behavior of a DCFT module from a collection of rules. However, we think this problem is inevitable, given the nondeterminism that the rules must capture: nondeterministic solutions will always be harder to understand than deterministic ones.

We do not advocate the use of rules as an overall architecture for applications. Asynchronous nondeterministic programming is fundamentally more difficult than traditional imperative programming, so it should only be used where it is absolutely necessary.

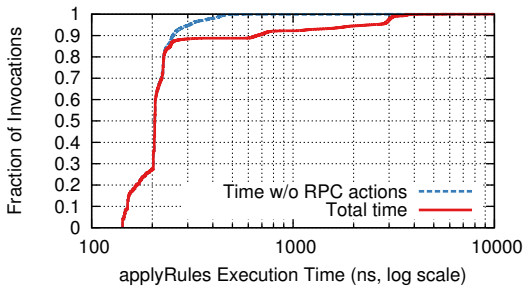


Figure 4: Cumulative distribution of the execution time of the `applyRules` method for the RAMCloud replica manager, measured using YCSB Workload A [9] to generate writes of 1000-byte objects. "Total time" includes the cost of actions as well as condition checks. "Time w/o RPC actions" excludes time spent in actions that initiate RPCs and process RPC results. Most invocations of `applyRules` occur while waiting for RPCs to complete; these invocations test conditions but no actions fire.

7 Rules Without RAMCloud

Although most of our experience with rules is in the context of RAMCloud, we believe that the rules-based approach also makes sense for other applications. This section discusses modifications of the rules approach that may be appropriate in environments other than RAMCloud.

A polling approach to rules evaluation makes sense in RAMCloud's low-latency environment, but in applications with high communication latency it makes more sense for the rules engine to sleep while waiting for events. The rules approach can accommodate sleeping with two modifications. First, the rules engine must be able to determine when it is safe to sleep. To do this, `applyRules` methods must return an indication of whether any rules triggered. If no rules triggered, then no state changes were made, so no rules will trigger in the future until an event occurs. In this situation, the pool can deactivate the task just as if it had reached its goal state. In fact, with this mechanism for sleeping, no special handling is needed for goal states: the mechanism for sleeping will handle them automatically. Once all of the tasks in a pool are inactive, the pool can sleep. The second modification for sleeping is that the rules engine must wake up again when necessary. To implement this, each event must be associated with a task; when the event occurs, the task is reactivated and its pool is awakened.

Most of the rest of the rules mechanism still applies, even in environments without polling. For example, the task abstraction still makes sense as a way of encapsulating a rule set with its state variables. Pools are still useful, both for minimizing the number of rules that must be tested and also as the mechanism for sleeping. Asynchronous RPCs retain their advantages over messages or synchronous RPCs, as described in Section 5.5.

One additional modification that may be appropriate in some environments is to relax RAMCloud's one-thread-per-DCFT-module restriction. If some actions involve significant local processing, then it may be desirable to allow other rules to execute concurrently with them. Concurrency can be implemented using an approach similar to that for asynchronous RPCs: the action dispatches its work to a separate worker thread and then returns, so that the rules engine can process other rules while the worker thread executes. When the worker thread completes, it sets a state variable just like an RPC completion, which can then cause other rules to trigger. If the worker thread needs to access state variables during its execution, then it must synchronize with the rules engine.

In summary, most of RAMCloud's rule mechanism carries over directly to other environments, and with a few small changes the mechanism can handle issues we have not yet experienced in RAMCloud, such as high-latency communication and long-running actions.

8 Event-driven state machines

The rules-based approach emerged so consistently in all of our DCFT modules that we initially thought it might be inevitable. However, we have since discovered that other systems use different approaches for DCFT modules. The most common alternative appears to be an event-driven state machine; Chubby [6] and Hadoop [2] are examples of this approach. In this section we compare the rules-based approach to this alternative, and we argue that the rules-based approach produces cleaner and simpler code.

An event-driven state machine is a system with one or more state variables, whose behavior is determined by events. When an event occurs, the state machine takes actions based on the current state and the event. The actions can alter the state, which affects the way that future events are handled.

The state machine definition is broad enough that it includes the rules-based approach as a special case. However, in most state machines the actions are determined directly from the events. Rules use a two-step approach where event handlers only modify state and never take actions. Rules then trigger based on the resulting state, which reflects the sum of the outside events.

The difference between these two approaches is subtle, but the rules-based approach results in a cleaner code factoring. In DCFT modules, the current state of the system is more important than how the system got to that state, so it is cleaner to structure code around state, not events. A single event may need to trigger many actions, and the same action might be triggered from multiple events. For example, the replica manager may

State Machine	States	Transitions	Distinct
`Job`	14	82	27
`Task`	7	24	16
`TaskAttempt`	13	57	15
Total	**34**	**163**	**58**

Table 3: Hadoop MapReduce 2.4 manages task scheduling using 3 state machines. For each state machine the table lists the number of explicitly named states, the total number of transitions between states, and the number of distinct actions among all the transitions for the state machine.

need to choose a backup for a particular replica either because a new segment is being created, or because an existing replica was lost in a crash. The traditional state machine approach results in considerable duplication of code, which is not present in the rules-based approach.

To demonstrate the advantages of the rules-based approach, we analyzed the job scheduler in Hadoop MapReduce 2.4, which uses the state machine approach described above. The overall goal of the job scheduler is to schedule a collection of tasks across a group of servers. The module manages a group of objects, with each object controlled at any given time by one of three state machines (see Table 3). In total, the three state machines contain 34 states, with 163 separately-defined *transitions*, where a transition describes the actions to take when a particular event occurs in a particular state.

Of the 163 transitions, only 58 have distinct actions: the other 105 transitions are duplicates. Furthermore, upon analysis of the actions, we found that many of the "distinct" transitions are near-duplicates. For example, rather than writing one error cleanup action that works across many states, MapReduce contains numerous nearly-identical cleanup actions, each specialized slightly for the state and event that trigger it.

For comparison, we reimplemented the MapReduce job scheduler using a rules-based approach, with each state machine replaced by one task. We used Python for the rules-based implementation because of its rapid-prototyping capabilities and verified by hand that each of the 163 transitions in the Java state machines is covered by the rules-based implementation. Our Python implementation is complete enough to schedule and run simple jobs. The source code for the Python implementation is available on GitHub along with the corresponding code for the Java state machine [3].

The rules-based implementation of the MapReduce scheduler is significantly simpler than the state machine implementation: a total of 19 rules in 3 tasks provided functionality equivalent to the 163 transitions in the state implementation. Each of the three `applyRules` methods fits in a screen or two of code (117 total lines of code and comments between the three `applyRules` methods), which makes it possible to view the entire behavior of each task at once. Furthermore, the order of the rules within each `applyRules` method shows the normal order of processing, which also helps visualization. In contrast, the state machine implementation required more than 750 lines of code just to specify the three transition tables, plus another 1,500 lines of code for the transition handlers.

Transition handler counts, rule counts, and lines of code are metrics that are easy to compare, but other metrics may provide more insight. For example, more elaborate code complexity metrics or a full user study could help highlight the differences between the approaches.

9 Other Related Work

Several formalisms exist for specifying and reasoning about concurrent code. For example, Dijkstra's Guarded Command Language (GCL) [11] provides nondeterministic conditional and loop constructs similar to the iterative conditional checks used in rules-based tasks. Hoare's Communicating Sequential Processes (CSP) [14] extends GCL to support specification of and reasoning about interconnected nondeterministic processes. GCL and CSP have been influential in the design of concurrency primitives of programming languages like the recent Go and Rust systems languages. The occam [5] programming language adheres to CSP even more closely; programs in occam tend to follow a rules-based style.

Lamport's Temporal Logic of Actions and his TLA+ specification language [18, 28] allow specification of concurrent systems. TLA+ supports a model checker and proof system. However, it is not a full programming language and cannot be used for implementation. Similar to our rules, TLA+ specifications use conditional atomic actions to transition from one state to the next; this may make it a good fit for verifying rules-based modules.

Rules are also used in other application domains to solve problems other than DCFT:

- Expert Systems [16] and the General Problem Solver [21, 20] are AI programs that reason using heuristic methods; they are implemented as a set of rules that iteratively transform a knowledge base to arrive at a solution.
- Make [12] is a utility that builds software based on user-provided rules. Each rule specifies how to rebuild a particular file, if the current version is out of date.
- Model Checkers are formal verification tools that, like rules-based modules, iteratively apply conditional transformations to state variables to ensure user-specified invariants are preserved.

Actors, originally conceived in the 1970s, have become popular in recent years for building distributed or concurrent applications [13, 1]. In the actors approach, a program is divided into independent modules with no shared state, called *actors*, which communicate using asynchronous messages. Actors often handle messages using an approach like that described in Section 8 for event-driven state machines (each actor is a state machine, and messages represent events). However, actors could also use a rules-based approach, and this would be advantageous for actors that implement DCFT modules. We also believe that an asynchronous RPC system would provide a better communication mechanism for DCFT actors than asynchronous messages, as discussed in Section 5.5.

Several groups have developed domain-specific languages and frameworks for specifying DCFT modules as a collection of event-driven state machines:

- Chubby [7] is a consensus-based configuration service that uses a custom state machine specification language to simplify the specification of its core algorithm.
- Mace [17] provides a restricted language for specifying state machines that allows specifications to be verified using a provided model checker. Unlike the rules-based approach, Mace programs only perform actions in response to events, though Mace also provides a construct that generates events in response to conditions on state.
- P [10] is a graphical language for specifying state machines that was used to implement the USB device driver stack of Microsoft Windows 8. P also allows model checking of state machines.
- SEDA [27] is an event-driven framework for building highly concurrent services that avoids the overhead of the thread-per-request approach. SEDA services are actor-like; they consist of pipelined stages interconnected by message queues. Each stage can optimize throughput by adjusting how many threads it uses. SEDA doesn't address fault-tolerance, and it increases response latency, making it inappropriate for RAMCloud.
- Bloom [4] is a language for building distributed systems in which programs are expressed using declarative rules over unordered sets of tuples. A key benefit of Bloom's "disorderly" approach is that it avoids artificial coordination compared to traditional imperative programs. Conditions on rules have something of a declarative style in rules-based code, but actions are programmed in an imperative style.

10 Conclusion

DCFT modules are becoming increasingly important in large-scale software systems, but they are difficult to implement and developers today have little guidance on how to implement them. In this paper we have described the problems we faced while implementing DCFT modules in RAMCloud and the rules-based solution that emerged from our experience. The rules-based approach results in a simple code factoring because it separates the deterministic and nondeterministic parts of a DCFT module. In addition, we found that a few other patterns and mechanisms encouraged a clean factoring of rules-based code, including tasks, pools, and an asynchronous RPC system. With this infrastructure, it was relatively easy to incorporate rules-based code into the RAMCloud system, and the rules-based approach has provided clean solutions to a variety of problems. In comparison to other approaches we considered, the rules-based approach encourages a cleaner factoring of code, which is particularly important given the inherent complexity of DCFT modules.

Experience with many more systems will be needed before agreement can be reached on the best way to implement DCFT modules. We hope that our experience can serve as a basis for additional discussion and experimentation.

11 Acknowledgments

Many people provided helpful feedback on this paper, including Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Eric Eide, Diego Ongaro, Henry Qin, David Silver, numerous anonymous conference reviewers, and our shepherd Ajay Gulati. This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by grants from Facebook, Google, Huawei, Mellanox, NEC, NetApp, Samsung, and VMWare.

References

[1] Akka, 2014. http://akka.io/.

[2] Welcome to Apache Hadoop!, 2014. http://hadoop.apache.org/.

[3] PlatformLab/mappy Git Repository, May 2015. https://github.com/PlatformLab/mappy.git.

[4] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[5] A. Burns. *Programming in Occam 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[6] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[10] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 321–332, New York, NY, USA, 2013. ACM.

[11] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[12] S. I. Feldman. Make — A Program for Maintaining Computer Programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[13] P. Haller and M. Odersky. Event-Based Programming Without Inversion of Control. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer Berlin Heidelberg, 2006.

[14] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC '10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[16] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[17] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[18] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[19] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, Feb. 1980.

[20] A. Newell, J. C. Shaw, and H. A. Simon. Report on a General Problem-solving Program. In *IFIP Congress*, pages 256–264, 1959.

[21] A. Newell and H. Simon. GPS, A Program That Simulates Human Thought. In *Computers & thought*, pages 279–293. 1995.

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[23] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[24] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[25] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[26] R. S. Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford, CA, USA, 2013.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[28] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication

Asaf Cidon[1], Robert Escriva[2], Sachin Katti[1], Mendel Rosenblum[1], and Emin Gün Sirer[2]

[1]Stanford University

[2]Cornell University

ABSTRACT

Cloud storage systems typically use three-way random replication to guard against data loss within the cluster, and utilize cluster geo-replication to protect against correlated failures. This paper presents a much lower cost alternative to full cluster geo-replication. We demonstrate that in practical settings, using two replicas is sufficient for protecting against independent node failures, while using three random replicas is inadequate for protecting against correlated node failures.

We present Tiered Replication, a replication scheme that splits the cluster into a primary and backup tier. The first two replicas are stored on the primary tier and are used to recover data in the case of independent node failures, while the third replica is stored on the backup tier and is used to protect against correlated failures. The key insight of our paper is that, since the third replicas are rarely read, we can place the backup tier on separate physical infrastructure or a remote location without affecting performance. This separation significantly increases the resilience of the storage system to correlated failures and presents a low cost alternative to geo-replication of an entire cluster. In addition, the Tiered Replication algorithm optimally minimizes the probability of data loss under correlated failures. Tiered Replication can be executed incrementally for each cluster change, which allows it to supports dynamic environments in which nodes join and leave the cluster, and it facilitates additional data placement constraints required by the storage designer, such as network and rack awareness. We have implemented Tiered Replication on HyperDex, an open-source cloud storage system, and demonstrate that it incurs a small performance overhead. Tiered Replication improves the cluster-wide MTTF by a factor of 20,000 compared to random replication and by a factor of 20 compared to previous non-random replication schemes, without increasing the amount of storage.

1. INTRODUCTION

Popular cloud storage systems like HDFS [33], GFS [15] and Azure [6] typically replicate their data on three random machines to guard against data loss within a single cluster, and geo-replicate the entire cluster to a separate location to guard against correlated failures.

In prior literature, node failure events are broadly categorized into two types: independent node failures and correlated node failures [4, 5, 7, 14, 25, 38]. Independent node failures are defined as events during which nodes fail individually and independently in time (e.g., individual disk failure, kernel crash). Correlated failures are defined as failures in which several nodes fail simultaneously due to a common root cause [7, 11] (e.g., network failure, power outage, software upgrade). In this paper, we are focused on events that affect data durability rather than data availability, and are therefore concerned with node failures that cause permanent data loss, such as hardware and disk failures, in contrast to transient data availability events, such as software upgrades.

The conventional wisdom is that three-way replication is cost-effective for guarding against node failures within a cluster. We also note that, in many storage systems, the third replica was introduced mainly for durability and not for read performance [7, 8, 13, 34].

Our paper challenges this conventional wisdom. We show that two replicas are sufficient to protect against independent node failures, while three replicas are inadequate to protect against correlated node failures.

We show that in storage systems in which the third replica is only read when the first two are unavailable (i.e., the third replica is not used for client reads), the third replica would be used almost only during correlated failure events. In such a system, the third replica's workload is write-dominated, since it would be written to on every system write, but very infrequently read from.

This property can be leveraged by storage systems to increase durability and reduce storage costs. Storage systems can split their clusters into two tiers: the *primary tier* would contain the first and second copy of each replica, while the *backup tier* would contain the backup third replicas. The backup tier would only be

used when data is not available in the primary tier. Since the backup tier's replicas will be read infrequently they do not require high performance for read operations. The relaxed read requirements for the third replica enable system designers to further increase storage durability, by storing the backup tier on a remote site (e.g., Amazon S3), which significantly reduces the correlation in failures between nodes in the primary tier and the backup tier. This is a much lower cost alternative to full cluster geo-replication, in which all three replicas are stored in a remote site. Since the backup tier does not require high read performance, it may also be compressed, deduplicated or stored on a low-cost storage medium that does not offer low read latency but supports high write bandwidth (e.g., tape) to reduce storage capacity costs.

Existing replication schemes cannot effectively separate the cluster into tiers while maintaining cluster durability. Random replication, the scheme widely used by popular cloud storage systems, scatters data uniformly across the cluster and has been shown to be very susceptible to frequent data loss due to correlated failures [2, 7, 9]. Non-random replication schemes, like Copyset Replication [9], have significantly lower probability of data loss under correlated failures. However, Copyset Replication is not designed to split the replicas into storage tiers, does not support nodes joining and leaving, and does not allow storage system designers to add additional placement constraints, such as supporting chain replication or requiring replicas to be placed on different network partitions and racks.

We present Tiered Replication, a simple dynamic replication scheme that leverages the asymmetric workload of the third replica. Tiered Replication allows system designers to divide the cluster into primary and backup tiers, and its incremental operation supports nodes joining and leaving. In addition, unlike Random Replication, Tiered Replication enables system designers to limit the frequency of data loss under correlated failures. Moreover, Tiered Replication can support any data layout constraint, including support for chain replication [37] and topology-aware data placement.

Tiered Replication is an optimization-based algorithm that places chunks into the best available replication groups. The insight behind its operation is to select replication groups that both minimize the probability of data loss under correlated failures by reducing the overlap between replication groups, and satisfy data layout constraints defined by the storage system designer. Tiered Replication increases the MTTF by a factor of 20,000 times compared to Random Replication, and by a factor of 20 compared to Copyset Replication.

We implemented Tiered Replication on HyperDex, a cloud storage system that can scale up to hundreds of thousands of nodes [13]. Our implementation of Tiered Replication is versatile enough to satisfy constraints on replica assignment and load balancing, including HyperDex's data layout requirements for chain replication [37]. We analyze the performance of Tiered Replication on a HyperDex installation on Amazon, in which the backup tier, containing the third replicas, is stored on a separate Amazon availability zone. We show that Tiered Replication incurs a small performance overhead for normal operations and preserves the performance of node recovery. Our open source implementation of Tiered Replication on HypderDex is publicly available.

2. MOTIVATION

In this section, we demonstrate why three-way replication is not cost-effective. First, we demonstrate that it is superfluous to use a replication factor of three to provide data durability against independent failures, and that two replicas provide sufficient redundancy for this type of failure. Second, building on previous work [7, 9], we show that random three-way replication falls short in protecting against correlated failures. These findings provide motivation for a replication scheme that more efficiently handles independent node failures and provides stronger durability in the face of correlated failures.

2.1 Analysis of Independent Node Failures

Consider a storage system with N nodes and a replication factor of R. Independent node failures are modeled as a Poisson Process with an arrival rate of λ. Typical parameters for storage systems are $N = 1,000$ to $N = 10,000$ and $R = 3$ [5, 7, 14, 33].

$\lambda = \frac{N}{MTTF}$, where $MTTF$ is the mean time to permanent failure of a standard node. We borrow the failure assumption used by Yahoo and LinkedIn, for which about 1% of the nodes in a typical cluster fail independently each month [7, 33]. Consequently, we use a node MTTF of 10 years. We also assume in the model that the number of nodes remains constant and that there is always an idle server available to replace a failed node.

When a node fails, the cluster re-replicates its data from several servers, which store replicas of the node's data and write the data into another set of nodes. The node's recovery time depends on the number of servers that can be read from in parallel to recover the data. Using previously defined terminology [9], we term $scatter\ width$ or S as the average number of servers that participate in a single node's recovery. For example, a node that has $S = 10$ has its data replicated uniformly on 10 other nodes, and when the node fails, the storage system can re-replicate the data by reading from and writing to 10 nodes in parallel.

A single node's recovery time is modeled as an exponential random variable, with a recovery rate of μ. We assume that recovery rate is a linear function of the

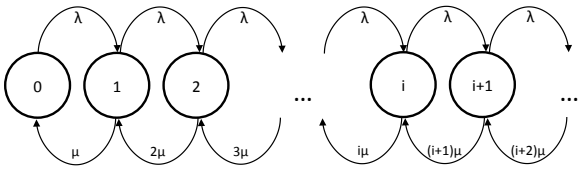


Figure 1: Markov chain of data loss due to independent node failures. Each state represents the number of nodes that are down simultaneously.

Number of Nodes	Pr(2 Failures)	Pr(3 Failures)	Pr(4 Failures)
1,000	6.51×10^{-7}	2.48×10^{-10}	7.07×10^{-14}
5,000	1.62×10^{-5}	3.08×10^{-8}	4.40×10^{-11}
10,000	6.44×10^{-5}	2.45×10^{-7}	7.00×10^{-10}
50,000	1.54×10^{-3}	2.93×10^{-5}	4.18×10^{-7}
100,000	5.81×10^{-3}	2.21×10^{-4}	6.31×10^{-6}

Table 1: Probability of simultaneous node failures due to independent node failures under different cluster sizes. The model uses $S = 10$, $R = 3$, $\tau = 60\ minutes$ and an average node MTTF of 10 years.

$scatter\ \ width$, or a linear function of the number of nodes that recover in parallel. $\mu = \frac{S}{\tau}$, where $\frac{\tau}{S}$ is the time to recover an entire server over the network with S nodes participating in the recovery. Typical values for $\frac{\tau}{S}$ are between 1-30 minutes [7, 14, 33]. For example, if a node stores 1 TB of data and has a scatter width of 10 (i.e., its data would be scattered across 10 nodes), and each node was read from at a rate of 500 MB/s, it would take about three minutes to recover the node's data. As a reference, Yahoo has reported that when the cluster recovers a node in parallel, it takes about two minutes to recover a node's data [33].

Throughout the paper we use $\tau = 60\ minutes$ with a scatter width of 10, which results in a recovery time of 6 minutes (three times higher than the recovery time reported by Yahoo [33]). Note that there is a practical lower bound to recovery time. Most systems first make sure the node has permanently failed before they start recovering the data. Therefore, we do not consider recovery times that are below one minute. We also assume that each node has a single 2 TB disk that can be recovered at a rate of 500 MB/s, and that each node's data is split into 10,000 chunks. These numbers match standard industry parameters [6, 9, 17].

The rate of data loss due to independent node failures is a function of two probabilities. The first is the probability that i nodes in the cluster have failed simultaneously at a given point in time: $Pr(i\ failed)$. The second is the probability of loss given i nodes failed simultaneously: $Pr(loss|i\ \ failed)$. In the next two subsections, we show how to compute these probabilities, and in the final subsection we show how to derive the overall rate of failure due to independent node failures.

2.1.1 Probability of i Nodes Failing

We first express $Pr(i\ \ failed)$ using a Continuous-time Markov chain, depicted in Figure 1. Each state in the Markov chain represents the number of failed nodes in a cluster at a given point in time.

The rate of transition between state i and $i + 1$ is the rate of independent node failures across the cluster, namely λ. The rate of the reverse transition between state i and $i - 1$ is the recovery rate of single node's data. Since there are i failed nodes, the recovery rate of a single node is $(i) \cdot \mu$ (in other words, as the number of nodes the cluster is trying to recover increases, the time it takes to recover the first node decreases, because more nodes participate in recovery). We assume that the number of failed nodes does not affect the rate of recovery. This assumption holds true as long as the number of failures is relatively small compared to the total number of nodes, which is true in the case of independent node failures in a large cluster (we demonstrate this below).

The probability of each state in a Markov chain with N states can always be derived from a set of N linear equations. However, since N is on the order of magnitude of 1,000 or more, and the number of simultaneous failures due to independent node failures in practical settings is very small compared to the number of nodes, we derived an approximate closed-form solution that assumes an infinite sized cluster. This solution is very simple to compute, and we provide the analysis for it in Appendix 8.

The probability of i nodes failing simultaneously is:

$$Pr(i\ failed) = \frac{\rho^i}{i!} e^{-\rho}$$

Where $\rho = \frac{\lambda}{\mu}$. The probabilities for different cluster sizes are depicted in Table 1. The results show that for clusters smaller than 10,000 nodes, the probability of two or more simultaneous independent failures is very low.

2.1.2 Data Loss Given i Node Failures

Now that we have estimated $Pr(i\ failed)$, we need to estimate $Pr(loss|i\ \ failed)$. Previous work has shown how to compute this probability for different types of replication techniques using simple combinatorics [9]. Replication algorithms map each chunk to a set of R nodes. A *copyset* is a set that stores all of the copies of a chunk. For example, if a chunk is replicated on nodes $\{7, 12, 15\}$, then these nodes form a copyset.

Random replication selects copysets randomly from the entire cluster. Facebook has implemented its own

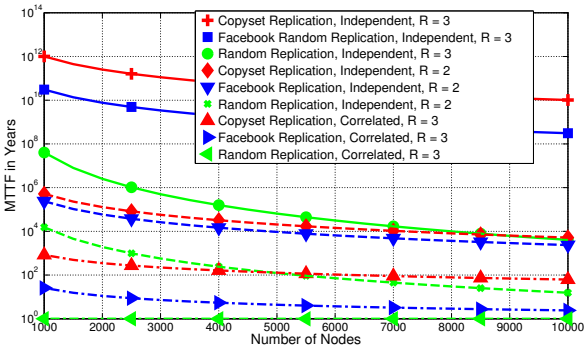


Figure 2: MTTF due to independent and correlated node failures of a cluster with a scatter width of 10.

random replication technique in which the R nodes are selected from a pre-designated window of nodes. For example, if the first replica is placed on node 10, the remaining two replicas will randomly be placed on two nodes out of a window of 10 subsequent nodes (i.e., they will be randomly selected from nodes $\{11, ..., 20\}$) [2, 9].

Unlike random schemes, Copyset Replication minimizes the number of copysets [9]. The following example demonstrates the difference between Copyset Replication and Facebook's scheme. Assume our storage system has: $R = 3$, $N = 9$ and $S = 4$. In Facebook's scheme, each chunk will be replicated on another node chosen randomly from a group of S nodes following the first node. E.g., if the primary replica is placed on node 1, the secondary replica will be randomly placed either on node 2, 3, 4 or 5. Therefore, if our system has a large number of chunks, it will create 54 distinct copysets.

In the case of a simultaneous failure of three nodes, the probability of data loss is the number of copysets divided by the maximum number of sets:

$$\frac{\#\ copysets}{\binom{N}{R}} = \frac{54}{\binom{9}{3}} = 0.64$$

Now, examine an alternative scheme using the same parameters. Assume we only allow our system to replicate its data on the following copysets:

$$\{1,2,3\},\{4,5,6\},\{7,8,9\}\{1,4,7\},\{2,5,8\},\{3,6,9\}$$

That is, if the primary replica is placed on node 3, the two secondary replicas can only be randomly placed on nodes 1 and 2 or 6 and 9. Note that with this scheme, each node's data will be split uniformly on four other nodes. The new scheme creates only 6 copysets. Now, if three nodes fail, the probability of data loss is:

$$\frac{\#\ copysets}{84} = \frac{6}{84} = 0.07.$$

Consequently, as we decrease the number of copysets, $Pr(loss|i\ failed)$ decreases. Therefore, this probability is significantly lower with Copyset Replication compared to Facebook's Random Replication.

Note however, that as we decrease the number of copysets, the frequency of data loss under correlated failures will decrease, but each correlated failure event will incur a higher number of lost chunks. This is a desirable tradeoff for many storage system designs, in which each data loss event incurs a fixed cost [9]. Another design choice that affects the number of copysets is the scatter width. As we increase the scatter width, the number of copysets used by the system also increases.

2.1.3 MTTF Due to Independent Node Failures

We can now compute the rate of loss due to independent node failures, which is:

$$Rate\ of\ Loss = \frac{1}{MTTF} = \lambda \sum_{i=1}^{N} Pr(i-1\ failed) \cdot (1 - Pr(loss|i-1\ failed)) \cdot Pr(loss|i\ failed)$$

The equation accounts for all events in which the Markov chain switches from state $i-1$, in which no loss occurs, to state i, in which data loss occurs. λ is the transition rate between state $i-1$ and i, $Pr(i-1\ failed)$ is the probability of state $i-1$, $(1-Pr(loss|i-1\ failed))$ is the probability that there was no data loss when $i-1$ nodes failed, and $Pr(loss|i\ failed)$ is the probability of data loss when i nodes failed. Since no data loss can occur when $i < R$, the sum can be computed from $i = R$.

In addition, Table 1 shows that under practical system parameters, the probability of i simultaneous node failures due to independent node failures drops dramatically as i increases. Therefore:

$$Rate\ of\ Loss = \frac{1}{MTTF} \approx \lambda \cdot Pr(R-1\ failed) \cdot Pr(loss|R\ failed)$$

Using this equation, Figure 2 depicts the MTTF of data loss under independent failures for $R = 2$ and $R = 3$ with three replication schemes, Random Replication, Facebook's Random Replication and Copyset Replication, as a function of the cluster's size. It is evident that Facebook's Random Replication and Copyset Replication have a much higher MTTF than Random Replication. The reason is that they use a much smaller number of copysets than Random Replication, and therefore their $Pr(loss|i\ failed)$ is smaller.

2.2 Analysis of Correlated Node Failures

Correlated failures occur when an infrastructure failure causes multiple nodes to be unavailable for a long period of time. Such failures include power outages that may affect an entire cluster, and network switch and

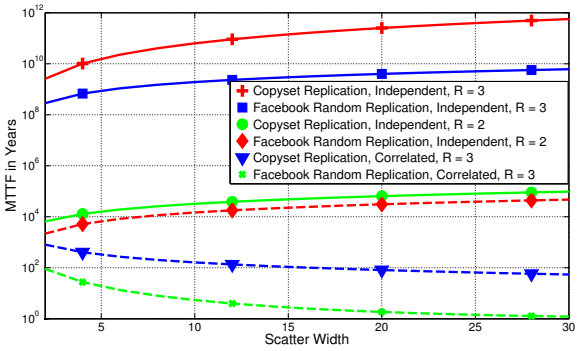


Figure 3: MTTF due to independent and correlated node failures of a cluster with 4000 nodes.

rack failures [7, 11]. Storage systems can avoid data loss related to some common correlated failure scenarios, by placing replicas on different racks or network segments [5, 7, 14]. However, these techniques only go so far to mitigate data loss, and storage systems still face unexpected simultaneous failures of nodes that share replicas. Such data loss events have been documented by multiple data center operators, such as Yahoo [33], LinkedIn [7] and Facebook [2, 5].

In order to analyze the affect of correlated failures on MTTF, we use the observation made by LinkedIn and Yahoo, that about once a year, 1% of the nodes do not recover after a cluster-wide power outage. This has been documented as the most common severe correlated failure [7, 33]. We can compute the probability of data loss for this event using combinatorics [9].

Figure 2 also presents the MTTF of data loss under correlated failures. It is evident from the graph that the MTTF due to correlated failures for $R = 3$ is three orders of magnitude lower than that for independent failures with $R = 2$ and six orders of magnitude lower than that for independent failures with $R = 3$, for any replication scheme.

We conclude that $R = 2$ is sufficient to protect against independent node failures, and that system designers should only focus on further increasing the MTTF under correlated failures, which is by far the main contributing factor to data loss. This has been corroborated in empirical studies conducted by Google [14] and LinkedIn [7].

This provides further evidence that random replication is very susceptible to correlated and independent failures. Therefore, in the rest of the paper we compare Tiered Replication only against Facebook's Random Replication and Copyset Replication.

Figure 3 plots the MTTF for correlated and independent node failures using the same model as before, as a function of the scatter width. This graph demonstrates that Copyset Replication provides a much higher MTTF than Facebook's Random Replication scheme. The figure also shows that increasing the scatter width has an opposite effect on MTTF for independent and correlated node failures. The MTTF due to independent node failures increases as a function of the scatter width, since a higher scatter width provides faster node recovery times, since more nodes participate in simultaneous recovery. In contrast, the MTTF due to correlated node failures decreases as a function of the scatter width, since a higher scatter width produces more copysets.

Since the MTTF is determined primarily by correlated failures, we can also conclude that if system designers wish to reduce the cluster-wide MTTF, they should use a small scatter width.

2.3 The Peculiar Case of the Nth (or Third) Replica

This analysis prompted us to investigate whether we can further increase the MTTF under correlated failures. We assume that the third replica was introduced in most cases to provide increased durability and not for increased read throughput [7, 8, 13, 34].

Therefore, consider a storage system in which the third replica is never read unless the first two replicas have failed. We estimate how frequently the system requires the use of a third replica, by analyzing the probability of data loss under independent node failures for a replication factor of two. If a system loses data when it uses two replicas, it means that if a third replica existed and did not fail, the system would recover the data from it.

In the independent failure model depicted by Figures 2 and 3, Facebook Random Replication and Copyset Replication require the third replica very rarely, on the order of magnitude of every 10^5 years.

To leverage this property, we can split our storage system into two tiers. The *primary tier* contains the first and second replicas of each chunk (or the N-1 replicas of each chunk), while the *backup tier* contains the third (or Nth) replica of each chunk. If possible, failures in the primary tier will always be recovered using nodes from the primary tier. We only recover from the backup tier if both the first and second replicas fail simultaneously. In case the storage system requires more than two nodes for read availability, the primary tier will contain the number of replicas required for availability, while the backup tier will contain an additional replica. Additional backup tiers (containing a single replica) can be added to support read availability in multiple geographies.

Therefore, the backup tier will be mainly used for durability during severe correlated failures, which are infrequent (on the order of once a year), as reported by various operators [5, 7, 33]. Consequently, the backup tier can be viewed as write-dominated storage, since it is written to on every write (e.g., thousands of times per second), but only read from a few times a year.

Splitting the cluster into tiers provides multiple advantages. The storage system designer can significantly reduce the correlation between failures in the primary tier and the backup tier similar to full cluster geo-replication. This can be achieved by storing the backup tier in a geographically remote location, or by other means of physical separation such as using different network and power infrastructure. It has been shown by Google that storing data in a physical remote location significantly reduces the correlation between failures across the two sites [14].

Another possible advantage is that the backup tier can be stored more cost-effectively than the primary tier, since it does not require low read latency. For example, the backup tier can be stored on a cheaper storage medium (e.g., tape, or disk in the case of an SSD-based cluster), its data may be compressed [17, 19, 22, 28, 30], deduplicated [12, 27, 40] or may be configured in other ways to be optimized for a write dominated workload.

The idea of using full cluster geo-replication has been explored extensively. However, existing geo-replication techniques replicate all replicas from one cluster to a second cluster, which multiplies the cost of storage [14, 23].

In the next section, we design a replication technique, Tiered Replication, that supports tiered clusters and does not duplicate the entire cluster. Unlike random replication, Tiered Replication is not susceptible to correlated node failures, and unlike previous non-random techniques like Copyset Replication, it supports data topology constraints such as tiered replicas and minimizes the number of copysets, even when the number of nodes in the cluster changes over time [9].

3. DESIGN

The goal of Tiered Replication is to create copysets (groups of nodes that contain all copies of a single chunk). When a node replicates its data, it will randomly choose a copyset that it is a member of, and place the replicas of the chunk on all the nodes in its copyset. Tiered Replication attempts to minimize the number of copysets while providing sufficient scatter width (i.e., node recovery bandwidth), and ensuring that each copyset contains a single node from the backup tier. Tiered Replication also flexibly accommodates any additional constraints defined by the storage system designer (e.g., split copysets across racks or network tiers).

Algorithm 1 describes Tiered Replication, while Table 2 contains the definitions used in the algorithm. Tiered Replication continuously creates new copysets until all nodes are replicated with sufficient scatter width. Each copyset is formed by iteratively picking candidate nodes with a minimal scatter width that meet the constraints of the nodes that are already in the copyset. Algorithm 2 describes the part of the algorithm that checks whether the copyset has met the constraints. The first constraint satisfies the tier requirements, i.e., having exactly one node in each copyset that belongs to the backup tier. The second constraint enforces the minimization of the number of copysets, by requiring that the nodes in the new copyset do not appear with each other in previous copysets. This constraint minimizes the number of copysets, because each new copyset contributes the maximum increase of scatter width.

Name	Description
cluster	list of all the nodes in the cluster
node	the state of a single node
R	replication factor (e.g., 3)
cluster.S	desired minimum scatter width of all the nodes in the cluster
node.S	the current scatter width of a node
cluster.sort	returns a sorted list of the nodes in increasing order of scatter width
cluster.addCopyset(copyset)	adds copyset to the list of copysets
cluster.checkTier(copyset)	returns false if there is more than one node from the backup tier, or R nodes from the primary tier
cluster.didNotAppear(copyset)	returns true if each node never appeared with other nodes in previous copysets

Table 2: Tiered Replication algorithm's variables and helper functions.

Algorithm 1 Tiered Replication

```
1:  while ∃ node ∈ cluster s.t. NODE.S < CLUSTER.S do
2:      for all node ∈ cluster do
3:          if NODE.S < CLUSTER.S then
4:              copyset = {node}
5:              sorted = CLUSTER.SORT
6:              for all sortedNode ∈ sorted do
7:                  copyset = copyset ∪ {sortedNode}
8:                  if CLUSTER.CHECK(copyset) == false then
9:                      copyset = copyset - {sortedNode}
10:                 else if COPYSET.SIZE == R then
11:                     CLUSTER.ADDCOPYSET(copyset)
12:                     break
13:                 end if
14:             end for
15:         end if
16:     end for
17: end while
```

Algorithm 2 Check Constraints Function

```
1: function CLUSTER.CHECK(copyset)
2:     if CLUSTER.CHECKTIER(copyset) == true AND
           CLUSTER.DIDNOTAPPEAR(copyset) AND
           ... // additional data layout constraints then
3:         return true
4:     else
5:         return false
6:     end if
7: end function
```

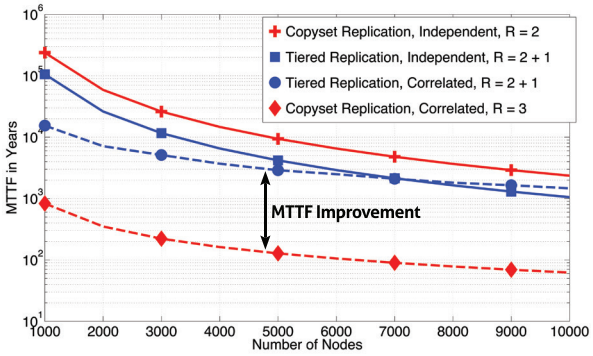


Figure 4: MTTF improvement of Tiered Replication with a scatter width of 10.

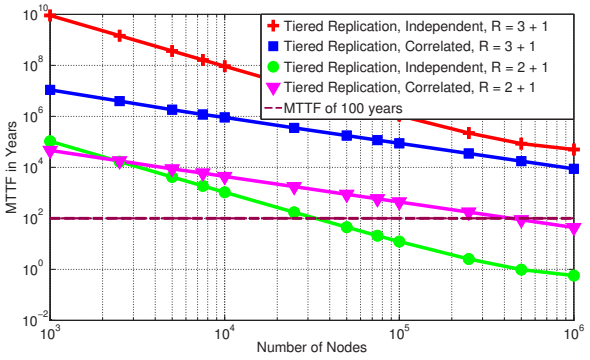


Figure 5: Tiered Replication would require 4 replicas to maintain an MTTF of 100 years once the cluster scales to 25,000 nodes with a scatter width of 10.

Note that there may be cases in which the algorithm does not succeed to find a copyset that satisfies all the constraints. In this case, the algorithm is run again, with a relaxed set of constraints (e.g., we can relax the constraint of minimizing the number of copysets, and allow more overlap between copysets). In practical scenarios, in which the number of nodes is an order of magnitude or more greater than S, the algorithm will easily satisfy all constraints.

3.1 Analysis of Tiered Replication

We evaluate the durability of Tiered Replication under independent and correlated node failures. To measure the MTTF under independent node failures, we use the same Continuous-time Markov model that we presented in Section 2. The results are presented in Figures 4 and 5. Note that $R = 2 + 1$ means we use Tiered Replication with two replicas in the primary tier and one replica in the backup tier.

Under Tiered Replication, when a replica fails in the primary tier, if possible, it is only recovered from other nodes in the primary tier. Therefore, fewer nodes will participate in recovery, because the backup tier nodes will not be recovered from. In order to compensate for this effect, system designers that use Tiered Replication may choose to increase the scatter width. For our analysis we compute the MTTF using the same scatter width for Tiered Replication and other replication schemes. Figure 4 shows that for $S = 10$, the MTTF under independent node failures is higher for Copyset Replication compared to Tiered Replication, because fewer nodes participate in the recovery of primary replicas and its single-node recovery time is therefore higher.

Also, note that in Figures 4 and 5, we assume that for $R = 2+1$, the third replica is never used to recover from independent node failures. In reality, the backup tier is used for any failure of two nodes from the primary tier, and therefore will be used in the rare case of an independent node failure that simultaneously affects two nodes in the primary tier that are in the same copyset. Hence, the MTTF under independent node failures for Tiered Replication is even higher than depicted by the graphs.

To evaluate the durability of Tiered Replication under correlated failures, we quantify the probability that all the nodes in one copyset or more fail. Since the primary and backup tiers are stored on separate infrastructure, we assume that their failures are independent.

Since each copyset includes two nodes from the primary tier, when these nodes fail simultaneously, data loss will occur only if the third copyset node from the backup tier failed at the same time. Since our assumption is that correlated failures occur once a year and affect 1% of the nodes each time (i.e., an MTTF of 100 years for a single node), while independent failures occur once in every 10 years for a node, it is 10 times more likely that if a backup node fails, it is due to an independent node failure. Therefore, the dominant cause of failures for Tiered Replication is when a correlated failure occurs in the primary tier, and at the same time an independent node failure occured in the backup tier.

To compute the MTTF due to this scenario, we need to compute the probability that a node failure will occur in the backup cluster while a correlated failure event is occurring in the primary cluster. To be on the conservative side, we assume that it takes 12 hours to fully recover the data after the correlated failure in the primary tier (LinkedIn data center operators report that unavailability events typically take 1-3 hours to recover from [7]). We compute the probability of data loss in this scenario, using the same combinatorial methods that we used to compute the MTTF under correlated failures before.

Figure 4 shows that the MTTF of Tiered Replication is more than two orders of magnitude greater than Copyset Replication. This is due to the fact that it is much less likely to lose data under correlated failures when one of the replicas is stored on an independent cluster. Recall

that Copyset Replication's MTTF was already three orders of magnitude greater than random replication.

In Figure 5 we explore the following: what is the turning point, when a storage system needs to use $R = 4$ instead of $R = 3$? We plot the MTTF of Tiered Replication and extend it $N = 1,000,000$, which is a much larger number of nodes than is used in today's clusters. Assuming that storage designers are targeting an MTTF of at least 100 years, our results show that at around 25,000 nodes, storage systems should switch to a default of $R = 4$. Note that Figure 4 shows that Copyset Replication needs to switch to $R = 4$ much sooner, at about 5,000 nodes. Other replication schemes, like Facebook's scheme, fail to achieve an MTTF of 100 years with $R = 3$, even for very small clusters.

3.2 Dynamic Cluster Changes

Since running Tiered Replication is fast to execute (on the order of milliseconds, see Section 4) and the algorithm is structured to create new copysets incrementally, the storage system can run it every time the cluster changes its configuration.

When a new node joins the cluster, we simply run Tiered Replication again. Since the new node does not belong to any copysets, it starts with a scatter width of 0. Tiered Replication's greedy operation ensures that the node is assigned to a sufficient number of copysets that will increase its scatter width to the value of S.

When a node dies (or leaves the cluster), it leaves behind copysets that are missing a single node. The simplest way to re-instate the copysets is to assume that the old copysets are down and run the algorithm again. The removal of these copysets will reduce the scatter width of the nodes that were contained in the removed copysets, and the algorithm will create a new set of copysets to replace the old ones. The data in the old copysets will need to be re-replicated R times again. We chose this approach when implementing Tiered Replication on HyperDex, due to its simplicity.

Alternatively, the algorithm can be optimized to look for a replacement node, which addresses the constraints of the remaining nodes in the copyset. In this scenario, if the algorithm succeeds in finding a replacement, the data will be re-replicated only once.

3.3 Additional Constraints

Tiered Replication can be extended to support different requirements of storage system designers by adding more constraints to the $cluster.check(copysets)$ function. The following provides two examples.

Controlled Power Down: Some storage designers would like to allow parts of the cluster to be temporarily switched off to reduce power consumption (e.g., according to diurnal patterns). For example, Sierra [36], allows a cluster with three replicas to power down two thirds of its cluster and still allow access to data. This feature can easily be added as a constraint to Tiered Replication by forcing each copyset to contain a node that belongs to a different tier. This feature is also important for supporting controlled software or hardware upgrades, during which parts of the cluster may be powered down without affecting the cluster availability.

Chain Replication: Chain replication can provide improved performance and consistency. Each replica is assigned a position in the chain (e.g., head, middle, tail) [37]. A desirable property of chain replication is that each node will have an equal number of replicas in each position. It is straightforward to incorporate this requirement into Tiered Replication. In order to ensure that nodes have an even distribution of chain positions for their replicas, when the algorithm assigns nodes to copysets and chain positions, it tries to balance the number of times the node will appear in each chain position. For example: if a node has been assigned to the head position twice, middle position twice and tail position once, the algorithm will enforce that it will be assigned to a tail position in the next copyset the node will belong to.

To demonstrate the ability to incorporate additional constraints to Tiered Replication, we implemented it on HyperDex [13], a storage system that uses chain replication. Note that Copyset Replication and Random Replication are inefficient for supporting balanced chain sets. Copyset Replication is not designed for incorporating such constraints because it randomly permutes the entire set of nodes. Random Replication is not effective for this requirement because its random placement of nodes frequently creates imbalanced chain positions.

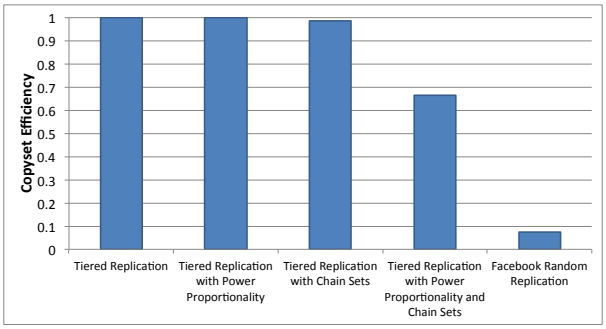


Figure 6: Adding constraints on Tiered Replication increases the number of copysets, on a cluster with $N = 4000$, $R = 3$ and $S = 10$.

3.4 Analysis of Additional Constraints

Figure 6 demonstrates the effect of adding constraints on the number of copysets. In the figure, *copyset effi-*

ciency is equal to the ratio between the number of copysets generated by an optimal replication scheme that minimizes the number of copysets, and the number of copysets generated by the replication scheme with the constraint. Note that in an optimal replication scheme, nodes will not appear more than once with each other in different copysets, or in other words, the number of copysets would be equal to $S \cdot (R - 1)$.

The graph shows that as we further constrain Tiered Replication, it is less likely to generate copysets that meet multiple constraints, and its copyset efficiency will decrease. The figure also shows that the chain set constraint has a greater impact on the number of copysets than the power down constraint. In any case, Tiered Replication with additional constraints significantly outperforms any Random Replication scheme.

4. IMPLEMENTATION

In order to evaluate Tiered Replication in a practical setting, we have extended the open-source HyperDex [13] key-value store to use Tiered Replication for replica set selection. HyperDex is a distributed, fault-tolerant, and strongly consistent key-value store. For our purposes, HyperDex's architecture is especially suited for evaluating different replication strategies. In HyperDex, a replicated state machine serves as the coordinator, and is responsible for maintaining the cluster membership and metadata. Part of this metadata includes the complete specification of all replica sets in the system.

Since Tiered Replication can be implemented efficiently, and is only run when nodes join or leave the cluster, we implement the greedy algorithm directly in the replicated HyperDex coordinator.

In HyperDex, a variant of chain replication [37] called value-dependent chaining specifies the replica set of an object as a chain of nodes through which writes propagate. We incorporate this into Tiered Replication in the form of additional constraints that track the positions of nodes within chains as well as the replica sets each node appears in. This helps ensure that each node will be balanced across different positions in the chain.

We implement Tiered Replication using synchronous replication. All writes will be acknowledged by all replicas in both tiers before the client is notified of success. Practically, this means writes will not be lost except in the case of correlated failure of all replicas in both tiers.

In total, our changes to HyperDex are minimal. We added or changed about 600 lines of C++ code, of which 250 lines constitute the greedy Tiered Replication algorithm. The rest of our implementation provides the administrator with tools to denote whether a node belongs to the primary or the backup tier of the cluster.

5. EVALUATION

In order to measure the performance impact of Tiered Replication in a practical setting, we do not attempt to measure the frequency of data loss under realistic scenarios, because it is impractical to run a cluster of thousands of nodes for decades.

5.1 Performance Benchmarks

We set up a 9 node HyperDex cluster on Amazon EC2 using M3 xlarge instances. Each node has a high frequency Intel Xeon E5-2670 v2 (Ivy Bridge) with 15 GiB of main memory and two 40 GB SSD volumes configured to store HyperDex data.

We compare Tiered Replication to HyperDex's default replication scheme, which does not support smart placement across two different availability zones. We ran 6 nodes in one availability zone (us-east-1a) and the three remaining nodes in a second availability zone (us-east-1b). In Amazon EC2, each availability zone runs on its own physically distinct, independent infrastructure. Common points of failures like generators and cooling equipment are not shared across availability zones. Additionally, they are physically separate, such that even extremely uncommon disasters such as fires, tornados or flooding would only affect a single availability zone [1].

In both deployments, the cluster is physically split across the availability zones, but only the tiered replication scheme ensures that there is exactly one node from the backup tier in each chain. In both deployments, we measured the throughput and latency of one million requests with the Yahoo! Cloud Serving Benchmark [10]. YCSB is an ideal choice of benchmark, because it has become the de-facto standard for benchmarking key-value stores, and provides a variety of workloads drawn from real workloads in place at Yahoo. We configured YCSB to run 32 client threads per host on each of the hosts in the cluster, with the database prepopulated with 10 million 1KiB objects. Figure 7 shows the throughput measured for both deployments. The difference in throughput is due to the fact that Tiered Replication does not load balance the nodes evenly in very small clusters. The figure includes error bars for the observed throughput over any one second interval through the course of the experiment.

5.2 Write Latency

We measure the write latency overhead of Tiered Replication. The workload consists of 50% read operations and 50% write operations. When we compared the write latency of Tiered Replication across two availability zones with Random Replication across two zones, we did not find any difference in latency.

Figure 8 comares the write latency of Tiered Replication across two zones with Random Replication across a single zone. As expected, Tiered Replication adds some

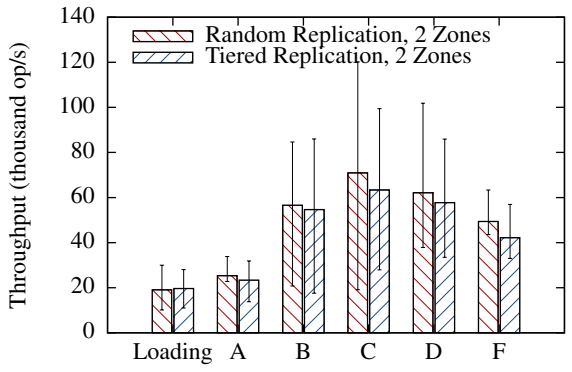


Figure 7: Tiered Replication throughput under YCSB benchmark. Each bar represents the throughput under a different YCSB workload.

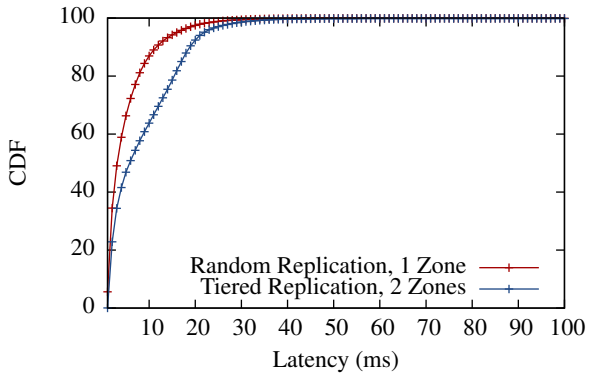


Figure 8: Comparison of write latency between Tiered Replication across two availability zones and Random Replication on a single availability zone under YCSB benchmark.

latency overhead, because it spreads the requests across two availability zones, and our implementation uses synchronous replication, which requires waiting for all replicas to be written before acknowledging the write.

5.3 Recovery Evaluation

We measured single node recovery times under two scenarios: a node failure in the primary tier and in the backup tier. We repeated the experiment from the first two benchmarks and ran YCSB workload B. Thirty seconds into the benchmark, we forcibly failed one node in the cluster, and initiated recovery thirty seconds later.

In the primary tier node failure experiment, it took approximately 80 seconds to recover the failed node. It then took another 48 seconds for the recovered node to synchronize missing data with nodes in the backup tier. This two-step reintegration process is required by chain replication: in the first step, the node performs state transfer with its predecessor, while in the second step, the node verifies its data with its successor. The recovered node only transfers to the backup tier nodes the data written during the thirty seconds of down time, and not the total data set.

When we repeated the same experiment and failed a node in the backup tier, it took 91 seconds to recover the node. Because this node is the last in the chain, the node can completely reintegrate with the cluster in one step. The recovery time for both of these experiments is similar to that of the default HyperDex replication algorithm.

5.4 Bandwidth Across Availability Zones

Tiered Replication relies on having two failure-independent locations, with a high bandwidth interconnect between them. Our testing shows that Amazon's availability zones are an ideal setup. They provide reliable links with high bandwidth and low latency. Using the same machines from the previous experiments, we conducted pairwise bandwidth tests using the iperf performance measuring tool. All servers in the cluster were able to communicate at 1.09Gbit/s (even when communicating across availability zones), which seems to be the maximum capability of the instances we purchased.

6. RELATED WORK

Several researchers have made observations that the MTTF under independent failures is much higher than from correlated failures [25, 38]. A LinkedIn field study reported no record of data loss due to independent node failures [7]. Google researchers have shown that the MTTF with three replicas under correlated failures is almost three orders of magnitude lower than the MTTF under independent node failures [14].

Google researchers developed an analysis based on a Markov model that computes the MTTF for a single stripe under independent and correlated failures. However, they did not provide an analysis for the MTTF of the entire cluster [14]. Nath et al. modeled the affect of correlated node failures and demonstrated that replication techniques that prevent data loss under independent node failures are not always effective for preventing correlated node failures [25]. In addition, several researchers have modeled the MTTF for individual device components, and in particular for disks [3, 18, 26, 31, 32].

Several replication schemes addressed the high probability of data loss under correlated failures. Facebook's HDFS implementation [2, 5] limits the scatter width of Random Replication, in order to reduce the probability of data loss under correlated failures. Copyset Replication [9] improved Facebook's scheme, by restricting the replication to a minimal number of copysets for a given scatter width. Tiered Replication is the first repli-

cation technique that not only minimizes the probability of data loss under correlated failures, but also leverages the much higher MTTF under independent failures to further increase the MTTF under correlated failures. In addition, unlike Copyset Replication, Tiered Replication can gracefully tolerate dynamic cluster changes, such as nodes joining and leaving and planned cluster power downs. It also supports chain replication and the ability to distribute replicas to different racks and failure domains, which is a desirable requirement of replication schemes [5, 24].

The common way to increase durability under correlated failures is to use geo-replication of entire cluster to a remote site [14, 21, 23, 35, 39]. Therefore, if the cluster was using three replicas, once it is geo-replicated, the storage provider will effectively use six replicas. Similarly, Glacier [16] and Oceanstore [20, 29] design an archival storage layer that provides extra protection against correlated failures by adding multiple new replicas to the storage system. While the idea of using archival replicas is not new, Tiered Replication is more cost-efficient, since does not require any additional storage for the backup: it migrates one replica from the original cluster to a backup tier. In addition, previous replication techniques utilize random placement schemes and do not minimize the number of copysets, which leaves them susceptible to correlated failures.

Storage coding is used for reducing the storage overhead of replication [17, 19, 22, 28, 30]. De-duplication is also commonly used to reduce the overhead of redundant copies of data [12, 27, 40]. Tiered Replication is fully compatible with any coding or de-duplication schemes for further reduction of storage costs of the backup tier. Moreover, Tiered Replication enables storage systems to further reduce costs by storing the third replicas of their data on a cheap storage medium such as tape, or hard disks in the case of an solid-state based storage cluster.

7. CONCLUSION

Cloud storage systems typically rely on three-way replication within a cluster to protect against independent node failures, and on full geo-replication of an entire cluster to protect against correlated failures. We provided an analytical framework for computing the probability of data loss under independent and correlated node failures, and demonstrated that the standard replication architecture used by cloud storage systems is not cost-effective. Three-way replication is excessive for protecting against independent node failures, and clearly falls short of protecting storage systems from correlated node failures. The key insight of our paper is that since the third replica is rarely needed for recovery from independent node failures, it can be placed on a geographically separated cluster, without causing a significant impact to the recovery time from independent node failures, which occur frequently in large clusters.

We presented Tiered Replication, a replication technique that automatically places the n-th replica on a separate cluster, while minimizing the probability of data loss under correlated failures, by minimizing the number of copysets. Tiered Replication improves the cluster-wide MTTF by a factor of 20,000 compared to random replication, without increasing the storage capacity. Tiered Replication supports additional data placement constraints required by the storage designer, such as rack awareness and chain replication assignments, and can dynamically adapt when nodes join and leave the cluster. An implementation of Tiered Replication on HyperDex, a key-value storage system, demonstrates that it incurs a small performance overhead.

8. APPENDIX

This section contains the closed-form solution for the Markov chain described in Section 2 and Figure 1 with an infinite number of nodes. The state transitions the Continuous-time Markov chain state i are:

$$i \cdot \mu \cdot Pr(i) = \lambda \cdot Pr(i-1)$$

Therefore:

$$Pr(i) = \frac{\rho}{i} Pr(i-1)$$

Where $\rho = \frac{\lambda}{\mu}$. If we apply this formula recursively:

$$Pr(i) = \frac{\rho}{i} Pr(i-1) = \frac{\rho^2}{i \cdot (i-1)} Pr(i-2) = \frac{\rho^i}{i!} Pr(0)$$

In order to find $Pr(0)$, we use the fact that the sum of all the Markov state probabilities is equal to 1:

$$\sum_{i=0}^{\infty} Pr(i) = 1$$

If we apply the recursive formula:

$$\sum_{i=0}^{\infty} Pr(i) = \sum_{i=0}^{\infty} \frac{\rho^i}{i!} Pr(0) = 1$$

Using the equality $\sum_{i=0}^{\infty} \frac{\rho^i}{i!} = e^{\rho}$, we get: $Pr(0) = e^{-\rho}$. Therefore, we now have a simple closed-form formula for all of the Markov state probabilities:

$$Pr(i) = \frac{\rho^i}{i!} e^{-\rho}$$

References

[1] How isolated are availability zones from one another? http://aws.amazon.com/ec2/faqs/#How_isolated_are_Availability_Zones_from_one_another.

[2] Intelligent block placement policy to decrease probability of data loss. https://issues.apache.org/jira/browse/HDFS-1094.

[3] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 289–300, New York, NY, USA, 2007. ACM.

[4] M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger. On correlated failures in survivable storage systems. Technical report, DTIC Document, 2002.

[5] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[7] R. J. Chansler. Data Availability and Durability with the Hadoop Distributed File System. *;login: The USENIX Magazine*, 37(1), February 2012.

[8] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. *NSDI*, 6:4–4, 2006.

[9] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[11] J. Dean. Evolution and future directions of large-scale storage and computation systems at Google. In *SoCC*, page 1, 2010.

[12] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: A scalable secondary storage. In *FAST*, volume 9, pages 197–210, 2009.

[13] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[14] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[16] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *IN PROC. OF NSDI*, 2005.

[17] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[18] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage*, 4(3):7:1–7:25, Nov. 2008.

[19] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: a durable and practical storage system. In *USENIX Annual Technical Conference*, pages 129–142, 2007.

[20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: an architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[22] R. Li, P. P. Lee, and Y. Hu. Degraded-first scheduling for MapReduce in erasure-coded storage clusters.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation*, 2013.

[24] J. MacCormick, N. Murphy, V. Ramasubramanian, U. Wieder, J. Yang, and L. Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Transactions On Storage (TOS)*, 4(4):11, 2009.

[25] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *NSDI*, volume 6, pages 225–238, 2006.

[26] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 17–29, 2007.

[27] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[28] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX, 2013.

[29] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, volume 3, pages 1–14, 2003.

[30] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 325–336. VLDB Endowment, 2013.

[31] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[32] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, 0:1–10, 2010.

[34] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability. In *IPTPS*, 2006.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[36] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. *Proceedings of Eurosys 11*, pages 169–182, 2011.

[37] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[38] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *USENIX WORLDS*, 2004.

[39] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

[40] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

Callisto-RTS : Fine-Grain Parallel Loops

Tim Harris
Oracle Labs

Stefan Kaestle
ETH Zurich

Abstract

We introduce Callisto-RTS, a parallel runtime system designed for multi-socket shared-memory machines. It supports very fine-grained scheduling of parallel loops—down to batches of work of around 1K cycles. Fine-grained scheduling helps avoid load imbalance while reducing the need for tuning workloads to particular machines or inputs. We use per-core iteration counts to distribute work initially, and a new asynchronous request combining technique for when threads require more work. We present results using graph analytics algorithms on a 2-socket Intel 64 machine (32 h/w contexts), and on an 8-socket SPARC machine (1024 h/w contexts). In addition to reducing the need for tuning, on the SPARC machines we improve absolute performance by up to 39% (compared with OpenMP). On both architectures Callisto-RTS provides improved scaling and performance compared with a state-of-the-art parallel runtime system (Galois).

1 Introduction

Callisto-RTS is a parallel runtime system for multi-socket shared-memory machines. We focus on supporting graph analytics workloads such as PageRank [24] and betweenness centrality (BC) [17]. These workloads are increasingly important commercially, and are the focus of benchmarking efforts [1, 29] along with myriad single-machine systems (such as Galois [21] and GreenMarl [12]) plus distributed systems (such as Grappa [20], Naiad [19], and Pregel [18]). It can be difficult to exploit parallelism in these workloads because of the difficulty of achieving good load balance in combination with low synchronization overhead.

As a running example, consider a PageRank superstep (Figure 1). The outer loop (`t`) ranges over the vertices. Within each iteration, `w` ranges over the vertices adjacent to `t` and updates the new PageRank value for `t` based on the current value for `w`. Using OpenMP [22] as an example, the pragma indicates that chunks of `BATCH_SIZE` iterations of the outer loop should be assigned dynamically to threads. Typically, implementations do this assignment using an atomic fetch-and-add on a shared counter.

```
#pragma omp for schedule(dynamic, BATCH_SIZE)
for (node_t t = 0; t < G.num_nodes(); t ++) {
    double val = 0.0;
    for (edge_t w_idx = G.r_begin[t];
         w_idx < G.r_begin[t+1]; w_idx ++)  {
        node_t w = G.r_node_idx [w_idx];
        val += G_pg_rank[w] /
               (G.begin[w+1] - G.begin[w]);
    }
    G_pg_rank_nxt[t] = (1 - d) / N + d * val;
}
```

Figure 1: PageRank loop, with `t` ranging over vertices.

Setting `BATCH_SIZE` introduces a trade-off. Setting it too large risks load imbalance with threads taking large batches of work and some threads finishing before others. Setting it too small introduces synchronization overheads. It is difficult to set `BATCH_SIZE` optimally: The distribution of work between iterations is uneven—for instance, in a social network, a celebrity has millions of times more neighbors than the average. Even if the iterations are divided evenly, the work performed by each thread can be uneven. In some cases, the number of instructions executed by each thread may be the same, but the execution times differ based on differing memory access times and cache locality.

With Callisto-RTS we reduce the need for tuning by making it efficient to select a very small `BATCH_SIZE` while still achieving good performance and scalability. Concretely, on machines with 1024 h/w contexts, we achieve good performance down to batches of around 1K cycles (compared with 200K cycles using dynamically-scheduled OpenMP loops).

Section 2 describes our programming model. We provide nested parallel loops, with control over how the h/w contexts in the machine are allocated to different levels of the loop hierarchy—for instance, an outer loop may

run with one thread per core, leaving additional threads per core idle until an inner level of parallelism is reached. This *non-work-conserving* approach to nesting can lead to better cache performance when iterations of the inner loop share state in a per-core cache.

Section 3 describes our techniques for fine-grained scheduling. We use a combining mechanism to allow threads requesting new work to aggregate their requests before accessing a shared loop iteration counter (e.g., combining requests via local synchronization in a core's L1$). In addition, we introduce a new asynchronous combining scheme in which a thread issues a request for new work while executing its current work: this provides more time for combining to occur. Furthermore, combining can be achieved with ordinary read/write operations, reducing the need for atomic read-modify-writes.

In Section 4 we evaluate the performance of Callisto-RTS. We use a 2-socket Intel 64 system (having 32 h/w contexts). We also use an 8-socket T5 SPARC system (having 1024 h/w contexts).

In addition to comparing with OpenMP, we compare our PageRank results with Galois, a state-of-the-art system based on scalable work-stealing techniques [21]. In contrast to work-stealing, we show that the shared-counter representation we use for parallel work enables single-thread performance improvements of 5%–26%. The asynchronous combining technique enables improved scalability on both processor architectures.

Section 5 discusses related work, and in particular, task-parallel models such as Cilk [8] and Intel Threading Building Blocks (TBB) [27]. Callisto-RTS differs from these systems in two main ways: First, compared with work-stealing, our implementation is specialized to distributing batches of loop iterations via shared counters. We use request aggregation to reduce contention on these counters rather than using thread-local work queues. Our approach avoids reifying individual batches of loop iterations as entries in work queues (as in Galois [21]), or requiring memory fence instructions (as in typical thread-safe work queues).

Second, we exploit the structure of the machine in the programming model as well as the runtime system. Our non-work-conserving approach to nesting contrasts with work-stealing implementations of task-parallelism in which all of the idle threads in a core would start additional iterations of the outer loop. In workloads with nested parallelism, our approach aims to reduce cache pressure when different iterations of an outer loop have their own iteration-local state: it can be better to have multiple threads sharing this local state, rather than extracting further parallelism from the outer loop.

As we say in Section 6, we hope that our techniques can be incorporated in runtime systems for other parallel programming models in the future.

2 Programming model: parallel loops

In this section we introduce the API supported by Callisto-RTS. Our initial workloads are graph analytics algorithms generated by a compiler from the Green-Marl DSL [12]. Therefore, while we aim for the syntax to be reasonably clear, our main goal is performance.

2.1 Flat parallelism

Callisto-RTS is based on parallel loops. As with OpenMP, and other systems, the programmer must ensure that iterations are safe to run concurrently. Loops are expressed using C++ templates, specializing a `parallel_for` function according to the type of the iteration variable and the loop body. Currently, all of the loops we support distribute their iterations across the entire machine (as with OpenMP `dynamic` loops). This reflects the fact that our graph algorithms typically have little temporal or spatial locality in their access patterns. In this setting, we are concerned more by reducing contention in the runtime system, and achieving good utilization of the h/w contexts across the machine and their associated memory.

A parallel loop to sum the numbers $0 \ldots 10$ is written:

```
struct example_1 {
  atomic<int> total {0}; // 0-initialized atomic
  void work(int idx) {
    total += idx;        // Atomic add
} } e1;

parallel_for<example_1, int>(e1, 0, 10);
cout << e1.total;
```

The `work` function provides the body of the loop. The `parallel_for` is responsible for distributing work across multiple threads. The struct `e1` is shared across the threads. Hence, due to the parallelism, atomic add operations are needed for each increment.

Per-thread state can be used to reduce the need for atomic operations. This per-thread state is initialized once in each thread that executes part of the loop, and then passed in to the `work` function:

```
struct per_thread { int val; };

struct example_2 {
  atomic<int> total {0}; // 0-initialized atomic

  void fork(per_thread &pt) { pt.val = 0; }

  void work(per_thread &pt, int idx) {
    pt.val += idx;       // Unsynchronized add
  }

  void join(per_thread &pt) {
    total += pt.val;     // Atomic add
} } e2;

parallel_for<example_2, per_thread, int>(e2,0,10);
cout << e2.total;
```

In this example the `fork` function is responsible for initializing the per-thread counter. The `work` function then operates on this per-thread state. The `join` function uses an atomic add to combine the results.

Design rationale. We considered whether to use C++ closures for loop bodies. Closures provide simpler syntax for short examples, and permit variables to be captured by reference in the `work` function. Unfortunately, performance using current compilers appears to depend a great deal on the behavior of optimization heuristics. We hope that future compiler implementations may provide more consistent performance in this regard.

For simplicity we have an implicit barrier at the end of each loop. This reflects the fact that, for our workloads, there is abundant parallel work, plus the fact that our implementation techniques are effective in reducing load imbalance (meaning that threads tend to arrive at the end of the loop at approximately the same time). We assume that Callisto-RTS runs within an environment where it has exclusive use of h/w contexts, and so thread preemption is not a concern.

In more variable multiprogrammed environments, dynamic techniques such as the prior work of Harris *et al.* [10], or abstractions and analyses such as those of Vajracharya and Grunwald may mitigate straggler problems [32].

Implementation. We initially describe the implementation with a single level of parallelism (we discuss nesting in Section 2.2). A set of worker threads is created at startup. A designated *leader* starts the `main` function. Other *follower* threads wait for work.

The definition of `parallel_for` instantiates a `work_item` object and publishes it via a shared pointer being watched by the followers. The work item has a single `run` function containing a loop which claims a batch of iterations before calling the workload-specific loop body. This repeats until there are no more iterations. A reference to the loop's shared state is held in the work item. Any per-thread state is stack-allocated within `run`. Consequently, only threads that participate in the loop will need to allocate per-thread state.

The thread which claims the last batch of iterations removes the work item from the shared pointer (preventing additional threads needlessly starting it). Finally, each work item holds per-socket counts of the number of active threads currently executing the item. The main thread waits for these counters to all be 0, at which point it knows that all of the iterations have finished execution.

Process termination is signaled by the leader publishing a designated "finished" work item. This approach means that a worker can watch the single shared location both for new work and for termination.

2.2 Nested parallelism

Parallel loops can be nested within one another, and Callisto-RTS provides control over the way in which h/w contexts are allocated to different levels. The workloads we target have a small number of levels of parallelism, dependent on the algorithm rather than on its input. For instance, our betweenness centrality workload (BC) uses an outer level to iterate over vertices, and then an inner level to implement a parallel breadth-first search (BFS) from each vertex.

Selecting which of these levels to run in parallel depends on the structure of the hardware. In the BC example, parallelizing just at the outer level can give poor performance on multi-threaded cores because multiple threads' local BFS states compete for space in each per-core cache. Conversely, parallelizing just at the inner level gives poor performance when the BFS algorithm does not scale to the complete machine. A better approach is to use parallelism at both levels, exploring different vertices on different cores, and using parallel BFS within a core.

A loop indicates how many levels are nested inside it. That is, a loop at level 0 is an inner loop with no further parallelism. A loop at level 1 encloses one level of parallelism, and so on.

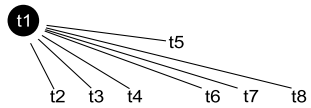
Concretely, writing `parallel_for` is short for a loop at level 0. For a loop at level N we write:

```
outer_parallel_for<...>(N, ...);
```

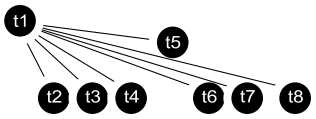
Design rationale. This "inside out" approach to counting levels provides composability. A leaf function using parallelism will always be at level 0, irrespective of the different contexts it may be called from.

If we numbered levels "outside in", or assigned them dynamically, then it would not be possible to distinguish (*i*) reaching an outer loop which should be distributed across all h/w contexts, versus (*ii*) an outer loop which should just be distributed at a coarse level leaving some idle h/w contexts for use within it. A given program may have loops with different depths of nesting—e.g., a flat initialization phase at level 0 over all h/w contexts, while a subsequent computation may start at level 1 and just be distributed at a per-socket granularity.

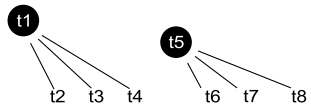
Implementation. Environment variables set how nesting levels map to the machine—e.g., indicating that loops at level 0 should be distributed across all h/w contexts, and that level 1 should be distributed across cores, core-pairs, sockets, or some other granularity. This flexibility lets a program express multiple levels of parallelism on large NUMA machines, but execute more simply on smaller systems.



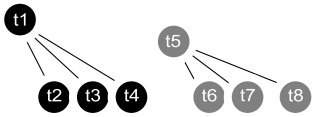
(a) Thread t1 executes sequential code. Other threads wait for work.



(b) Thread t1 enters a loop at level 0. All threads participate in the loop.



(c) Thread t1 enters a loop at level 1. Thread t1 and t5 participate. Threads t6–t8 now wait for work from t5.

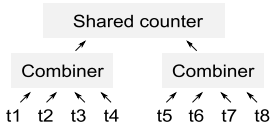


(d) Threads t1 and t5 enter loops at level 0, threads participate in the respective loops.

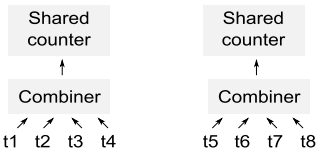
Figure 2: Allocation of threads to loops. Thread t1 is at the top level, t5 at level 1, and other threads at level 0. This allocation might be appropriate in a 2-socket machine with 4 threads per socket.

Based on this, threads are organized into a tree which selects which threads participate in which loops. Each thread has a *level* in this tree, and a *parent* at the next non-empty level above it (aside from a designated top-level thread which forms the root of the tree). Dynamically, each thread has a *status* (leading or following). Initially, the root is leading and all other threads following. A thread's *leader* is the closest parent with leading status (including the thread itself). A thread at level n becomes a leader if it encounters a loop at level $k \leq n$. A follower at level n executes iterations from a loop if its leader encounters a loop at level $k \leq n$; otherwise, it remains idle.

Figure 2 illustrates this dynamically with a possible organization of 8 threads across 2 sockets. The main thread is t_1 and is the parent to $t_2 \ldots t_4$ in its own socket (level 0), and t_5 in the second socket (level 1). In turn, t_5 is parent to $t_6 \ldots t_8$. Initially t_1 is the only active thread and hence leader to all of the threads $t_1 \ldots t_8$ (Figure 2a). If t_1 encounters a loop at level 0 then all threads participate in the same loop (Figure 2b). If, instead, t_1 encounters a loop at level 1 then just t_1 and t_5 participate (Figure 2c).



(a) Distribution during a level-0 loop led by t_1 in which all threads participate, using separate request combiners on each socket.



(b) Distribution during a level-0 loop led by t_1 (left) and by t_5 (right).

Figure 3: Work scheduling in different loops. A top-level loop spans the complete machine, with local requests for work being combined in each set of nearby threads. Multiple instances of an inner loop may run concurrently on the two parts of the machine.

If t_5 then encounters a new loop at level 0 then it becomes a leader of $t_5 \ldots t_8$ (Figure 2d).

3 Work scheduling

We now introduce our techniques for distributing iterations. We take a hierarchical approach to defining work scheduling policies, with a number of basic policies that can be combined to form more complex variants. An individual thread makes a request to the leaves of a tree of work distributors, and the implementation of this may involve a call to a higher level distributor, and so on.

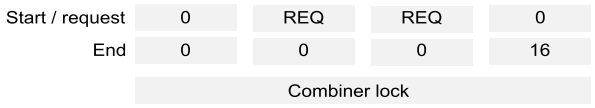
Our hierarchical approach lets us reflect the structure of the machine within the hierarchy used for work scheduling. In addition, it lets us explore a range of complex policies—for instance, exploring whether data structures should be per-core, per-L2$, or per-socket, Figure 3 illustrates this using the example 8-thread machine. Separate work distributors are used for each parallel loop—for instance, the 4-thread loop led by t_1 is handled separately from the 4-thread loop led by t_5.

Shared counter. The simplest work distributor is a single shared counter, initialized with loop bounds, and with threads claiming iterations using an atomic fetch-and-add. We include this initial implementation to reflect the techniques used for dynamically scheduled loops in many OpenMP runtime systems.

Distributed counters. The iteration space is distributed evenly across a number of stripes according to the number of sockets, cores, or threads within the machine. Each thread is associated with a *home* stripe (e.g., with per-socket distribution, this would correspond to the thread's socket). In addition, each thread has a *current* stripe. A thread claims iterations by an atomic increment on its current stripe until that portion of the iteration space has been completed. At that point it moves on to the next stripe, and so on until it returns to its home stripe.

Request combining. Request combiners attempt to aggregate requests for work which are made "nearby" in time and in the machine. Rather than have multiple threads across the machine compete for atomic updates to a single cache line, sets of threads can compete at a finer granularity, and then a smaller number of threads compete at a global level. This reduces the number of atomic read-modify-write instructions, and it increases the likelihood that contention remains in a local cache.

Each thread using a combiner has a *slot* comprising a pair of loop indices (`start/request`, and `end`). For instance, in a 4-slot combiner:



Slot $(0,0)$ is quiescent. Slot $(\texttt{REQ},0)$ represents a request for work. Slot $(0,16)$ represents supplied work (in this case the iterations $0\ldots16$). In addition, each combiner has a lock which needs to be held by a thread collecting requests to make to the upstream counter. In pseudocode:

```
  my_slot->start = REQ; // Issue request

  while (1) {
    // Try to acquire the combiner lock
    if (!spinlock_tryacquire(&my_combiner->lock)) {
      // Lock busy.  Wait for it to be released, then
      // test if we received work.
      while (spinlock_is_held(&my_combiner->lock)) {
      }
    } else {
      // We acquired combiner lock, collect requests
      // from other threads, issue aggregate request,
      // distribute work, and then release lock.
      ...
      spinlock_release(&my_combiner->lock);
    }
    // Test if request has been satisfied
    if (my_slot->start != REQ) {
      return (my_slot->start, my_slot->end);
} }
```

A thread starts by writing `REQ` in its slot and then trying to acquire the lock. If the lock is already held then the current thread waits until the lock is available, and tests if its request has been satisfied. Note that the `REQ` flag is set without holding the lock, and so the lock holder is not guaranteed to see the thread's request. If a thread succeeds in acquiring the lock it scans the other slots for `REQ` and issues an upstream request for a separate batch of iterations for each requester (for brevity we omit the pseudocode for this). Work is distributed by writing to the `end` field and then overwriting `REQ` in the `start` field. Hence, on a TSO memory model, a thread receiving work sees the start-end pair consistently once `REQ` is overwritten.

If all threads using a combiner share a common L1$ then the request slots are packed onto as few cache lines as possible, Otherwise, each slot has its own line. Combiners can be configured in various ways. For instance, threads within a core could operate with a per-core combiner, and then additional levels of combining could occur at a per-L2$ level (if this is shared between cores), or at a per-socket level. We examine some of these alternatives in our evaluation (Section 4).

Asynchronous combining. With asynchronous combining, a thread sets its request flag *before* executing its current batch, rather than after finishing it. This asynchrony exposes a request over a longer interval: other threads using the same combiner can handle the request while the thread's current batch is being executed.

In the best case, in a set of n threads, all but 1 will find they have received new work immediately after finishing their current batches. Furthermore, if additional combining occurs, then it increases the size of the aggregate requests issued from the combiner (reducing contention on the next level in the work scheduling tree), and it reduces contention on the lock used within the combiner (if most threads receive work then they never need to acquire the lock). The fast-path for the $n-1$ threads receiving work is (*i*) reading the work provided to them, and (*ii*) setting their request flag. On a TSO memory model this avoids fences or atomic read-modify-write instructions.

4 Evaluation

We evaluate the performance of Callisto-RTS using machines with two different processor architectures:

Intel 64. We use an Oracle X4-2 machine. This is a 2-socket machine with Intel Xeon E5-2650 IvyBridge processors. Processors have a per-socket L3$, and per-core L2$ and L1$. Each core provides 2 h/w contexts for a total of 32 h/w contexts in the machine.

We use GCC 4.7.4 and Linux 2.6.32. We confirmed a subset of our results on Linux 3.14.33 but saw no difference: the runtime systems are set to employ user-mode synchronization using atomic instructions rather than `futex` system calls.

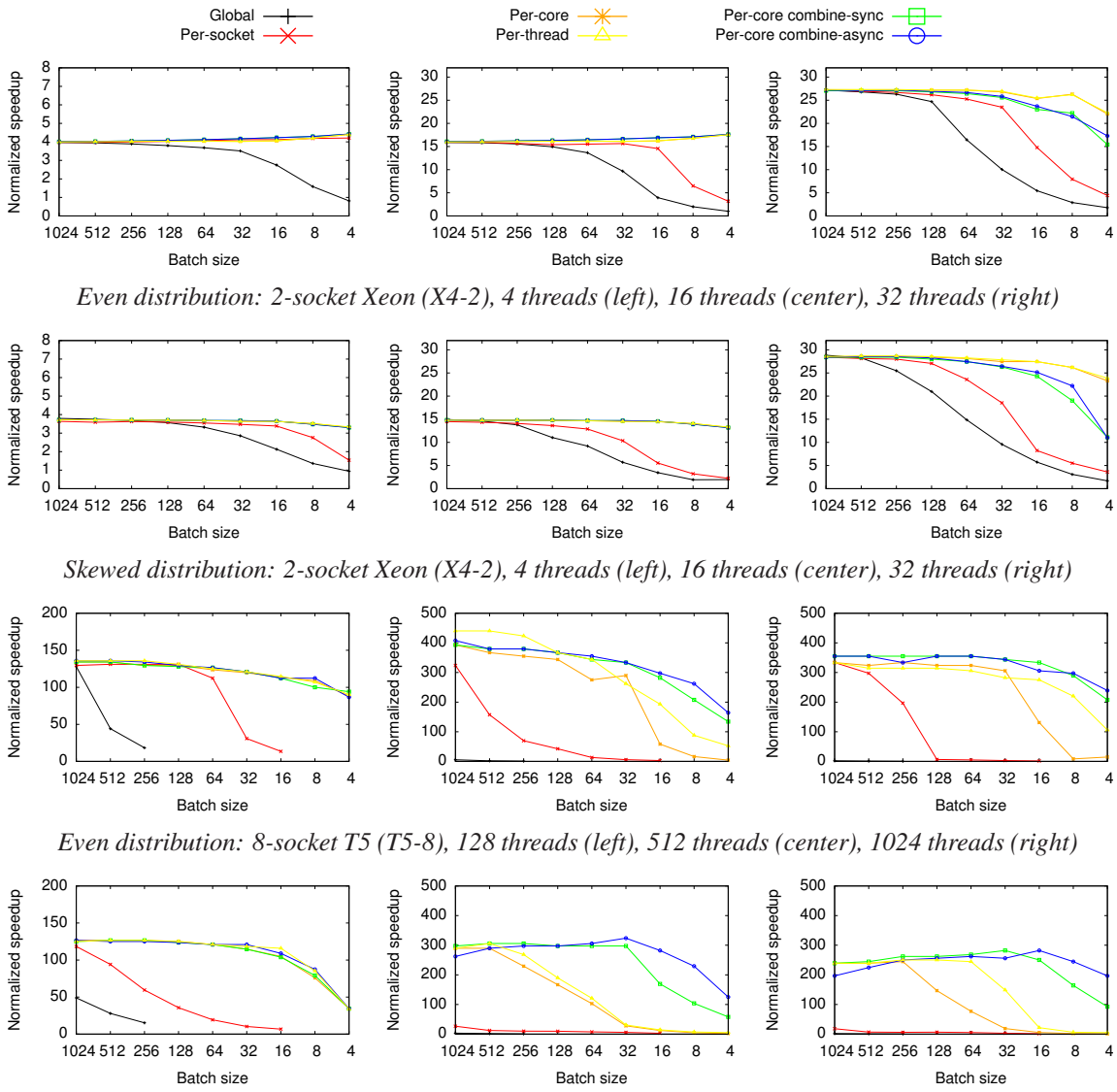


Figure 4: Microbenchmark scalability on X4-2 and T5-8 systems.

SPARC. We use an Oracle T5-8 machine. This is an 8-socket machine with SPARC T5 processors. Each socket has 16 cores, and each core supports 8 h/w contexts for a total of 1024 h/w contexts in the machine. As with the Intel 64 system, the T5-8 has per-socket L3$ caches, and per-core L2$ and L1$. We use Solaris Studio 12.4 on Solaris 11.2.

Both architectures provide atomic compare-and-swap (CAS). The Intel 64 architecture provides additional atomic operations such as fetch-and-add. Conversely, the T5 processor provides a user-mode-accessible `wrpause` instruction which lets a h/w context wait for a configurable number of cycles, avoiding it consuming pipeline resources while waiting. This can be important in the multi-threaded SPARC processor: when only a single h/w context is runnable, that context can issue instructions to multiple pipelines in each clock cycle. On the T5-8 we use `wrpause` for 128 cycles in loops which are expected to unblock quickly (e.g., during request combining), and for approximately 4096 cycles in loops which are expected to unblock less quickly (e.g., waiting on entry to a loop).

We spread software threads as widely as possible

within the machine. We use OpenMP with active synchronization (i.e., spinning, rather than blocking in the OS). For each algorithm-machine combination, the fastest result is achieved with active synchronization rather than blocking. We report median-of-3 results.

We use three evaluation workloads: a scalability microbenchmark (Section 4.1), graph algorithms with a single level of parallelism (Section 4.2), and an additional graph workload using nested parallelism (Section 4.3).

4.1 Work scheduling microbenchmarks

We start using a microbenchmark with a single large loop. Each iteration performs a variable amount of work (incrementing a stack-allocated variable a set number of times). We can vary (*i*) the number of increments used in the different iterations, (*ii*) the number of threads, (*iii*) the work scheduling mechanism we use, and (*iv*) the batch size in which threads claim work. We investigate two ways of organizing the work within the loop:

Even distribution. Here, each iteration performs the same amount of work: good load balancing can be achieved by splitting the iteration space evenly. We evaluate six scheduling techniques: a single shared counter, distributed counters at per-socket, per-core, and per-thread granularities, and finally per-core work combiners coupled with per-core counters (Figure 4). For each machine we show a workload with a modest number of threads (left column), and then a workload with 1 thread per core (center column), and a workload with all h/w contexts in use (right column). We plot the speedup relative to unsynchronized sequential code on the same machine.

On the Intel 64 system, a single iteration is around 50 cycles. The per-core and per-thread counters perform well across the experiments. Request combining performs slightly worse than simple per-thread or per-core counters: little combining occurs with only two threads per core.

On the SPARC system, each iteration is around 140 cycles. At large batch sizes, we see good scaling to 512 h/w contexts. The number of instructions per cycle is 0.34 and so, with 2 pipelines per core, we would expect to saturate the cores with 750 threads. Beyond this point, contention between threads for pipeline resources can limit performance. We believe this is an example workload where the user-mode `mwait` instruction in the future SPARC M7 processor [25] could provide improved scalability—unlike `wrpause`, the `mwait` instruction permits a thread to monitor a memory location while waiting, rather than needing to pick a specific interval in advance.

Combining shows slight benefits at high thread counts and low batch sizes. As expected, the CAS loop used to increment the counters starts to need re-execution under higher contention (on Intel 64 we can use an atomic fetch-and-add). Re-execution consumes pipeline resources that could otherwise be used productively.

Asynchronous combining generally aggregates requests from all of the active threads in a core irrespective of the batch size used (e.g., with 256 threads, there are 2 threads per core, and each combined request is for 2 batches). Synchronous combining is effective only when the batch sizes are small, making requests more likely to "collide".

Skewed distribution. With the skewed work distribution, the first *n* iterations each contain 1024x the work of the others. We set *n* so that the total work across all iterations is the same as the even distribution. The aim is to study the impact of different work scheduling techniques when there is contention in the runtime system: threads which start at the "light" end of the iteration space will complete their work quickly and start to contend for work with threads at the "heavy" end.

On the Intel 64 system, per-core and per-thread counters perform well. As with the even workload, two threads per core provides little opportunity for combining.

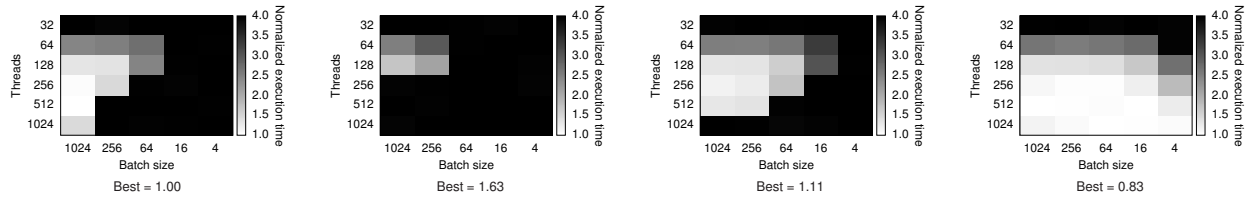
On the SPARC systems, the use of combining has significant benefits at high thread counts (512 or 1024), with some additional benefit from asynchronous combining. The skewed workload means that we see CAS failures and re-execution when incrementing shared counters. In contrast, per-core combining allows most threads to request work by setting their request flag (which remains core-local in the L1$) and then waiting "politely" using `wrpause`.

Summary. Based on these results, we use per-thread counters as the default on Intel 64, and per-core counters with asynchronous combining on SPARC.

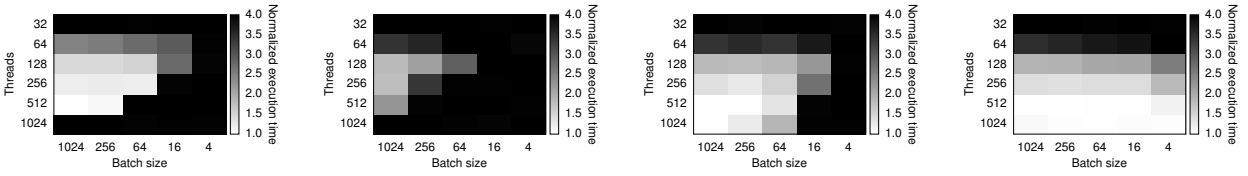
In addition to the results shown here we explored two-level combining schemes in which threads combine requests within a core, before using a further level of combining within a socket. We saw combining occurring at both levels, but the the overall benefits from reduced contention did not offset the cost of the additional operations used. Per-core combining with per-core counters performed better across all workloads, and so we omit the two-level results.

4.2 Graph algorithms

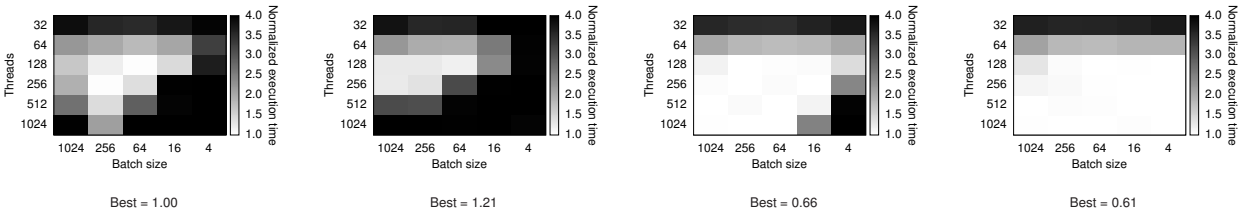
We now evaluate Callisto-RTS using the PageRank and Triangle Counting algorithms from Green-Marl [12] (Figure 5). In Section 4.3 we use a betweenness central-



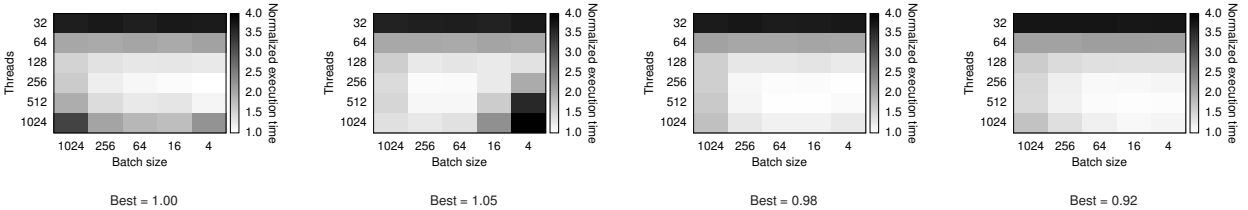
T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), PageRank – LiveJournal. The best OpenMP execution took 0.26s (512 threads, 1024 batch size).



T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), PageRank – Twitter. The best OpenMP execution took 6.0s (512 threads, 1024 batch size).



T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), Triangle counting – LiveJournal. The best OpenMP execution took 0.21s (256 threads, 256 batch size).



T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), Triangle counting – Twitter. The best OpenMP execution took 55.3s (256 threads, 4 batch size).

Figure 5: Graph algorithms on LiveJournal (4.8M vertices) and Twitter (42M vertices). Execution times normalized to the best OpenMP result. Below each plot we show the ratio of the best configuration's execution time to the best OpenMP result.

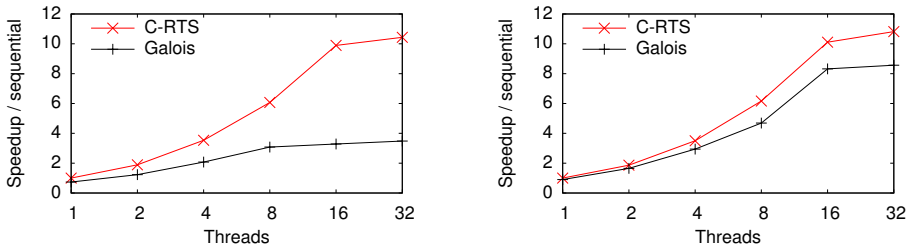
ity algorithm [17] as an example with nested parallelism.

We use the SNAP LiveJournal dataset (4.8M vertices, 69M edges) [16] and the Twitter data set of Kwak *et al.* (42M vertices, 1.5B edges) [14].

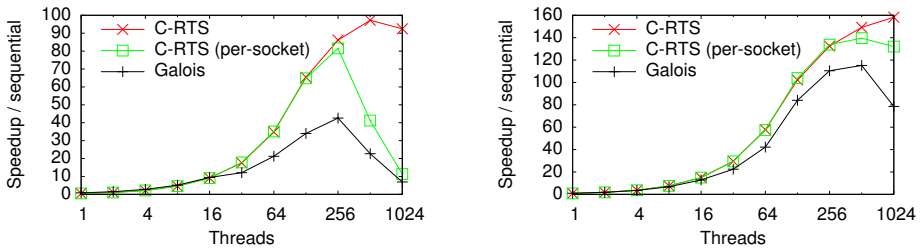
We focus on the SPARC machine. As the microbenchmark results illustrated, the smaller 2-socket Intel 64 system does not exhibit a great deal of sensitivity to work scheduling techniques with per-thread counters.

For each machine-algorithm combination we show: the original OpenMP implementation, and then Callisto-RTS using a single global counter, per-socket counters, and per-core counters with asynchronous combining. Each plot shows the performance of the given technique across thread counts (32…1024), and batch sizes (1024…4). Each square shows the execution time, normalized to the best OpenMP result. Below each plot, we show the time of the best configuration, normalized to the best OpenMP result. Note that the dark rows at the top of the plots indicate there are insufficient threads to perform well on these scalable workloads, even with perfect work scheduling and no overheads.

On the LiveJournal input, careful tuning is needed to get good performance with OpenMP or with a single counter: different numbers of threads are best for the dif-



2-socket Xeon (X4-2), PageRank with LiveJournal input (left) and Twitter input (right).



8-socket T5 (T5-8), PageRank with LiveJournal input (left) and Twitter input (right).

Figure 6: PageRank on Callisto-RTS and Galois.

ferent algorithms, and there is a sharp fall-off in performance if the best configuration is not selected.

The OpenMP implementations often perform better than Callisto-RTS using a single global counter. This is because they use `static` scheduling on some loops where work is known to be distributed evenly (e.g., copying from one array to another). Static scheduling works well on such loops, but not on the main parts of the algorithm.

Using a single global counter leads to poor performance at small batch sizes, and work imbalance with large batches. Per-socket counters provide significant improvement at smaller batch sizes. As in the microbenchmark, per-core counters with asynchronous combining provide good performance over a wide range of configurations. We see similar trends on the Twitter input.

Comparison with Galois. The Galois system is a lightweight infrastructure for parallel in-memory processing. In prior work, Nguyen *et al.* demonstrated that Galois has good performance and scalability across a range of graph benchmarks [21]. We use version 2.2.1. We adapted the Galois PageRank code to use the same in-memory compressed sparse row representation as with Callisto-RTS. Compared with the Galois original, this modified implementation is faster across every test. We disabled concurrency control and confirmed that we obtained identical performance between Galois and Callisto-RTS. Thread placement is identical between the two runtime systems. We use Galois' default batch size (32) in both systems.

Figure 6 shows the resulting performance on the X4-2 and T5-8. All results are normalized to the single-threaded implementation without concurrency control. Callisto-RTS performs better on both machines and both inputs.

On the X4-2, Callisto-RTS scales similarly on both graphs up to 16 threads (1 thread per core), with a slight additional benefit from hyperthreading. Galois scales well on the Twitter graph, with 15-20% overhead compared with Callisto-RTS. Galois does not scale well on the LiveJournal graph with shorter loop iterations. Both differences are due to the way Galois distributes chunks of work. Each chunk is reified in memory as a block listing the iterations to execute, with each thread holding a current working block, and per-socket queues of blocks. On the smaller graph, the iterations are short-running and contention on per-socket queues appears to limit scaling. On the Twitter graph, each iteration is longer and contention less significant. However, the inner loop of fetching an iteration and executing it remains slower than with Callisto-RTS.

We see similar trends on the T5-8. Galois and Callisto-RTS both scale well to 128 threads (1 per core), as does the additional Callisto-RTS variant using per-socket iteration counters. Beyond this point, Callisto-RTS continues to scale well with asynchronous work distribution, whereas the other implementations are harmed by contention between threads when distributing

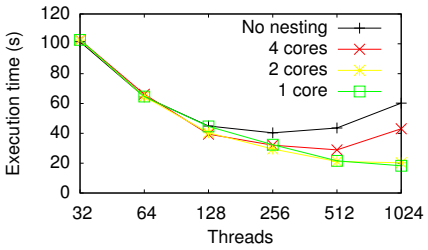


Figure 7: Betweenness centrality using nested parallelism at different levels.

work. On both graphs, Callisto-RTS performs well using the complete machine of 1024 threads.

Summary. Compared with the OpenMP implementation, using per-core counters with asynchronous combining improves the best-case performance in all four of the workloads in Figure 5 by 6%, 8%, 17%, and 39%. In addition, and perhaps more significantly, the performance achieved is more stable over different thread and batch settings, and does not require the programmer to select between static and dynamic scheduling.

4.3 Nested parallelism

Our final results use nested parallelism to compute betweenness centrality [17]. For each vertex, the computation executes breadth-first-search (BFS) traversals. The execution time can be large even for a modestly sized graph. We use the SNAP Slashdot data set [16] (82.1K vertices, 948K edges). Figure 7 compares flat parallelism (in which we process each vertex sequentially), versus nested parallelism at different levels. We use a parallel BFS algorithm with 13 different parallel loops, some initializing per-BFS data structures, and others performing parallel expansion of the next level of vertices. There is a barrier in between each loop (just between the threads executing that BFS, rather than system-wide).

On this workload, flat parallelism scales well to the level of 1 thread per core (128 threads on the T5-8 SPARC system). We see little improvement from further threads, and then some degradation at 512…1024 threads. We recorded values from the SPARC CPU performance counters. With 1 thread per core, 9.8% of load instructions miss in the L2-D$. With flat parallelism, this rises steadily to 29% with 8 threads per core.

We obtain the best performance using nesting within a single core, corresponding to the L2-D$ in this machine. Using nested parallelism, the miss rate rises only slightly to 10.8% when moving from 128 to 1024 threads.

In addition to the results shown here we tried (*i*) nested parallelism at a per-socket level, and (*ii*) parallelism only at the inner level in the BFS algorithm. Both of these alternatives were substantially worse than flat parallelism.

5 Related work

We discuss related work under three sections: programming models providing parallel loops, implementations of task parallelism, and prior work on combining techniques:

Parallel loops. Our techniques could be used in implementations of programming models which include parallel loops. Examples include OpenMP [22], parallel loops in Intel Threading Building Blocks (TBB) [27], and the proposed C Parallel Language Extensions [6]. Currently, the GCC 4.9 OpenMP implementation uses a per-loop shared counter with atomic fetch-and-add. As our results show, this approach requires careful tuning.

Task-parallelism. Systems such as Cilk [8], TBB [27], Wool [7, 26], and the Java ForkJoin framework [15] support task-parallel programming by distributing lightweight tasks using work-stealing systems such as those of Blumofe *et al.* [3] or Chase-Lev [5]. Cilk and TBB provide parallel loops built over task-parallel abstractions, recursively decomposing loops until a minimum size is reached (analogous to the batch size).

Typically, the common execution path involves a thread taking a task from a local work queue, decomposing the task, pushing part of the task back onto the queue, and executing the extracted iterations. While these steps can remain local to a thread, they require an atomic operation or memory fence [2]. Our request combining technique avoids these operations (aside from the one thread performing the aggregate request). Asynchronous combining reduces our fast path to a read of the current batch, followed (without a fence) by a write for a new request.

Tzannes *et al.* [30, 31] observe that a thread can avoid repeated operations on a work-queue by only pushing tasks on to the queue when it is below a threshold size (if the queue is above this size then that indicates that other threads are busy because otherwise items from the queue would have been stolen).

Using work stealing provides the opportunity to benefit from large amounts of prior work on scalable implementations (dating back at least as far as the work of Burton and Sleep [4], and stretching to ongoing work such as that of Tzannes *et al.* [31]). As discussed in our evaluation, Galois is a state-of-the-art example of this kind of implementation, specialized to shared-memory NUMA systems, However, reifying each loop iteration as an entry in a work-stealing queue introduces storage and processing costs, especially when loops contain short iterations.

Combining algorithms. Many systems have used combining techniques in which operations are aggregated to reduce contention. Direct software implementations of early techniques such as combining trees [9] and combining funnels [28] have typically not performed well. Hendler *et al.* illustrate this in the evaluation of their *flat combining* technique for handling requests on a lock-based shared data structure [11]. As in our request combiners, with flat combining each thread has a structure to publish requests for work, and a lock which is held while collecting requests. Unlike our design, flat combining requires threads to watch both the lock and their own request—our approach allows threads to just watch the lock (enabling the use of instructions such as `mwait`), and we make requests asynchronously with working.

Oyama *et al.* described a technique in which a lock protects a data structure and threads add requests to a LIFO queue associated with the lock [23]. Each thread must perform a successful CAS on the head of the list, whereas we allow threads to publish requests by writing to a per-thread flag. We use combining within a core, and empirically the cost of scanning the flags is better than the cost of maintaining a list.

Klaftenegger *et al.* described a queue-based delegation model in which a thread making a write-only request can proceed concurrently with the request's execution [13]. Our specialized use of delegation avoids maintaining an explicit queue, and handles a read-modify-update operation involving aggregation as well as delegation.

6 Conclusions and future work

In this paper we have introduced runtime system techniques for supporting parallel loops with fine-grain work scheduling. We are able to scale down to batches of work of around 1000 cycles on machines with 1024 h/w contexts, and we are able to achieve good scaling with workloads where the distribution of work between loop iterations is skewed. In addition, on an example workload with nested parallelism, we were able to obtain further scaling by matching the point at which we switch to the inner level parallelism to the position of the L2-D$ in the machine. This lets multiple threads execute the inner loops while sharing data in their common cache.

We believe that the techniques used in Callisto-RTS are applicable to other parallel programming models. The combining techniques could be applied transparently in implementations of OpenMP dynamically scheduled loops—either with, or without, asynchronous combining.

In addition, the same techniques could be applied to work-stealing systems. It may be profitable to use per-core queues and for threads within a core to use combining to request multiple items at once. As with loop scheduling in Callisto-RTS, this may reduce the number of atomic operations that are needed, and enable asynchrony between requesting work and receiving it. Furthermore, using per-core queues with combining may make loop termination tests more efficient than with per-thread queues (typical termination tests must examine each queue at least once before deciding that all of the work is complete).

Finally, we see the trend toward increasingly non-uniform memory performance making it important to exercise control over how nesting maps to hardware. In Callisto-RTS we do this by explicit programmer control and non-work-conserving allocation of work to threads. Future systems could use feedback-directed techniques, or potentially static analyses.

Acknowledgments

We would like to thank the reviewers along with Gavin Bierman, Callum Cameron, Hassan Chafi, Nawal Copty, Dave Dice, Sungpack Hong, Daniel Goodman, Darryl Gove, Jinha Kim, Martin Maas, Zoran Radovic, Paul Thomson, Vasileios Trigonakis, and Georgios Varisteas for discussions and feedback on earlier drafts of this paper.

References

[1] ANGLES, R., BONCZ, P., LARRIBA-PEY, J., FUNDULAKI, I., NEUMANN, T., ERLING, O., NEUBAUER, P., MARTINEZ-BAZAN, N., KOTSEV, V., AND TOMA, I. The Linked Data Benchmark Council: A graph and RDF industry benchmarking effort. *SIGMOD Rec. 43*, 1 (May 2014), 27–31.

[2] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL' 11: Proceedings of the 38th Symposium on Principles of Programming Languages* (Jan. 2011), pp. 487–498.

[3] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM 46*, 5 (Sept. 1999), 720–748.

[4] BURTON, F. W., AND SLEEP, M. R. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (1981), pp. 187–194.

[5] CHASE, D., AND LEV, Y. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the 17th Symposium on Parallelism in Algorithms and Architectures* (July 2005), pp. 21–28.

[6] CPLEX STUDY GROUP, WG14. C extensions for parallel programming. Working draft, 2014-09-18, N1862.

[7] FAXÉN, K.-F. Wool—a work stealing library. *SIGARCH Computer Architecture News 36*, 5 (June 2009), 93–100.

[8] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1998), pp. 212–223.

[9] GOTTLIEB, A., GRISHMAN, R., KRUSKAL, C. P., MCAULIFFE, K. P., RUDOLPH, L., AND SNIR, M. The NYU ultracomputer—designing a MIMD, shared-memory parallel machine. In *ISCA '98: 25 Years of the International Symposia on Computer Architecture, Selected Papers* (June 1998), pp. 239–254.

[10] HARRIS, T., MAAS, M., AND MARATHE, V. J. Callisto: co-scheduling parallel runtime systems. In *EuroSys '14: Proceedings of the 9th ACM European Conf. on Computer Systems* (Apr. 2014).

[11] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat combining and the synchronization-parallelism tradeoff. In *SPAA '10: Proceedings of the 22nd Symposium on Parallelism in Algorithms and Architectures* (June 2010), pp. 355–364.

[12] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS '12: Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2012), pp. 349–362.

[13] KLAFTENEGGER, D., SAGONAS, K. F., AND WINBLAD, K. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 14: Parallel Processing – 20th International Conference* (Aug. 2014), pp. 572–583.

[14] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th International Conference on World Wide Web* (Apr. 2010), pp. 591–600.

[15] LEA, D. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 Conference on Java Grande* (June 2000), pp. 36–43.

[16] LESKOVEC, J., AND KREVL, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[17] MADDURI, K., EDIGER, D., JIANG, K., BADER, D. A., AND CHAVARRIA-MIRANDA, D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS '09: Proceedings of the International Symposium on Parallel and Distributed Processing* (May 2009), pp. 1–8.

[18] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *SIGMOD '10: Proceedings of the International Conference on Management of Data* (June 2010), pp. 135–146.

[19] MCSHERRY, F., ISAACS, R., ISARD, M., AND MURRAY, D. G. Composable incremental and iterative data-parallel computation with Naiad. Tech. Rep. MSR-TR-2012-105, Microsoft Research, 2012.

[20] NELSON, J., MYERS, B., HOLT, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Grappa: A latency-tolerant runtime for large-scale irregular applications. Tech. Rep. UW-CSE-14-02-01, University of Washington, 2014.

[21] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *SOSP '13: Proceedings of the 24th Symposium on Operating Systems Principles* (Nov. 2013), pp. 456–471.

[22] *OpenMP Application Program Interface, Version 4.0*. July 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[23] OYAMA, Y., TAURA, K., AND YONEZAWA, A. Executing parallel programs with synchronization bottlenecks efficiently. In *PDSIA '99: Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications* (July 1999), pp. 182–204.

[24] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[25] PHILLIPS, S. M7: Next generation SPARC. In *HotChips '14: A Symposium on High Performance Chips* (Aug. 2014).

[26] PODOBAS, A., BRORSSON, M., AND FAXÉN, K. A comparative performance study of common and popular task-centric programming frameworks. *Concurrency and Computation: Practice and Experience 27*, 1 (2015), 1–28.

[27] REINDERS, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[28] SHAVIT, N., AND ZEMACH, A. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing 60*, 11 (Nov. 2000), 1355–1387.

[29] SUZUMURA, T., UENO, K., SATO, H., FUJISAWA, K., AND MATSUOKA, S. Performance characteristics of Graph500 on large-scale distributed environment. In *IISWC '11: Proceedings of the International Symposium on Workload Characterization* (Nov. 2011), pp. 149–158.

[30] TZANNES, A., CARAGEA, G. C., BARUA, R., AND VISHKIN, U. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *PPoPP '10: Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming* (Jan. 2010), pp. 179–190.

[31] TZANNES, A., CARAGEA, G. C., VISHKIN, U., AND BARUA, R. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS: ACM Transactions on Programming Languages and Systems 36*, 3 (Sept. 2014), 10:1–10:51.

[32] VAJRACHARYA, S., AND GRUNWALD, D. Dependence driven execution for multiprogrammed multiprocessor. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing* (July 1998), pp. 329–336.

LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache

Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo
Peking University

Chen Ding
University of Rochester

Song Jiang
Wayne State University

Zhenlin Wang
Michigan Technological University

Abstract

The in-memory cache system is a performance-critical layer in today's web server architecture. Memcached is one of the most effective, representative, and prevalent among such systems. An important problem is memory allocation. The default design does not make the best use of the memory. It fails to adapt when the demand changes, a problem known as *slab calcification*.

This paper introduces locality-aware memory allocation (LAMA), which solves the problem by first analyzing the locality of the Memcached requests and then repartitioning the memory to minimize the miss ratio and the average response time. By evaluating LAMA using various industry and academic workloads, the paper shows that LAMA outperforms existing techniques in the steady-state performance, the speed of convergence, and the ability to adapt to request pattern changes and overcome slab calcification. The new solution is close to optimal, achieving over 98% of the theoretical potential.

1 Introduction

In today's web server architecture, distributed in-memory caches are vital components to ensure low-latency service for user requests. Many companies use in-memory caches to support web applications. For example, the time to retrieve a web page from a remote server can be reduced by caching the web page in server's memory since accessing data in memory cache is much faster than querying a back-end database. Through this cache layer, the database query latency can be reduced as long as the cache is sufficiently large to sustain a high hit rate.

Memcached [1] is a commonly used distributed in-memory key-value store system, which has been deployed in Facebook, Twitter, Wikipedia, Flickr, and many other internet companies. Some research also proposes to use Memcached as an additional layer to accelerate systems such as Hadoop, MapReduce, and even virtual machines [2, 3, 4]. Memcached splits the memory cache space into different *classes* to store variable-sized objects as *items*. Initially, each class obtains its own memory space by requesting free *slabs*, 1MB each, from the allocator. Each allocated slab is divided into *slots* of equal size. According to the slot size, the slabs are categorized into different classes, from Class 1 to Class n, where the slot size increases exponentially. A newly incoming item is accepted into a class whose slot size is the best fit of the item size. If there is no free space in the class, a currently cached item has to be first *evicted* from the class of slabs following the LRU policy. In this design, the number of slabs in each class represents the memory space that has been allocated to it.

As memory is much more expensive than external storage devices, the system operators need to maximize the efficiency of memory cache. They need to know how much cache space should be deployed to meet the service-level-agreements (SLAs). Default Memcached fills the cache at the cold start based on the demand. We observe that this demand-driven slab allocation does not deliver optimal performance , which will be explained in Section 2.1. Performance prediction [5, 6] and optimization [7, 8, 9, 10, 11] for Memcached have drawn much attention recently. Some studies focus on profiling and modelling the performance under different cache capacities [6]. In the presence of workload changing, default Memcached server may suffer from a problem called *slab calcification* [12], in which the allocation cannot be adjusted to fit the change of access pattern as the old slab allocation may not work well for the new workload. To avoid the performance drop, the operator needs to restart the server to reset the system. Recent studies have proposed adaptive slab allocation strategies and shown a notable improvement over the default allocation [13, 14, 15]. We will analyze several state-of-the-art solutions in Section 2. We find that these approaches are still far behind a theoretical optimum as they do not

exploit the locality inherent in the Memcached requests.

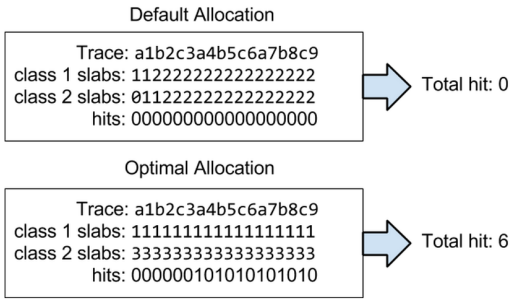
We propose a novel, dynamic slab allocation scheme, *locality-aware memory allocation* (LAMA), based on a recent advance on measurement of data locality [16] described in Section 2.2. This study provides a low-overhead yet accurate method to model data locality and generate miss ratio curves (MRCs). Miss ratio curve (MRC) reveals relationship between cache sizes and cache miss ratios. With MRCs for all classes, the overall Memcached performance can be modelled in terms of different class space allocations, and it can be optimized by adjusting individual classes' allocation. We have developed a prototype system based on Memcached-1.4.20 with the locality-aware allocation of memory space (LAMA). The experimental results show LAMA can achieve over 98% of the theoretical potential.

2 Background

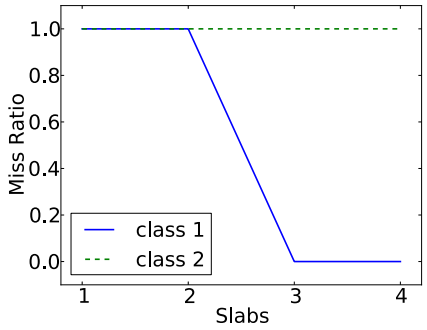
This section summarizes the Memcached's allocation design and its recent optimizations, which we will compare against LAMA, and a locality theory, which we will use in LAMA.

2.1 Memory Allocation in Memcached

Default Design In most cases, Memcached is demand filled. The default slab allocation is based on the number of items arriving in different classes during the cold start period. However, we note that in real world workloads, a small portion of the items appears in most of the requests. For example, in the Facebook ETC workload [17], 50% of the items occur in only 1% of all requests. It is likely that a large portion of real world workloads have similar data locality. The naive allocation of Memcached may lead to low cache utilization due to negligence of data locality in its design. Figure 1 shows an example to illustrate the issue of a naive allocation. Let us assume that there are two classes of slabs to receive a sequence of requests. In the example, the sequence of items for writing into Class 1 is "*abcabcabc...*", and the sequence into Class 2 is "*123456789...*". We also assume that each slab holds only one item in both classes for the sake of simplicity, and there are a total of four slabs. If the access rates of the two classes are the same, the combined access pattern would be "*a1b2c3a4b5c6a7b8c9...*". In the default allocation, every class will obtain two slabs (items) because they both store two objects during the cold start period. Note that the reuse distance of any request is larger than two for both classes. The number of hits under naive allocation would be 0. As the working set size of Class 1 is 3, the hit ratio of Class 1 will be 100% with an allocation of 3 slabs according to the MRC in Figure 1(b). If we reallocate one slab from Class 2 to Class 1, the working set of Class 1 can be fully cached and every reference to Class 1 will be a hit. Although the hit ratio of Class 2 is still 0%, the overall hit ratio of cache server will be 50%. This is much higher than the hit ratio of the default allocation which is 0%. This example motivates us to allocate space to the classes of slabs according to their data locality.



(a) Access detail for different allocation



(b) MRCs for Class 1&2

Figure 1: Drawbacks of default allocation

Automove The open-source community has implemented an automatic memory reassignment algorithm (Automove) in a recent version of Memcached [18]. In every 10 seconds window, the Memcached server counts the number of evictions in each class. If a class takes the highest number of evictions in three consecutive monitoring windows, a new slab is reassigned to it. The new slab is taken from the class that has no evictions in the last three monitoring stages. This policy is greedy but lazy. In real workloads, it is hard to find a class with no evictions for 30 seconds. Accordingly, the probability for a slab to be reassigned is extremely low.

Twitter Policy To tackle the slab calcification problem, Twitter's implementation of Memcached (Twemcache) [13] introduces a new eviction strategy to avoid frequently restarting the server. Every time a new item needs to be inserted but there is no free slabs or expired

ones, a random slab is selected from all allocated slabs and reassigned to the class that fits the new item. This random eviction strategy aims to balance the eviction rates among all classes to prevent performance degradation due to workload change. The operator no longer needs to worry about reconfiguring the cache server when calcification happens. However, random eviction is aggressive since frequent slab evictions can cause performance fluctuations, as observed in our experiments in Section 4. In addition, a randomly chosen slab may contain data that would have been future hits. The random reallocation apparently does not consider the locality.

Periodic Slab Allocation (PSA) Carra et al. [14] address some disadvantages of Twemcache and Automove by proposing *periodic slab allocation* (PSA). At any time window, the number of requests of Class i is denoted as R_i and the number of slabs allocated to it is denoted as S_i. The risk of moving one slab away from Class i is denoted as R_i/S_i. Every M misses, PSA moves one slab from the class with the lowest risk to the class with the largest number of misses. PSA has an advantage over Twemcache and Automove by picking the most promising candidate classes to reassign slabs. It aims to find a slab whose reassignment to another class dose not result in more misses. Compared with Twemcache's random selection strategy, PSA chooses the lowest risk class to minimize the penalty. However, PSA has a critical drawback: classes with the highest miss rates can also be the ones with the lowest risks. In this case, slab reassignment will only occur between these classes. Other classes will stay untouched and unoptimized since there is no chance to adjust slab allocation among them. Figure 2 illustrates a simple example where PSA can get stuck. Assume that a cache server consists of three slabs and every slab contains only one item. The global access trace is "$(aa1aa2baa1aa2aa1ba2)^*$", which is composed of Class 1 "*121212...*" and Class 2 "$(aaaabaaaaaaba)^*$". If Class 1 has taken only one slab (item) and Class 2 has taken two items, Class 1 would have the highest miss rate and the lowest risk. The system will be in a state with no slab reassignment. The overall system hit ratio under this allocation will be 68%. However, if a slab (item) were to be reassigned from Class 2 to Class 1, the hit ratio will increase to 79% since the working set size of Class 1 is 2. Apart from this weak point, in our experiments, PSA shows good adaptability for slab calcification since it can react quikly to workload changing. However, since the PSA algorithm lacks a global perspective for slab assignment, the performance still falls short when compared with our locality-aware scheme.

Facebook Policy Facebook's optimization of Memcached [15] uses adaptive slab allocation strategy to balance item age. In their design, if a class is currently evicting items, and the next item to be evicted was used at least 20% more recently than the average least recently used item of all other classes, this class is identified as needing more memory. The slab holding the overall least recently used item will be reassigned to the needy class. This algorithm balances the age of the least recently used items among all classes. Effectively, the policy approximates the global LRU policy, which is inherently weaker than optimal as shown by Brock et al. using the footprint theory we will describe next [19].

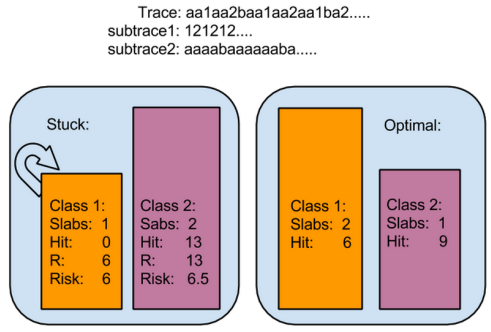


Figure 2: Drawbacks of PSA

The policies of default Memcached, Twemcache, Automove, and PSA all aim to equalize the eviction rate among size classes. The Facebook policy aims to equalize the age of the oldest item in size classes. We call the former performance balancing and the latter age balancing. Later in the evaluation section, we will compare these policies and show their relative strengths and weaknesses.

2.2 The Footprint Theory

The locality theory is by Xiang et al., who define a metric called *footprint* and propose a linear time algorithm to measure it [16] and a formula to convert it into the miss ratio [20]. Next we give the definition of footprint and show its use in predicting the miss ratio.

The purpose of the footprint is to quantify the locality in a period of program execution. An execution trace is a sequence of memory accesses, each of which is represented by a memory address. Accesses can be tagged with logical or physical time. The logical time counts the number of accesses from the start of the trace. The physical time counts the elapsed time. An execution window is a sub-sequence of consecutive accesses in the trace.

The locality of an execution window is measured by the working-set size , which is the amount of data accessed by all its accesses [21]. The footprint is a function $fp(w)$ as the average working-set size for all windows of

the same length w. While different window may have different working-set size, $fp(w)$ is unique. It is the expected working-set size for a randomly selected window.

Consider a trace `abcca`. Each element is a window of length $w = 1$. The working-set size is always 1, so $fp(1) = 5/5 = 1$. There are 4 windows of length $w = 2$. Their working-set sizes are 2, 2, 1, and 2. The average, i.e., the footprint, is $fp(2) = 7/4$. For greater window lengths, we have $fp(3) = 7/3$ and $fp(w) = 3$ for $w = 4, 5$, where 5 is the largest window length, i.e., the length of the trace. We also define $fp(0) = 0$.

Although the footprint theory is proposed to model locality of data accesses of a program, the same theory can be applied in modeling the locality of Memcached requests where data access addresses are replaced by the keys. The linear time footprint analysis leads to linear time MRC construction and thus a low-cost slab allocation prediction, as discussed next.

3 Locality-aware Memory Allocation

This section describes the design details of LAMA.

3.1 Locality-based Caching

Memcached allocates the memory at the granularity of a slab, which is 1MB in the default configuration. The slabs are partitioned among size classes.

For every size class, Memcached allocates its items in its collection of slabs. The items are ordered in a priority list based on their last access time, forming an LRU chain. The head item of the chain has the most recent access, and the tail item the least recent access. When all the allocated slabs are filled, eviction will happen when a new item is accessed, i.e. a cache miss. When the tail item is evicted, its memory is used to store the new item, and the new item is re-inserted at the first position to become the new head.

In a web-service application, some portion of items may be frequently requested. Because of their frequent access, the hot items will reside near the top of the LRU chain and hence be given higher priority to cache. A class' capacity, however, is important, since hot items can still be evicted if the amount of allocated memory is not large enough.

A slab may be reassigned from one size class to another. The *SlabReassign* routine in Memcached releases a slab used in a size class and gives it to another size class. The reassignment routine evicts all the items that are stored in the slab and removes these items from the LRU chain. The slab is now unoccupied and changes hands to store items for the new size class.

Memcached may serve multiple applications at the same time. The memory is shared. Since requests are pooled, the LRU chain gives the priority of all items based on the aggregate access from all programs.

3.2 MRC Profiling

We split the global access trace into different sub-traces according to their classes. With the sub-trace of each class, we generate the MRCs as follows. We use a hash table to record the last access time of each item. With this hash table, we can easily compute the reuse time distribution r_t, which represents the number of accesses with a reuse time t. For access trace of length n, if the number of unique data is m, the average number of items accessed in a time window of size w can be calculated using Xiang's formula [16]:

$$\begin{aligned} fp(w) &= m - \frac{1}{n-w+1}\Big(\sum_{i=1}^{m}(f_i - w)I(f_i > w) \\ &\quad + \sum_{i=1}^{m}(l_i - w)I(l_i > w) \\ &\quad + \sum_{t=w+1}^{n-1}(t-w)r_t\Big) \end{aligned} \tag{1}$$

The symbols are defined as:

- f_i: the first access time of the i-th datum
- l_i: the *reverse* last access time of the i-th datum. If the last access is at position x, $l_i = n + 1 - x$, that is, the first access time in the reverse trace.
- $I(p)$: the predicate function equals to 1 if p is true; otherwise 0.
- r_t: the the number of accesses with a reuse time t.

Now we can profile the MRC using fp distribution. The miss ratio for cache size of x is the fraction of reuses that have an average footprint smaller than x:

$$MRC(x) = 1 - \frac{\sum_{\{t|fp(t)<x\}} r_t}{n} \tag{2}$$

3.3 Target Performance

We consider two types of target performance: the total miss ratio and the average response time.

If Class i has taken S_i slabs, and I_i represents the number of items per slab in Class i. Then there should be $S_i * I_i$ items in this class. The miss ratio of this class

should be $MR_i = MRC_i(S_i * I_i)$. Let the number of requests of Class i be R_i. The total miss ratio is calculated as:

$$Miss\ Ratio = \frac{\sum_{i=1}^{n} R_i * MR_i}{\sum_{i=1}^{n} R_i} = \frac{\sum_{i=1}^{n} R_i * MRC_i(S_i * I_i)}{\sum_{i=1}^{n} R_i} \quad (3)$$

Let the average request hit time for Class i be $T_h(i)$, and the average request miss time (including retrieving data from database and setting back to Memcached) be $T_m(i)$. The average request time ART_i of Class i now can be presented as:

$$ART_i = MR_i * T_m(i) + (1 - MR_i) * T_h(i) \quad (4)$$

The overall ART of the Memcached server is:

$$ART = \frac{\sum_{i=1}^{n} R_i(ART_i)}{\sum_{i=1}^{n} R_i} \quad (5)$$

We target the overall performance by all size classes rather than equal performance in each class. The metrics take into account the relative total demands for different size classes. If we consider a typical request as the one that has the same proportional usage, then the optimal performance overall implies the optimal performance for a typical request.

3.4 Optimal Memory Repartitioning

When a Memcached server is started, the available memory is allocated by demand. Once the memory is fully allocated, we have a partition among all size classes. LAMA periodically measures the MRCs and repartitions the memory.

The optimization problem is as follows. Given the MRC for each size class, how to divide the memory among all size classes so that the target performance is maximized, i.e., the total miss ratio or the average response time is minimized?

The repartitioning algorithm has two steps:

Step 1: Cost Calculation First we split the access trace into sub-traces based on their classes. For each sub-trace $T[i]$ of Class i, we use the procedure described in Section 3.2 to calculate the miss ratio $M[i][j]$ when allocated j slabs, $0 \le j \le MAX$, where MAX is the total number of slabs. We compute the cost for different optimization targets.

To minimize total misses, $Cost[i][j]$ is the number of misses for Class i given its allocation j as follows:

$Cost[i][j] \leftarrow M[i][j] * length(T[i])$.

To minimize ART, $Cost[i][j]$ is the average access time of Class i as follows:

$$Cost[i][j] \leftarrow (M[i][j] * T_m[i] + (1 - M[i][j]) * T_h[i]) * length(T[i])$$

Algorithm 1 Locality-aware Memory Allocation

```
Input: Cost[][] // cost function, could be OPT_MISS or OPT_ART
Input: S_old[] // number of slabs in each class
Input: MAX // total number of slabs
 1: function SLABREPARTITION(Cost[][], S_old[], MAX)
 2:     F[][] ← +∞
 3:     ▷ F[][] minimal cost for Class 1..i using j slabs
 4:     for i ← 1..n do
 5:         for j ← 1..MAX do
 6:             for k ← 0..j do
 7:                 Temp ← F[i−1][j−k] + Cost[i][k]
 8:                                 ▷ Give k slabs to Class i.
 9:                 if Temp < F[i][j] then
10:                     F[i][j] ← Temp
11:                     B[i][j] ← k
12:                       ▷ B[][] saves the slab allocation.
13:                 end if
14:             end for
15:         end for
16:     end for
17:     Temp ← MAX
18:     for i ← n..1 do
19:         S_new[i] ← B[i][Temp]
20:         Temp ← Temp − B[i][Temp]
21:     end for
22:     MR_old ← 0
23:     MR_new ← 0
24:     for i ← n..1 do
25:         MR_old ← MR_old + Cost[i][S_old[i]]
26:         MR_new ← MR_new + Cost[i][S_new[i]]
27:     end for
28:     if MR_old − MR_new > threshold then
29:         SlabReassign(S_old[], S_new[])
30:     end if
31: end function
```

Step 2: Repartitioning We design a dynamic programming algorithm to find new memory partitioning (Algorithm 1). Lines 4 to 16 show a triple nested loop. The outermost loop iterates the set of size classes i from 1 to n. The middle loop iterates the number of slabs j from 1 to MAX. The target function, $F[i][j]$, stores the optimal cost of allocating j slabs to i size classes. The innermost loop iterates the allocation for the latest size class to find this optimal value.

Once the new allocation is determined, it is compared with the previous allocation to see if the performance improvement is above a certain threshold. If it is, slabs are reassigned to change the allocation. Through this pro-

cedure, LAMA reorganizes multiple slabs across all size classes. The dynamic programming algorithm is similar to Brock et al. [19] but for a different purpose.

The time complexity of the optimization is $O(n * MAX^2)$, where n is the number of size classes and MAX is the total number of slabs.

In order to avoid the cost of reassigning too many slabs, we set N slabs as the upper bound on the total reassignment. At each repartitioning, we choose N slabs with the lowest risk. We use the risk definition of PSA, which is the ratio between reference rate and number of slabs for each class. The re-allocation is global, since multiple candidate slabs are selected from possibly many size classes. In contrast, PSA selects a single candidate from one size class.

The bound N is the maximal number of slab reassignments. In the steady state, the repartitioning algorithm may decide that the current allocation is the best possible and does not reassign any slab. The number of actual reassignments can be 0 or any number not exceeding N.

Algorithm 1 optimizes the overall performance. The solution may not be fair, i.e., different miss ratios across size classes. Fairness is not a concern at the level of memory allocation. Facebook solves the problem at a higher level by running a dedicated Memcached server for critical applications [17]. If fairness is a concern, Algorithm 1 can use a revised cost function to discard unfair solutions and optimize both for performance and fairness. A recent solution is the baseline optimization by Brock et al. [19] and Ye et al. [22].

3.5 Performance Prediction

We can also predict the performance of the default Memcached. Using Equation1 in Section 3.2, we can obtain the average footprint of any window size. For a stable access pattern, we define the request ratio of Class i as q_i. Let the number of requests during the cold start period be M. The allocation for Class i by the default Memcached is the number of items it requests during this period . We predict this allocation as $fp_i(M * q_i)$. The length M of the cold-start period, i.e., the period during which the memory is completely allocated, satisfies the following equation:

$$\sum_{i=1}^{n} fp_i(M * q_i) = C \quad (6)$$

Once we get the expected items (slabs) each class can take, the system performance can be predicted by Equation 3. By predicting M and the memory allocation for each class, we can predict the performance of default Memcached for all memory sizes. The predicted allocation is similar to the natural partition of CPU cache memory, as studied in [19]. Using the footprint theory, our approach delivers high accuracy and low overhead. This is important for a system operator to determine how many caches should be deployed to achieve required Quality of Service (QoS).

4 Evaluation

In this section, we evaluate LAMA in detail, including describing the experimental setup for evaluation and comprehensive evaluation results and analysis.

4.1 Experimental setup

LAMA Implementation We have implemented LAMA in Memcached-1.4.20. The implementation includes MRC analysis and slab reassignment. The MRC analysis is performed by a separate thread. Each analysis samples recent 20 million requests which are stored using a circular buffer. The buffer is shared by all Memcached threads and protected by a mutex lock for atomic access. During the analysis, it uses a hash table to record the last access time. The cost depends on the size of the items being analyzed. It is 3% - 4% of all memory depending for the workload we use. Slab reassignment is performed by dynamic programming as shown in Section 3.4. Its overhead is negligible, both in time and in space.

System Setup To evaluate LAMA and other strategies, we use a single node, Intel(R) Core(TM) I7-3770 with 4 cores, 3.4GHz, 8MB shared LLC with 16GB memory. The operating system is Fedora 18 with Linux-3.8.2. We set 4 server threads to test the system with memory capacity from 128MB to 1024MB. The small amount of memory is a result of the available workloads we could find (in previous papers as described next). In real use, the memory demand can easily exceed the memory capacity of modern systems. For example, one of our workloads imitates the Facebook setup that uses hundreds of nodes with over 64GB memory per node [17].

We measure both the miss ratio and the response time, as defined in Section 3.4. In order to measure the latter, we set up a database as the backing store to the Memcached server. The response time is the wall-clock time used for each client request by the server, including the cost of the database access. Memcached is running on local ports and the database is running from another server on the local network.

Workloads Three workloads are used for different aspects of the evaluation:

- **The *Facebook ETC* workload to test the steady-state performance.** It is generated using Mutilate [23], which emulates the characteristics of the

ETC workload at Facebook. ETC is the closest workload to a general-purpose one, with the highest miss ratio in all Facebook's Memcached pools. It is reported that the installation at Facebook uses hundreds of nodes in one cluster [17]. We set the workload to have 50 million requests to 7 million data objects.

- **A *3-phase* workload to test dynamic allocation.** It is constructed based on Carra et al. [14]. It has 200 million requests to data items in two working sets, each of which has 7 million items. The first phase only accesses the first set following a generalized Pareto distribution with location $\theta = 0$, scale $\phi = 214.476$ and shape $k = 0.348238$, based on the numbers reported by Atikoglu et al. [17]. The third phase only accesses the second set following the Pareto distribution $\theta = 0$, $\phi = 312.6175$ and $k = 0.05$. The middle, transition phase increasingly accesses data objects from the second set.

- **A *stress-test* workload to measure the overhead.** We use the Memaslap generator of libmemcached [24], which is designed to test the throughput of a given number of server threads. Our setup follows Saemundsson et al. [6]: 20 million records with 16 byte keys and 32 byte values, and random requests generated by 10 threads. The proportion of GET requests to SET is 9:1, and 100 GETs are stacked in a single MULTI-GET request.

4.2 Facebook ETC Performance

We test and compare LAMA with the policies of default Memcached, Automove, PSA, Facebook, and Twitter's Twemcache (described in Section 2). In our experiments, Automove finds no chance of slab reassignment, so it has the same performance as Memcached. LAMA has two variants: LAMA_OPT_MR, which tries to minimize the miss ratio; and LAMA_OPT_ART, which tries to minimize the average response time. Figures 3 and 4 show the miss ratio and ART over time from the cold-start to steady-state performance. The total memory is 512MB.

The default Memcached and PSA are designed to balance the miss ratio among size classes. LAMA tries to minimize the total miss ratio. Performance optimization by LAMA shows a large advantage over performance balancing by Memcached and PSA. If we compare the steady-state miss ratio, LAMA_OPT_MR is 47.20% and 18.08% lower than Memcached and PSA. If we compare the steady-state ART, LAMA_OPT_ART is 33.45% and 13.17% lower.

There is a warm-up time before reaching the steady state. LAMA repartitions at around every 300 seconds and reassigns up to 50 slabs. We run PSA at 50 times the LAMA frequency, since PSA reassigns 1 slab each time. LAMA, PSA and Memcached converge to the steady state at the same speed. Our implementation of optimal allocation (Section 4.6) shows that this speed is the fastest.

The Facebook method differs from others in that it seeks to equalize the age of the oldest items in each size class. In the steady state, it performs closest to LAMA, 5.4% higher than LAMA_OPT_MR in the miss ratio and 6.7% higher than LAMA_OPT_ART in the average response time. The greater weakness, however, is the speed of convergence, which is about 4 times slower than LAMA and the other methods.

Twemcache uses random rather than LRU replacement. In this test, the performance does not stabilize as well as the other methods, and it is generally worse than the other methods. Random replacement can avoid slab calcification, which we consider in Section 4.5.

Next we compare the steady-state performance for memory sizes from 128MB to 1024MB in 64MB increments. Figures 5 and 6 show that the two LAMA solutions are consistently the best at all memory sizes. The margin narrows in the average response time when the memory size is large. Compared with Memcached, LAMA reduces the average miss ratio by 41.9% (22.4%–46.6%) for the same cache size, while PSA and Facebook reduce the miss ratio by 31.7%(9.1%–43.9%) and 37.6%(21.0%–47.1%). For the same or lower miss ratio, LAMA saves 40.8% (22.7%–66.4%) memory space, PSA and Facebook save 29.7%(14.6%–46.4%) and 36.9%(15.4%–55.4%) respectively.

Heuristic solutions show strength in specific cases. Facebook improves significantly over PSA for smaller memory sizes (in the steadstate). With 832MB and larger memory, PSA catches up and slightly outperforms Facebook. At 1024MB, Memcached has a slightly faster ART than both PSA and Facebook. The strength of optimization is universal. LAMA maintains a clear lead against all other methods at all memory sizes.

Compared to previous methods on different memory sizes, LAMA converges among the fastest and reaches a greater steady-state performance. The steady-state graphs also show the theoretical upper bound performance (TUB), which we discuss in Section 4.6.

4.3 MRC Accuracy

To be optimal, LAMA must have the accurate MRC. We compare the LAMA MRC, obtained by sampling and footprint version, with the actual MRC, obtained by measuring the full-trace reuse distance. We first show the MRC in individual size classes of Facebook ETC workload. There are 32 size classes. The MRCs differ in most cases. Figure 7 shows three MRCs to demonstrate. The

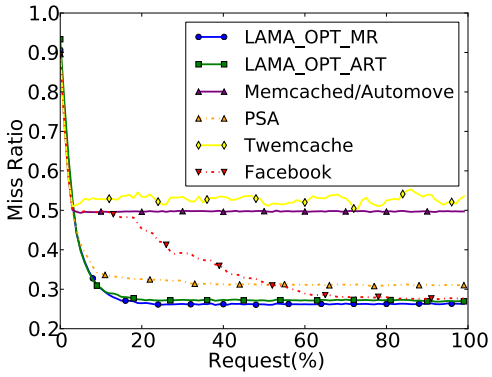


Figure 3: Facebook ETC miss ratio from cold-start to steady state

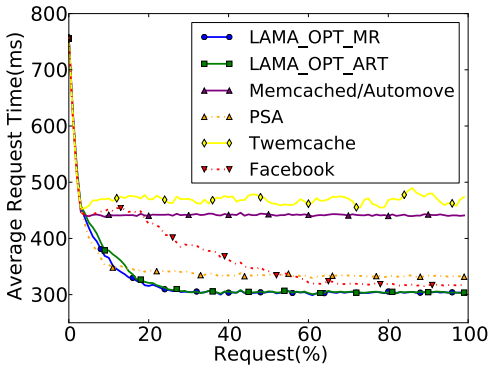


Figure 4: Average response time from cold-start to steady state

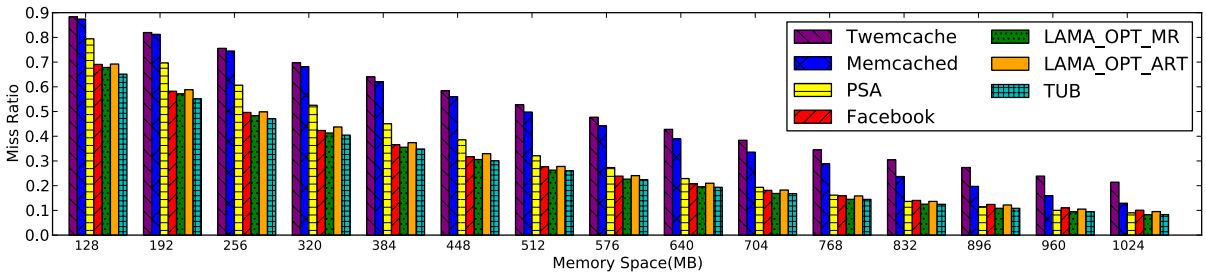


Figure 5: Steady-state miss ratio with different memory sizes

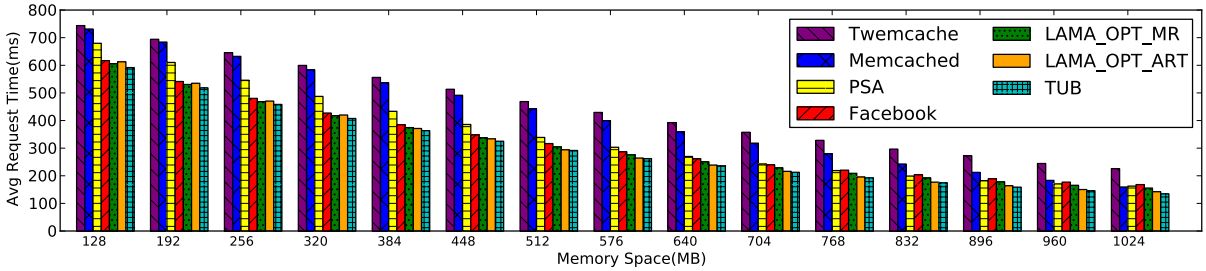


Figure 6: Steady-state average response time when using different amounts of memory

three curves have different shapes and positions in the plots, which means that data locality differs in different size classes. The shape of the middle curve is not entirely convex, which means that the traditional greedy solution, i.e. Stone et al. [25] in Section 5, cannot always optimize, and the dynamic-programming method in this work is necessary.

Figure 7 shows that the prediction is identical to the actual miss ratio for these size classes. The same accuracy is seen in all size classes. Table 1 shows the overall miss ratio of default Memcached for memory sizes from 128MB to 1024MB and compares between the prediction and the actual. The steady-state allocation prediction for default Memcached uses Equation 6 in Section 3.5. The prediction miss ratio uses Equation 4 based on predicted allocation. The actual miss ratio is measured from each run. The overall miss ratio drops as the memory size grows. The average accuracy in our test is 99.0%. The high MRC accuracy enables the effective optimization that we have observed in the last section.

4.4 LAMA Parameters

LAMA has two main parameters as explained in Section 3.4: the repartitioning interval M, which is the number of items accesses before repartitioning; and the reassignment upper bound N, which is the maximal number of slabs reassigned at repartitioning. We have tested different values of M and N to study their effects. In this section, we show the performance of running the Face-

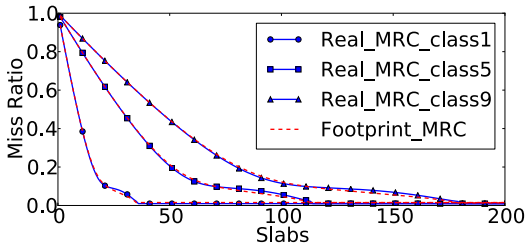


Figure 7: MRCs for class 1&5&9

Table 1: prediction miss ratio vs. real miss ratio

Capacity	Real	Prediction	Accuracy
128MB	87.56%	88.21%	99.26%
256MB	74.68%	75.40%	99.05%
384MB	62.34%	62.63%	99.54%
512MB	50.34%	50.83%	99.04%
640MB	39.36%	39.52%	99.60%
768MB	29.04%	29.27%	99.21%
896MB	20.18%	20.61%	97.91%
1024MB	13.36%	13.46%	99.26%

book ETC workload with 512MB memory.

Figure 8 shows the dynamic miss ratio over the time. In all cases, the miss ratio converges to a steady state. Different M,N parameters affect the quality and speed of convergence. Three values of M are shown: 1, 2, and 5 million accesses. The smallest M shows the fastest convergence and the lowest steady-state miss ratio. They are the benefits of frequent monitoring and repartitioning. Four values of N are shown: 10, 20, 50, and 512. Convergence is faster with a larger N. However, when N is large, 512 especially, the miss ratio has small spikes before it converges, caused by the increasing cost of slab reassignment. For fast and steady convergence, we choose $M = 1,000,000$ *and* $N = 50$ for LAMA.

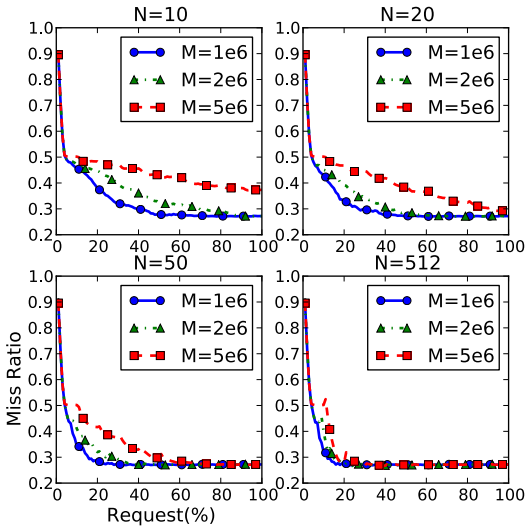


Figure 8: Different combinations of the repartitioning interval M and the reassignment upperbound N

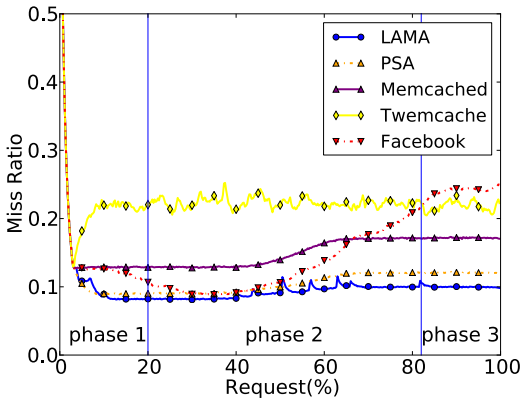


Figure 9: Miss ratio over time by different policies

4.5 Slab Calcification

LAMA does not suffer from slab calcification. Partly to compare with prior work, we use the 3-phase workload (Section 4.1) to test how LAMA adapts when the access pattern changes from one steady state to another. The workload is the same as the one used by Carra et al. [14] using 1024MB memory cache to evaluate the performance of different strategies. Figure 9 shows the miss ratio over time obtained by LAMA and other policies. The two vertical lines are phase boundaries.

LAMA has the lowest miss ratio in all three phases. In the transition Phase 2, the miss ratio has 3 small, brief increases due to the outdated slab allocation based on the previous access pattern. The allocation is quickly updated by LAMA repartitioning among all size classes. In LAMA, the slabs are "liquate" and not calcified.

Compared with LAMA, the miss ratio of the default Memcached is about 4% higher in Phase 1, and the gap increases to about 7% in Phase 3, showing the effect in Phase 3 of the calcified allocation made in Phase 1. PSA performs very well but also sees its gap with LAMA increases in Phase 3, indicating that PSA does not completely eradicate calcification. Facebook uses global LRU. Its miss ratio drops slowly, reaches the level of PSA in Phase 2, and then increases fairly rapidly. The reason is the misleading LRU information when the working set changes. The items of the first set stay a long time in the LRU chain. The random eviction by Twemcache does not favor the new working set over the previous working set. There is no calcification, but the perfor-

Table 2: Cost of MRC measurement in LAMA compared to reuse distance (RD)

Size class	Length (millions)	RD MRC (secs)	LAMA MRC (secs)	cost reduction
1	1.5953	3.6905	0.1159	96.85%
2	1.8660	4.5571	0.1378	96.97%
3	2.1091	5.2550	0.1597	96.96%
4	2.1140	5.3431	0.1598	97.00%
5	2.0646	5.2025	0.1554	97.01%
6	2.0875	5.2588	0.1585	96.98%
7	1.8725	4.6751	0.1404	96.99%
8	1.5546	3.7395	0.1131	96.97%
9	1.3022	3.0752	0.0932	96.96%

mance is significantly worse than others (except for the worst of Facebook).

4.6 Theoretical Upper Bound

To measure the theoretical upper bound (TUB), we first measure the actual MRCs by measuring the full-trace reuse distance in the first run, compute the optimal slab allocation using Algorithm 1, and re-run a workload to measure the performance. The results for Facebook ETC were shown in Figures 5 and 6. The theoretical upper bound (TUB) gives the lowest miss ratio/ART and shows the maximal potential for improvement over the default Memcached. LAMA realizes 97.6% of the potential in terms of miss ratio and 92.1% in terms of ART.

We have also tested the upper bound for the 3-phase workload. TUB shows the maximal potential for improvement over the default Memcached. In this test, LAMA realizes 99.2% of the potential in phase 3, while the next best technique, PSA, realizes 41.5%. At large memory sizes, PSA performs worse than the default Memcached. It shows the limitation of heuristic-based solutions. A heuristic may be more or less effective compared to another heuristic, depending on the context. Through optimization, LAMA matches or exceeds the performance of all heuristic solutions.

4.7 LAMA Overhead

To be optimal, LAMA depends on accurate MRCs for all size classes at the slab granularity. In our implementation, we buffer and analyze 20 million requests before each repartitioning. In Table 2, we list the overhead of MRC measurement for Facebook ETC for the first 9 size classes. MRC based on reuse distance measurement (RD MRC), takes 3 to 5.4 seconds for each size class. LAMA uses the footprint to measure MRC. The cost is between 0.09 and 0.16 second, a reduction of 97% (or equivalently, 30 times speedup). In our experiments, the repartitioning interval is about 300 seconds. The cost of LAMA MRC, 0.1 second per size class, is acceptable for online use.

We have shown that LAMA reduces the average response time. A question is whether the LAMA overhead affects some requests disproportionally. To evaluate, we measure the cumulative distribution function (CDF) for the response time of LAMA and the default Memcached. The results are shown in Figure 10. The workload is ETC workload, and the memory size is 1024MB.

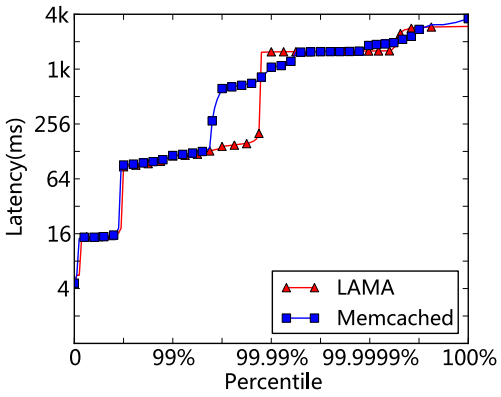


Figure 10: CDFs of request response latency

99.9% of the response times in LAMA are the same or lower than default Memcached. LAMA reduces the latency from over 512ms to less than 128ms for the next 0.09% requests. The latency is similar for the top 0.001% longest response times. The most significant LAMA overhead is the contention on the mutex lock when multiple tasks record their item access in the circular buffer. This contention and the other LAMA overheads do not cause a latency increase in the statistical distribution. LAMA's improved performance, however, reduces the latency by over 75% for 90% of the longest running requests.

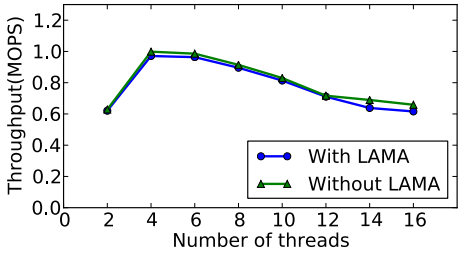


Figure 11: Throughput vs. number of threads

In this last experiment, we evaluate the throughput using stress test described in Section 4.1. The purpose is

to test the degradation when LAMA is activated. We repeat each test 10 times and report the average throughput. Figure 11 shows the overall throughput as different number of threads are used. Although the throughput of LAMA is lower than the default Memcached in the stress test, the average degradation is only 3.14%. In comparison, the Memcached performance profiler MIMIR [6], which we will introduce in Section 5, brings 8.8% degradation for its most accurate tracking. In actual use, LAMA is activated at the beginning and whenever the request pattern changes. Once LAMA produces the optimal partition, there is only the benefit and no overhead, as long as the system performance maintains stable.

5 Related Work

We have discussed related techniques on memory allocation in Section 2. Below we discuss additional related work in two other areas.

MRC Measurement Fine-grained MRC analysis is based on tracking the *reuse distance* or *LRU stack distance* [26]. Many techniques have been developed to reduce the cost of MRC profiling, including algorithmic improvement [27], hardware-supported sampling [28, 29], reuse-distance sampling [30, 31, 32], and parallel analysis [33, 34, 35]. Several techniques have used MRC analysis in online cache partitioning [36, 37, 29], page size selection [38], and memory management [39, 40]. The online techniques are not fine-grained. For example, RapidMRC has 16 cache sizes [29], and it requires special hardware for address sampling.

Given a set of cache sizes, Kim et al. divided the LRU stack to measure their miss ratios [40]. The cost is proportional to the number of cache sizes. Recently for Memcached, Bjornsson et al. developed MIMIR, which divides the LRU stack into variable sized buckets to efficiently measure the hit ratio curve (HRC) [6]. Both methods assume that items in cache have the same size, which is not the case in Memcached.

Recent work shows a faster solution using the footprint (Section 2.2), which we have extended in LAMA (Section 3.2). It can measure MRCs at per-slab granularity for all size classes with a negligible overhead (Section 4). For CPU cache MRC, the correctness of footprint-based prediction has been evaluated and validated initially for solo-use cache [16, 20]. Later validation includes optimal program symbiosis in shared cache [41] and a study on server cache performance prediction [42]. In Section 4.3, we have evaluated the prediction for Memcached size classes and shown a similar accuracy.

MRC-based Cache Partitioning The classic method in CPU cache partitioning is described by Stone et al. [25]. The method allocates cache blocks among N processes so that the miss-rate derivatives are as equal as possible. They provide a greedy solution, which allocates the next cache block to the process with the greatest miss-rate derivative. The greedy solution is of linear time complexity. However, the optimality depends on the condition that the miss-rate derivative is monotonic. In other words, the MRC must be convex. Suh et al. gave a solution which divides MRC between non-convex points [43]. Our results in Section 4.3 show that the Memcached MRC is not always convex.

LAMA is based on dynamic programming and does not depend on any assumption about MRC curve property. It can use any cost function not merely the miss ratio. We have shown the optimization of ART. Other possibilities include fairness and QoS. The LAMA optimization is a general solution for optimal memory partitioning. A similar approach has been used to partition CPU cache for performance and fairness [22, 19].

6 Conclusion

This paper has described LAMA, a locality-aware memory allocation for Memcached. The technique measures the MRC for all size classes periodically and repartitions the memory to reduce the miss ratio or the average response time. Compared with the default Memcached, LAMA reduces the miss ratio by 42% using the same amount of memory, or it achieves the same memory utilization (miss ratio) with 41% less memory. It outperforms four previous techniques in steady-state performance, the convergence speed, and the ability to adapt to phase changes. LAMA predicts MRCs with a 99% accuracy. Its solution is close to optimal, realizing 98% of the performance potential in a steady-state workload and 99% of the potential in a phase-changing workload.

7 Acknowledgements

We are grateful to Jacob Brock and the anonymous reviewers, for their valuable feedback and comments. The research is supported in part by the National Science Foundation of China (No. 61232008, 61272158, 61328201, 61472008 and 61170055); the 863 Program of China under Grant No.2012AA010905, 2015AA015305; the Research Fund for the Doctoral Program of Higher Education of China under Grant No.20110001110101; the National Science Foundation (No. CNS-1319617, CCF-1116104, CCF-0963759, CCF-0643664, CSR-1422342, CCF-0845711, CNS-1217948).

References

[1] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[2] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Accelerating mapreduce with distributed memory cache. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 472–478. IEEE, 2009.

[3] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: filling the gaps in data center memory. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 53–64. ACM, 2014.

[4] Gurmeet Singh and Puneet Chandra Rashid Tahir. A dynamic caching mechanism for hadoop using memcached.

[5] Steven Hart, Eitan Frachtenberg, and Mateusz Berezecki. Predicting memcached throughput using simulation and modeling. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, page 40. Society for Computer Simulation International, 2012.

[6] Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[7] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 36–47. ACM, 2013.

[8] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 236–243. IEEE, 2012.

[9] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, pages 371–384, 2013.

[10] Jinho Hwang and Timothy Wood. Adaptive performance-aware distributed memory caching. In *ICAC*, pages 33–43, 2013.

[11] Wei Zhang, Jinho Hwang, Timothy Wood, KK Ramakrishnan, and Howie Huang. Load balancing of heterogeneous workloads in memcached clusters. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*. USENIX Association, 2014.

[12] Caching with twemcache. https://blog.twitter.com/2012/caching-with-twemcache, 2014. [Online].

[13] Twemcache. https://twitter.com/twemcache, 2014. [Online].

[14] Damiano Carra and Pietro Michiardi. Memory partitioning in memcached: An experimental performance analysis. *Communications (ICC), 2014 IEEE International Conference on*, pages 1154–1159, 2014.

[15] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

[16] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360. IEEE, 2011.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[18] Memcached-1.4.11. https://code.google.com/p/memcached/wiki/ReleaseNotes1411, 2014. [Online].

[19] Jacob Brock, Yechen Li, Chencheng Ye, and Chen Ding. Optimal cache partition-sharing : Dont ever take a fence down until you know why it was put up. robert frost. In *Proceedings of ICPP*, 2015.

[20] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *ASPLOS*, pages 343–356, 2013.

[21] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[22] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. Recu: Rochester elastic cache utility – unequal cache sharing is good economics. In *Proceedings of NPC*, 2015.

[23] Mutilate. https://github.com/leverich/mutilate, 2014. [Online].

[24] Libmemcached. http://libmemcached.org/libMemcached.html, 2014. [Online].

[25] Harold S Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054–1068, 1992.

[26] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[27] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.

[28] J Torrellas, Evelyn Duesterwald, Peter F Sweeney, and Robert W Wisniewski. Multiple page size modeling and optimization. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 339–349. IEEE, 2005.

[29] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 121–132. ACM, 2009.

[30] Kristof Beyls and Erik H DHollander. Discovery of locality-improving refactorings by reuse path analysis. In *High Performance Computing and Communications*, pages 220–229. Springer, 2006.

[31] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.

[32] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*, pages 91–100. ACM, 2008.

[33] Huimin Cui, Qing Yi, Jingling Xue, Lei Wang, Yang Yang, and Xiaobing Feng. A highly parallel reuse distance analysis algorithm on gpus. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1080–1092. IEEE, 2012.

[34] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing*, 73(7):1011–1027, 2013.

[35] Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.

[36] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, pages 1–12. ACM, 2001.

[37] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[38] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 339–349, 2005.

[39] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 177–188. ACM, 2004.

[40] Yul H Kim, Mark D Hill, and David A Wood. *Implementing stack simulation for highly-associative memories*, volume 19. ACM, 1991.

[41] Xiaolin Wang, Yechen Li, Yingwei Luo, Xiameng Hu, Jacob Brock, Chen Ding, and Zhenlin Wang. Optimal footprint symbiosis in shared cache. In *CCGRID*, 2015.

[42] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[43] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data

Xingbo Wu[1], Yuehai Xu[1], Zili Shao[2], and Song Jiang[1]

[1] Wayne State University, {wuxb,yhxu,sjiang}@wayne.edu
[2] The Hong Kong Polytechnic University, cszlshao@comp.polyu.edu.hk

Abstract

Key-value (KV) stores have become a backbone of large-scale applications in today's data centers. The data set of the store on a single server can grow to billions of KV items or many terabytes, while individual data items are often small (with their values as small as a couple of bytes). It is a daunting task to efficiently organize such an ultra-large KV store to support fast access. Current KV storage systems have one or more of the following inadequacies: (1) very high data write amplifications, (2) large index set, and (3) dramatic degradation of read performance with overspill index out of memory.

To address the issue, we propose LSM-trie, a KV storage system that substantially reduces metadata for locating KV items, reduces write amplification by an order of magnitude, and needs only two disk accesses with each KV read even when only less than 10% of metadata (Bloom filters) can be held in memory. To this end, LSM-trie constructs a trie, or a prefix tree, that stores data in a hierarchical structure and keeps re-organizing them using a compaction method much more efficient than that adopted for LSM-tree. Our experiments show that LSM-trie can improve write and read throughput of LevelDB, a state-of-the-art KV system, by up to 20 times and up to 10 times, respectively.

1 Introduction

Key-value (KV) stores play a critical role in the assurance of service quality and user experience in many websites, including *Dynamo* [22] at Amazon, *Voldemort* [7] at LinkedIn, *Cassandra* [1] at Apache, *LevelDB* [4] at Google, and *RocksDB* [11] at Facebook. Many highly-demanding data-intensive internet applications, such as social networking, e-commerce, and online gaming, rely on quick access of data in the stores for quality service.

A KV store has its unique advantage on efficient implementation with a flat data organization and a much simplified interface using commands such as Put(key,value) for writing data, Get(key) for reading data, and Delete(key). However, there are several trends on workload characteristics that are seriously challenging today's state-of-the-art KV store implementations for high performance and high scalability.

First, very small KV items are widespread. As an example, Facebook had reported that 90% of its Memcached KV pools store KV items whose values are smaller than 500 bytes [13]. In one KV pool (USR) dedicated for storing user-account statuses all values are of 2 bytes. In its nonspecific, general-purpose pool (ETC) 2-, 3-, or 11-byte values add up to 40% of the total requests to the store. In a replicated pool for frequently accessed data, 99% of KV items are smaller than 68 bytes [26]. In the wildcard (the default pool) and a pool devoted for a specific application, 75% of items are smaller than 363 bytes. In Twitter's KV workloads, after compression each tweet has only 362 bytes, which contains only 46 bytes of text [3]. In one of Instagram's KV workloads the key is the media ID and the value is the user ID. Each KV item is just as large as a couple of bytes [10]. For a store of a given capacity, smaller KV items demand more metadata to locate them. The metadata may include index for locating a data block (e.g., a 4 KB disk block) and Bloom filters for determining data existence in the block.

Second, demand on a KV store's capacity at individual KV servers keeps increasing. The rising demand is not only due to data-intensive applications, but also because of the cost benefit of using fewer servers to host a distributed KV store. Today it is an economical choice to host a multi-terabytes KV store on one server using either hard disks or SSDs. However, this would significantly increase metadata size and make memory constrained, which is especially the case when significant applications, such as MapReduce jobs, are scheduled to the cluster hosting the store, competing the memory resource with the storage service [19, 33].

Third, many KV stores require high performance for both reads and writes. It has been reported that ratio of read and write counts in typical low-latency workloads at Yahoo had shifted from anywhere between 2 and 9 to around 1 in recent years [29]. Among the five core workloads in Yahoo's YCSB benchmark suite two of them have equal share of read and write requests [18]. There are KV stores, such as LevelDB, that are optimized for writes by organizing data in multiple levels. However, when not all metadata can be held in memory, multiple disk reads, each for medadata of a level, are needed to serve a read request, degrading read performance. In the meantime, for some KV stores, such as SILT [24], major efforts are made to optimize reads by minimizing metadata size, while write performance can be compromised without conducting multi-level incremental compactions.

In this paper, we propose LSM-trie, a KV storage system that can accommodate multi-billions of small items with a capacity of multi-terabytes at one server with limited memory demand. It supports a sustained throughput of over 500 K writes per second, and a sustained throughput of over 50 K reads per second even for workloads without any locality and thus with little help from caching[1]. To achieve this, LSM-trie uses three novel techniques. First, it integrates exponential growth pattern in the LSM tree (Log-Structured Merge-tree)—a commonly adopted KV-store organization—with a linear growth pattern. This enables a compaction design that can reduce write amplification by an order of magnitude and leads to much improved write throughput. A high write throughput is desired as data modifications and deletions are also processed as writes in the store implementation. Second, using a trie, or a prefix tree, to organize data in the store, LSM-trie almost eliminates index. This allows more and stronger Bloom filters to be held in memory, making service of read requests faster. Third, when Bloom filters become too large to be entirely held in the memory, LSM-trie ensures that on-disk Bloom filters are clustered so that in most cases only one 4 KB-block read is required to locate the data.

Experiments show that LSM-trie significantly improves write throughput over schemes in comparison, including LevelDB, RocksDB, and SILT, by up to 20 times regardless of system configurations such as memory size, store size, storage devices (SSD or HDD), and access pattern (uniform or Zipfian key distributions). LSM-trie can also substantially improve read throughput, especially when memory available for running the KV store is limited, by up to 10 times.

Note that LSM-trie uses hash functions to organize its data and accordingly does not support range search. This is a choice similarly made in the design of many important KV stores, including Amazon's Dynamo [22], LinkedIn's Voldermort [7], and SILT [24], as this command is not always required by their users. Furthermore, there are techniques available to support the command by maintaining an index above these hash-based stores with B-link tree [17] or dPi-tree [25], and experimental studies indicate that "there is no absolute winner" in terms of range-search performance between stores natively supporting it and those relying on external support [28].

[1]The throughput of read is significantly lower than that of write because one read needs access of at least one 4 KB block, while multiple small KV items in write requests can be compacted into one block.

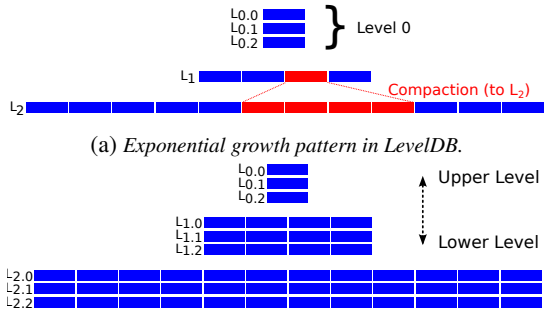
2 The design of LSM-trie

The design of LSM-trie was motivated by the excessively large write amplification of LSM-tree due to its data organization and compaction scheme [27]. In this section we will describe the issue in the context of LevelDB, a popular implementation of LSM-tree from Google. Then we will describe a trie-based LSM-tree implementation that can dramatically reduce write amplification in Section 2.3. However, this optimized LSM-tree still retains an index, which grows with the store size and eventually becomes a barrier to the system's scalability. In addition, it may require multiple reads of Bloom filters on the disk with a large store. In Section 2.4, we describe **LSM-trie**, where KV items are hashed into individual buckets, indices are accordingly removed, and Bloom filters are grouped together to support efficient access.

2.1 Write Amplification in LSM-tree

A KV store design based on LSM-tree has two goals: (1) new data must be quickly admitted into the store to support high-throughput write; and (2) KV items in the store are sorted to support fast data location. We use a representative design, LevelDB, as an example to explain the challenges on simultaneously achieving both of the goals.

To meet the first goal LevelDB writes to the disk in a large unit (a couple of megabytes) to generate an on-disk data structure called *SSTable*. Specifically, LevelDB first uses an in-memory buffer, called *MemTable*, to receive incoming KV items. When a MemTable is full, it is written to the disk to become an immutable SSTable. KV items in an SSTable are sorted according to their keys. An SSTable is stored as a file, and KV items are placed in 4 KB blocks of the file. To locate a KV item in the SSTable, LevelDB places an index in the file recording the key of the first KV item in each block. Conducting binary search on the index, LevelDB knows in which block a KV item can possibly be located. Because 4 KB block is a disk access unit, it is not necessary to maintain a larger index to determine byte offset of each item in a



(a) *Exponential growth pattern in LevelDB.*

(b) *Alternating use of linear and exponential growth patterns.*

Figure 1: *Using multi-level structure to grow an LSM-tree store. Each solid rectangle represents an SSTable.*

block. However, the index does not tell whether an item is actually in the block. If not, accessing the block is unnecessary and can substantially increase read latency. To this end, LevelDB maintains a Bloom filter for each block to indicate whether an item is in it [16]. To minimize its false positive rate, the filter must be sized proportionally to the number of items in a block, usually 10–16 bits per item.

To meet the second goal LevelDB builds a multi-level tree-like structure to progressively sort KV items. As shown in Figure 1a, new SSTables, which are just converted from MemTables, are placed in Level 0. To quickly admit incoming items, items in new SSTables are not immediately sorted with those in existing SSTables at Level 0. Instead, each of the SSTables becomes a sub-level ($L_{0.0}, L_{0.1}, L_{0.2}, \ldots$) of Level 0 (See Figure 1a). In the background, LevelDB merge-sorts a number of L_0 SSTables to produce a list of non-overlapping SSTables at Level 1 (L_1), an operation called **compaction**. To quickly have more data sorted into one list, starting from Level 1 there are no sub-levels and the ratio of two adjacent levels' sizes is large ($Size(L_{k+1})/Size(L_k)$, where $k = 0, 1, \ldots$). We name the ratio amplification factor, or AF in short, which is 10 in LevelDB by default. As every level (L_{k+1}) can be 10 times as large as its immediate upper level (L_k), the store keeps producing exponentially larger sorted list at each level and becomes very large with only a few levels.

However, this exponential growth pattern leads to an excessively large write amplification ratio, a ratio between actual write amount to the disk and the amount of data requested for writing by users. Because the range of keys covered by each level is roughly the same, to push one SSTable at a level down to its next lower level LevelDB needs to read this SSTable and ten SSTables in the lower level (in the worst case) whose entire key range matches the SSTable's key range. It then merge-sorts them and writes the 11 resulting SSTables to the lower level. That is, the write amplification ratio is 11, or $AF + 1$. For a new KV item to reach Level k ($k = 0, 1, 2, \ldots$), the write amplification ratio can go up to $k \times (AF + 1)$. When the k value reaches 5 or larger, the amplification ratio can become unacceptably large (55 or larger). Such an expensive compaction operation can consume most of the I/O bandwidth and leave little for servicing frontend user requests.

For a store of given capacity, efforts on reducing the write amplification by limiting number of levels would have counter effect. One example is the SILT KV store [24], which essentially has two levels (HashStore and SortedStore). When the store grows large, its SortedStore has to be much larger than HashStore (even when multiple HashStores are employed). This causes its very high write amplification (see Section 3 for measurements), which justifies the use of multiple levels for progressive compaction in the LSM-tree-based stores.

2.2 Challenge on Reducing Write Amplification in the LSM-tree Compaction

A compaction entails reading sorted lists (one SSTable from L_k and a number of SSTables matching its key range from L_{k+1}), merging-sorting them into one sorted list, and writing it back to L_{k+1}. While any data involved in the operation contribute to the write amplification, it is the larger data set from the lower level (L_{k+1}) that makes the amplification ratio excessively large. Because the purpose of the compaction is to push data to the lower level, the contribution to the amplification from accessing data at the upper level is necessary. If we manage to allow only data at the upper level to be involved in a compaction, the write amplification can be minimized.

To this end, we introduce the linear growth pattern. As shown in Figure 1b, in addition to Level 0 other levels also consist of a number of its sub-levels. Sub-levels belonging to the same level are of the same (maximum) size. When a new sub-level is produced at a level, the store linearly grows at this level. However, when a new level is produced, the store *exponentially* grows (by AF times). During growth of the store, new (sub)-levels are produced alternatively using the linear and exponential growth patterns. In other words, each LevelDB's level is replaced by multiple sub-levels. To minimize write amplification, we can merge-sort data in the sub-levels of a level (L_k) to produce a new sub-level of its next lower level (L_{k+1}). As similar amount of data in each sub-level, but no data in the next lower level, are involved in a compaction, write amplification can be minimized.

A key consideration in LevelDB's implementation is to bound each compaction's maximum cost in terms of number of SSTables involved, or $AF + 1$, to keep service of user requests from being disruptively slowed down by the background operation. For the same purpose, in the

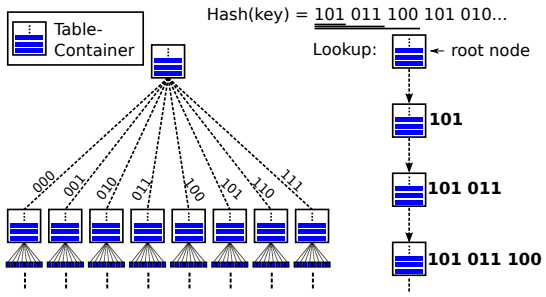


Figure 2: *A trie structure for organizing SSTables. Each node represents a table container, which contains a pile of SSTables.*

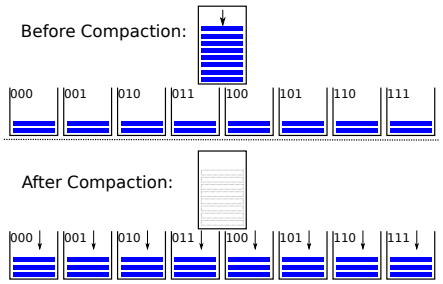


Figure 3: *A compaction operation in the trie.*

use of linear growth pattern in a compaction we select one SSTable at each sub-level of a level (L_k), and merge-sort these SSTables into a sequence of non-overlapping SSTables at Level L_{k+1}. The range of keys involved in a compaction represents the *compaction's key range*. Among all compactions moving data from L_k to L_{k+1}, we must make sure their key ranges are not overlapped to keep any two SSTables at Level L_{k+1} from having overlapped key ranges. However, this cannot be achieved with the LevelDB data organization because the sorted KV-items at each sub-level are placed into the SSTables according to the tables' fixed capacity (e.g., 32 MB). The key range size of an SSTable can be highly variable and the ranges' distribution can be different in different sub-levels. Therefore, ranges of the aforementioned compactions are unlikely to be un-overlapped.

2.3 *SSTable-trie*: A Design for Minimizing Write Amplification

To enable distinct key range in a compaction, we do not use a KV-item's ranking (or its position) in a sorted list to determine the SSTable it belongs to in a level. Instead, we first apply a cryptographic hash function, such as SHA-1, on the key, and then use the hashed key, or *hashkey* in short, to make the determination. This essentially converts the LevelDB's multi-level structure into a trie, as illustrated in Figure 2. Accordingly we name this optimized LevelDB **SSTable-trie**.

An SSTable-trie is a prefix tree whose nodes are table containers, each containing a number of SSTables. Each node has a fixed number of child nodes and the number is equivalent to the *AF* (amplification factor) in LevelDB. If the number is assumed to be 8, a node's children can be distinguished by a three-bit binary ($000, 001, \ldots,$ or 111). A node in the trie can also be identified by a binary, usually of more bits. Starting from the root node, we can segment the binary into consecutive three-bit groups with the first group indicating a root's child. As each bit group identifies a corresponding node's child, we can follow the bit groups to find a path to the node corresponding to the binary. All nodes of the same depth in a trie constitute a level in the trie structure, which is equivalent to a level in LevelDB. Each container has a pile of SSTables (see Figure 2). A trie level consists of a number of SSTable piles. All SStables at the same position of the piles at a trie level constitute a sub-level of the trie, which corresponds to a sub-level in LevelDB.

As each KV item is also identified by a binary (the hashkey), its location in a level is determined by matching the hashkey's prefix to the identity of a node in the level (see Figure 2). In contrast to the KV-item placement in a level of LevelDB, a KV-item's location in a trie level is independent of other keys in the same level. A compaction operation involves a pile of SSTables in only one container. After a compaction KV items in a pile are moved into the container's children according to their respective hashkeys, rather than their rankings in the sorted list as LevelDB does. By using hashkeys each compaction's key range is unique and SSTables produced by a compaction are non-overlapping. Such a compaction incurs minimal write amplification. Figure 3 illustrates a compaction operation in a trie. Note that use of SHA-1 as the hash function to generate hashkey guarantees a uniform distribution of KV items at each (sub)-level regardless of distribution of original keys.

2.4 *LSM-trie*: a Large Store for Small Items

Our goal is to enable very large KV stores in terms of both capacity and KV-item count in a server. A big challenge on designing such a store is the management of its metadata that often have to be out of core (the DRAM).

2.4.1 Out-of-Core Metadata

For a given KV item, there is at most one SSTable at each (sub)-level that may store the item in LevelDB because every (sub)-level is sorted and its SSTables' key ranges are not overlapped. The store maintains a very small in-memory search tree to identify the SSTable at each level. At the end of each SSTable file an index and Bloom filters are stored to facilitate search in the table. The index

is employed to identify a 4 KB block and a Bloom filter is maintained for each block to tell whether a KV item is possibly in the block. The indices and Bloom filters in a KV store can grow very large. Specifically, the size of the indices is proportional to the store's capacity (or number of 4 KB blocks), and the size of the Bloom filters is proportional to total item count. For a large store the metadata can hardly be accommodated in memory. For example, a 10 TB store holding 200 B-KV-items would require about 125 GB space for 10-bit-per-key Bloom-filters and 30 GB for indices. While it is well affordable now and even so in the near future to have an HDD array or even an SSD array as large as 10 TB in a server, it is not cost-effective to dedicate such a large DRAM only for the metadata. Therefore, we have to assume that significant portion of the metadata is only on the disk when the store grows large. Because locality is usually not assumed in KV-store workloads [14, 31], the fact can be that most reads require retrieval of metadata from the disk before data can be read. The critical issue is how to minimize number of metadata reads in serving a read request for a KV item. These metadata are possibly stored in multiple SSTables, each at a different level. As the metadata are associated with individual SSTables and are distributed over them, having multiple reads seems to be unavoidable in the current LSM-tree's structure.

SSTable-trie introduces the linear growth pattern, which leads to the design of **LSM-trie** that removes almost all indices and enables one metadata disk access per read request. Before describing the design, let us first address a concern with SSTable-trie. Using the linear growth pattern one can substantially increase number of levels. As a multi-level KV-item organization requires continuous search of levels, starting from Level 0, for a requested item until it is found, it relies on Bloom filters in each level to skip as many levels without the item as possible. However, as each Bloom filter has a false positive rate (about 0.82% for a setting of 10 bits per item), the probability of searching levels without the item increases with the increase of level count (e.g., from 5.7% for a 7-level structure to 46% for a 56-level one). Therefore, the Bloom filter must be beefed up by using more bits. For example, using a setting of 16 bits per item would ensure less than 5% false positive rate for an entire 120-level structure. Compared with the disk capacity, the additional on-disk space for the larger Bloom filters is minimal. As we will show, LSM-trie removes indices and uses only one disk access to read Bloom filters.

2.4.2 Removing Indices by Using HTables

LSM-trie represents an improvement over SSTable-trie by incorporating an efficient metadata management. A major change is to replace the SSTable in SSTable-trie

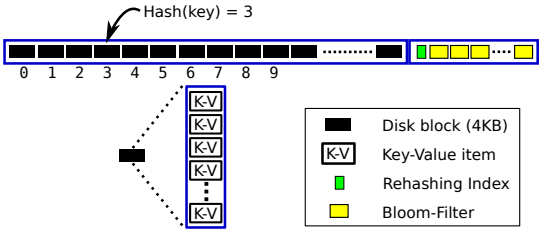
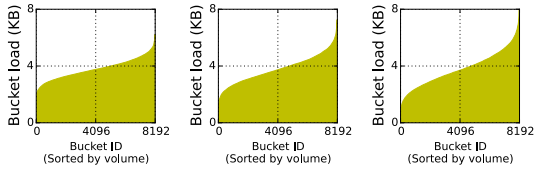


Figure 4: *The structure of an HTable.*



(a) *100 B* (b) *200 B* (c) *300 B*

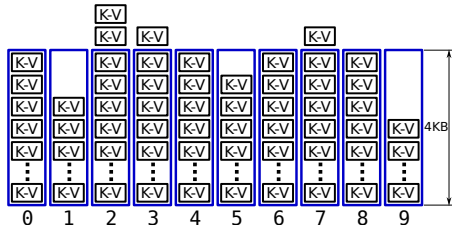
Figure 5: *Distribution of bucket load across buckets of an HTable with a uniform distribution of KV-item size and an average size of 100 B (a), 200 B (b), and 300 B (c). The keys follow the Zipfian distribution. For each plot, the buckets are sorted according to their loads in terms of aggregate size of KV items in a bucket.*

with **HTable**, a hash-based KV-item organization (see Figure 4). In an SSTable, items are sorted and index is needed for locating a block. In HTable, each block is considered as a bucket for receiving KV items whose keys are hashed into it. While each KV item has a SHA-1-generated 160 bit hashkey and its prefix has been used to identify an SSTable in SSTable-trie, or an HTable in LSM-trie, we use its suffix to determine a bucket within an HTable for the KV item. Specifically, if there are m buckets in an HTable, a KV item with Hashkey h would be placed in Bucket $(h \mod m)$.

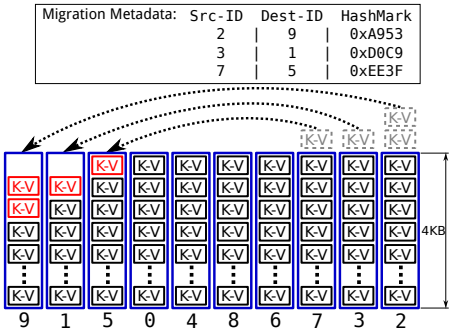
To eliminate the index in an HTable, LSM-trie must use buckets of fixed size. Further, as Bloom filter is applied on individual buckets, an entire bucket would be read should its filter indicate a possible existence of a lookup item in the bucket. Therefore, for access efficiency buckets should be of the same size as disk blocks (4 KB). However, a challenging issue is whether the buckets can be load balanced in terms of *aggregate size of KV items hashed into them*. It is known that using a cryptographic hash function allows each bucket to have statistically equal chance to receive a new item, and item count in each bucket follows a normal distribution. In addition to key's distribution, item size[2] and variation of item size also add to variation of the bucket load.

Figure 5 shows the distribution of bucket load across the buckets in an HTable after we store KV items, whose keys are of the Zipfian distribution, into a 32 MB HTable of 8192 4 KB-buckets. For each plot, the item size is of

[2]With larger KV items it is harder to balance the load across the buckets in an HTable.



(a) *KV items are assigned to the buckets by the hash function, causing unbalanced load distribution.*



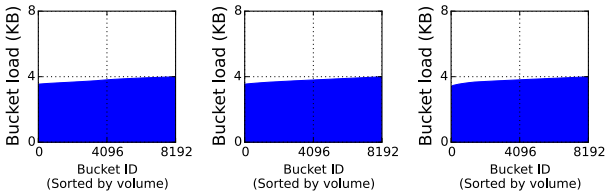
(b) *Buckets are sorted according to their loads and balanced by using a greedy algorithm.*

Figure 6: *Balancing the load across buckets in an HTable.*

the uniform distribution with different average sizes, and the size is in the range from 1 B to a size about doubling their respective averages. In each experiment we keep writing KV items to the store until it is 95% full. By using the highly-skewed Zipfian distribution, the results represent a conservative estimation of non-uniformity of bucket load distribution.[3] As shown, there are increasingly more over-loaded buckets and more under-loaded buckets with the increase of average item size.

Obviously LSM-trie must move excessive items out of over-loaded buckets to make sure every bucket has 4 KB or less data. Like SSTable, HTable is also immutable. During the construction of an HTable, we use a greedy algorithm to migrate some items that were originally hashed to an overloaded bucket to an under-loaded bucket for storage. As illustrated in Figure 6, the buckets are first sorted into a list according to their initial loads. We then conduct a paired migration operation within the list, in which a minimal number of KV items are moved out of the most overloaded bucket (the source) to the most under-loaded bucket (the destination) until the remaining items in the source can fit in the bucket. The source bucket is removed from the list and we keep the list sorted. We then repeat the migration operation on the shorter list. The operation continues until either a list's source bucket is not overloaded or the list's destination bucket is also overloaded. To minimize the chance

[3]Interestingly the results are little affected by the key distribution. Even the uniform key distribution produces similar results.



(a) *100 B* (b) *200 B* (c) *300 B*

Figure 7: *Bucket load distribution after load balancing for HTables with different average item sizes.*

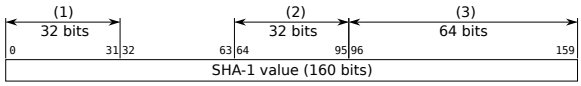


Figure 8: *Uses of a 160-bit SHA1 key: (1) the prefix is used for trie encoding; (2) The infix is used for sorting KV items in a bucket; and (3) the suffix is used for locating the KV items in an HTable.*

of having the second scenario, we set a limit on the aggregate size of KV items that can be stored in an HTable, which is 95% of the fixed HTable capacity (32 MB by default). This approach is effective. For example, with such a small reduction on usable capacity we have not observed a single item that is moved out of an over-loaded bucket but cannot be accommodated in an under-loaded bucket for HTables whose item sizes are 400 B on average and are uniformly distributed between 1 B and 800 B. Figure 7 shows the bucket load distribution after the load is balanced.

To handle the case of overflown items that cannot be accepted into any regular buckets, mostly due to excessively large KV items, during creation of a new HTable, LSM-trie sets up a special bucket to receive them. Items in the special bucket are fully indexed. The index is saved in the HTable file and is also cached in memory for efficiently locating the items. As the bucket is designed only for a few large KV items, its index should be of minimal size. Generally, workloads for accessing consistently large items (a few KBs or larger) should use SSTable-trie. In fact, such workloads do not pose a challenge on their metadata management in most KV stores.

There are several issues to address on the load balancing strategy. One is how to efficiently identify KV items overflown out of a bucket. To minimize the bookkeeping cost for the purpose, we use a hash function on the keys to rank KV items in a bucket and logically place them into the bucket according to their rankings. We then use the bucket capacity (4 KB) as the watermark. Any items that are across or above the watermark are considered as overflown items for migration. We only need to record the hash value for the item at the watermark, named **HashMark**, for future lookups to know whether an item has been migrated. For the hash function, we simply select a 32-bit infix in the 160-bit hashkey (e.g., from 64th bit to 95th bit), as illustrated in Figure 8. We

also record where the items are migrated (the destination bucket ID). A migrated item can be further migrated and searching for the item would need to walk over multiple buckets. To minimize the chance for an item to be repeatedly migrated, we tune the hash function by rotating the 32-bit infix by a particular number of bits, where the number is a function of bucket ID. In this way, different functions can be applied on different buckets, and an item is less likely to keep staying above buckets' watermarks for repeated migrations.

The metadata for each bucket about its overflown items comprise a source bucket ID (2 B), a migration destination ID (2 B), and a HashMark (4 B). They are stored in the bucket on the disk. A design issue is whether to cache the metadata in memory. If we cache every bucket's metadata, the cost would be comparable to the indices in SSTable, which records one key for each block (bucket). Actually it is not necessary to record all buckets' metadata if we do not require exactly one bucket read in an HTable lookup. As shown in Figure 5, distribution of overflown items over the buckets is highly skewed. So we only need to cache metadata for the most overloaded buckets (20% by default) and make lookup of these items be re-directed to their respective destination buckets without a disk read. In this way, with slightly increased disk reads LSM-trie can significantly reduce its cached metadata. For example, when KV items are of 100 B in average and their sizes are uniformly distributed between 1 B and 200 B, only 1.01 bucket reads per lookup are needed with only 14 KB (1792×8 B) of the metadata cached, about 1/10 of the size of an SSTable's indices.

Similar to LevelDB, LSM-trie maintains a Bloom filter for each bucket to quickly determine whether a KV item could be there. The migration of KV items out of a bucket does not require updating the bucket's Bloom filter, as these KV items still logically remain in the bucket and are only physically stored in other bucket(s). Their physical locations are later revealed through the bucket's migration-related metadata.

2.4.3 Clustering Bloom Filters for Efficient Access

LSM-trie does not assume that all Bloom filters can always be cached in memory. A Bloom filter at each (sub)-level needs to be inspected until a requested item is found. LSM-trie makes sure that all Bloom filters that are required to service a read request in a level but are not cached can be retrieved into memory with only one disk read. To this end LSM-trie gathers all Bloom filters associated with a column of buckets[4] at different sub-levels of an HTable container into a single disk block named

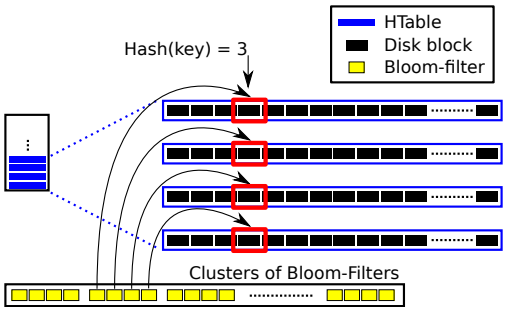


Figure 9: *Clustering Bloom filters*

bits/key	50 Levels	100 Levels	150 Levels
10	40.95%	81.90%	122.85%
12	15.70%	31.40%	47.10%
14	6.00%	12.00%	18.00%
16	2.30%	4.59%	6.89%
18	0.88%	1.76%	2.64%

Table 1: *Bloom filter false-positive rate.*

BloomCluster, as illustrated in Figure 9. Because the same hash function is applied across the sub-levels, a KV item can appear only in one particular column of buckets if it is in the container. In this way, only one disk read of Bloom filters is needed for a level.

While LSM-trie is designed to support up to a 10 TB store, its data is organized so that at most one read of metadata (Bloom filters) is required to access any item in the store. The prototyped LSM-trie system uses 32 MB HTables and an amplification factor (AF) of 8. The store has five levels. In the first four levels, LSM-trie uses both linear and exponential growth pattern. That is, each level consists of eight sub-levels.[5] All the Bloom filters for the first 32 sub-levels are of 4.5 GB, assuming a 64 B average item size and 16 bit Bloom filter per key. Adding metadata about item migration within individual HTables (up to 0.5 GB), LSM-trie needs up to only 5 GB memory to hold all necessary metadata. At the fifth level, which is the last level, LSM-trie uses only linear growth pattern. As one sub-level of this level has a capacity of 128 G, it needs 8 such sub-levels for the store to reach 1 TB, and 80 such sub-levels to reach 10 TB. All the sub-levels' Bloom filters are well clustered into a BloomCluster so that only one disk read of Bloom filter is required for a read request. Though the false positive rate increases with level count, it can be well capped by using additional bits per KV item, as shown in Table 1. When LSM-trie uses 16-bit-per-item Bloom filters, the false positive rate is only about 5% even for a 112-sub-level 10 TB KV store. In the worse case there are only 2.05 disk reads, one for a BloomCluster and 1.05 on average for data.

[4] As shown in Figure 9, the column of buckets refers to all buckets at the same position of respective HTables in a container.

[5] Actual number of sub-levels in a level can change during compaction operations. It varies between 0 and 16 with an average of 8.

	SSD	HDD
Random Read 4 KB (IOPS)	52,400	70
Sequential Write (MB/s)	230	144
Sequential Read (MB/s)	298	138

Table 2: *Basic disk performance measurements.*

In the LSM-trie structure, multiple KV items of the same key, including special items for Delete operations, can simultaneously stay in different sub-levels of the last level without being merged as there are no merge-sort operations at this level. Among the items of the same key, only the item at the highest sub-level is alive and the others are considered as garbage. This may lead to under-utilized disk space, especially when the level contains substantial amount of garbage. To ameliorate the effect, we periodically sample a few random HTable containers and assess their average garbage ratio. When the ratio is larger than a threshold, we schedule garbage-collection operations in a container-by-container manner either periodically or when the system is not loaded.

3 Performance Evaluation

To evaluate LSM-trie's performance, we implement a prototype and extensively conduct experiments to reveal insights of its performance behaviors.

3.1 Experiment Setup

The experiments are run on a Dell CS23-SH server with two Intel Xeon L5410 4-core processors, 64 GB FB-DIMM memory, and 64-bit Linux 3.14. The SSD (Samsung 840 EVO, MZ-7TE1T0BW) has 1 TB capacity. Because of its limited storage capacity (1 TB), we install DRAM of moderate size on the computer (64 GB), a configuration equivalent to 256 GB memory with a 4 TB store. We also build a KV store on a hard disk, which is 3 TB Seagate Barracuda (ST3000DM001) with 64 MB cache and 7200 RPM. Table 2 lists the disks' performance measurements. As we can see, the hard disk's random read throughput is too small and it's not competitive considering SSD's rapidly dropping price. Therefore, we do not run read benchmarks on the hard disk. All experiments are run on the SSD(s) unless stated otherwise. In LSM-trie immediately after a table is written to the disk, we issue `fsync()` to persist its data.

In the evaluation, we compare LSM-trie with LevelDB [4], RocksDB (an optimized LevelDB from Facebook) [11], and SILT [24]. LSM-trie uses 32 MB HTables, LevelDB and RocksDB use 32 MB SSTables, and SILT uses 32 MB HashStore. We run SILT using its source code provided by its authors with its default setup [9]. We do not include experiments for SSTable-trie as its write performance is the same as LSM-trie, but its read performance can be unacceptably worse than that of LevelDB when there are many levels and Bloom filters cannot be cached.

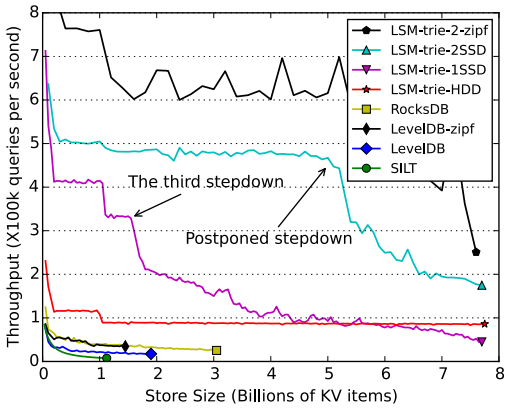


Figure 10: *Write throughput of different stores. For each store, the execution stops when either the store reaches 1TB or the run time reaches 24 hours, whichever occurs earlier.*

We use Yahoo's YCSB benchmark suite to generate read and write requests [18]. Average value size of the KV items is 100 B with a uniform distribution between 1 B to 200 B. The key size is 16 B. We use constant value size (100 B) for SILT as it does not support varied value size. By default, we use the uniform key distribution, as it represents the least locality and minimal overwrites in the workload, which helps increase a store's write pressure.[6]

3.2 Experiment Results

In this section we present and analyze experiment results for write and read requests.

3.2.1 Write Throughput

Figure 10 plots the write throughput, in terms of number of PUT queries served per second (QPS), for LSM-trie, LevelDB, RocksDB, and SILT with different store sizes, or numbers of KV items in the store. We have a number of interesting observations on the plots.

The LSM-trie store has throughput way higher than other stores. Even the throughput for LSM-trie on the hard disk (see the "LSM-trie-HDD" curve) more than doubles those of other stores on the SSD. It takes about 24 hours for LSM-trie to build a 1 TB store containing nearly 8 billions of small items on an HDD. As it is too slow for the other stores to reach the size of 1 TB within a reasonable time period, we stop their executions after they run for 24 hours. By estimation it would take RocksDB and LevelDB about 4–6 days and even longer time for SILT to build such a large store on the SSD.

[6] We do have a test for the Zipfian distribution in Section 3.2.

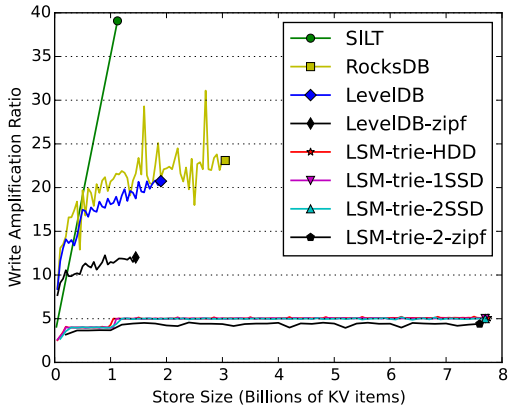


Figure 11: *Write amplification ratios of different stores. For each store, the execution stops when either the store reaches 1TB or the run time reaches 24 hours, whichever occurs earlier.*

Admittedly SILT is designed mainly to service read requests [24]. However, taking so long to build a large store is less desirable in the first place. To understand their big performance gaps, we draw the write amplification ratio (WAR) plots for the stores in Figure 11.

It's not a surprise to see SLIT's WAR increases almost linearly with the store size, as SILT does not adopt a multi-level organization. By maintaining a large SortedStore and merge-sorting much smaller HashStores into it, most of its compaction I/O is to access data in the SortedStore, and contributes to the WAR. While both LevelDB and RocksDB adopt LSM-tree's multi-level organization, its exponential growth pattern significantly compromises its WAR. The WAR curve of RocksDB is obtained by running its performance monitoring tool (`db_bench`). The curve exhibits large variations, mainly because of its choice of sampling points for performance measurements. While RocksDB generally has a higher WAR, its write throughput is higher than that of LevelDB because of its use of multiple threads to better utilize parallelism available in SSD and CPU. The WAR curves for LSM-trie ("LSM-trie-*" curves in Figure 11) have small jumps at about 0.12 and 1.0 billion items in the KV store, corresponding to the timings when the store grows into Levels 3 and 4, respectively (Figure 11). Once the store reaches its last level (Level 4), the WAR curves become flat at around 5 while the store increases up to 10 TB.

The write throughput curve for the hard disk ("LSM-trie-HDD") in Figure 10 has two step-downs, well matching the two jumps in its corresponding WAR curve. After the store reaches 1 billion items, its throughput does not reduce with the increase of the store. For LSM-trie on the SSD, we do see the first and second step-downs on the curve ("LSM-trie-1SSD" in Figure 10) corresponding to the two WAR jumps. However, we had been confused by the third step-down, as marked in Figure 10, when the store size reaches about 1.7 billion items or 210 GB. One might attribute this throughput loss to the garbage collection. However, we had made efforts to use large HTables (32 MB) and aligned them to the erase block boundaries. After investigation, it turns to be due to SSD's internal *static* wear-leveling.

As we know, frequency of data re-writing at different levels dramatically varies. The ratio of the frequencies between two adjacent levels (lower level vs. upper level) can be as high as 8. For data at Level 4 and at Level 0, the ratio of their re-write frequencies could be 4096 (8^4)! With such a large gap between the frequencies, dynamical wear-leveling is insufficient and SSD's FTL (Flash Translation Layer) has to proactively move data at the lower level(s) around to even out flash wear across the disk. The impact of the wear-levering becomes increasingly serious when more and more SSD's space is occupied. To confirm our speculation, we introduce a second SSD and move data at the two upper level (about only 2.5 GB) to it, and run LSM-trie on the two SSDs (see "LSM-trie-2SSD" in Figure 10). The third step-down is postponed to a significantly later time (from about 1.7 billion items to about 5.2 billion items). The new third step-down is caused by re-write frequency gaps among data at Levels 2, 3, and 4 in the first SSD. Using more SSDs and separating them onto different SSDs would eliminate the step-down. In practice, it is a viable solution to have a few small but wear-resistent SSDs (e.g., SLC SSD) to separate the first several levels of data.

We also issue write requests with the Zipfian key distribution to LSM-trie on two SSDs. It has a smaller WAR than those with the uniform key distribution (see "LSM-trie-2-zipf" in Figure 11), and higher throughput (see "LSM-trie-2-zipf" in Figure 10). Strong locality of the workload produces substantial overwrites, which are merged during the compactions. As a result, about one third of items are removed before they reach the last level, reducing write amplification and increasing throughput. The Zipfian distribution also allows LevelDB to significantly reduce its WAR (compare "LevelDB" and "LevelDB-zipf" in Figure 11) and to increase its write throughput (compare "LevelDB" and "LevelDB-zipf" in Figure 10).

In almost all scenarios, LSM-trie dramatically improves WAR, leading to significantly increased write throughput. The major reason of the improvements is the introduction of the linear growth pattern into the LSM tree and the adoption of the trie structure to enable it.

3.2.2 Performance of Read

Figures 12 and 13 plot the read throughput for various stores on one SSD with 64Gb and 4GB memory, respectively, except SILT. Keys of read requests are uniformly distributed. As explained, we cannot build a sufficiently

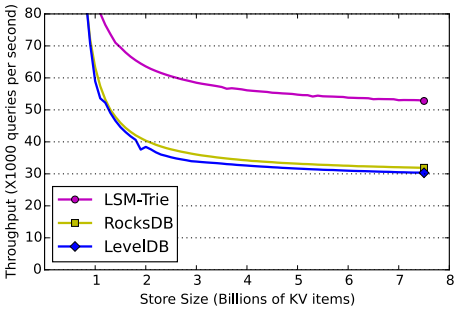


Figure 12: *Read throughput with 64 GB memory.*

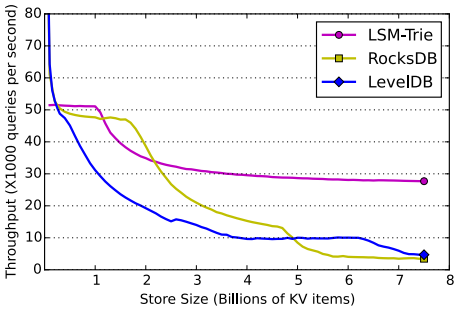


Figure 13: *Read throughput with 4 GB memory.*

large SILT store to measure its read performance. Instead, we will use the results reported in its paper for comparison [24]. To accelerate the building of the LevelDB and RocksDB stores, we use YCSB to generate a trace of write requests whose keys are sorted. The stores can then be quickly built without any compactions.

As shown in Figure 12, when the store size is relatively small (with fewer than about 1 billion KV items or 128 GB data), almost half of accessed data can be cached in memory and the throughput is very high (much higher than 80K QPS). This throughput is not explicitly shown in the figure, as it is less I/O related. LSM-trie has higher throughputs than LevelDB and RocksDB for both small and large store sizes. With a small store size, LSM-trie uses less memory to cache metadata and leaves more for caching data than other stores, producing higher hit ratios and read throughputs. When the store becomes larger, theh working set becomes larger due to uniform key distribution and the memory size becomes less relevant to the throughput. LSM-trie's higher throughputs with larger store are due to the alignment of its block to the SSD pages in its implementation. Without the alignment, one access of an SSTable-file's block may result in access of an additional page. For the following experiment we augment LevelDB and RocksDB by aligning their blocks to the SSD pages. LSM-trie's throughput with a large store (over 6 billions KV items) is around 96% of one SSD's raw read throughput in terms of number of 4 KB-blocks read per second. This is the same percentage reported in the SILT paper [24].

Considering the scenario where a server running a KV store may simultaneously run other application(s) demanding substantial memory resource, or where a KV store runs within a disk drive with small memory [8], we evaluate LSM-trie's performance with a constrained memory size. Figure 13 shows read throughput when the memory is only 4 GB [7]. Current LSM-trie's implementation always keeps metadata for the first four levels in the memory. More and more requests require one read of out-of-core metadata in addition to one read of data after the store grows beyond the first four levels. This is why the curve for LSM-trie starts to drop beyond 1.2-billion-item store size. The throughput curves of LevelDB and RocksDB also drop with the increase of store size. They drop much more than that of LSM-trie. RocksDB's throughput is higher than that of LevelDB initially, as it caches more metadata by giving metadata a caching priority higher than data.

Latency Percentile	5% read	50% read	95% read
95%	690 μs	790 μs	700 μs
99%	860 μs	940 μs	830 μs

Table 3: *Read Latency under mixed read/write workload.*

Our measurements show that all requests can be completed in 1 ms, and its 99% percentile latency is 0.92 ms. To know how read latency is affected by concurrent write requests, we list the 95% and 99% percentile latencies for different percentages of read requests among all the read/write requests in Table 3. The read latencies are not sensitive to write intensity. The KV store store many small items in write requests into one block while each read request has to retrieve an entire block. Thanks to the much reduced write compaction in LSM-trie, intensity of write requests has a small impact on read latency.

4 Related Work

Key-value stores have become an increasingly popular data management system with its sustained high performance with workloads challenging other systems, such as those generating a huge number of small data items. Most related works aim for efficient writes and reads.

4.1 Efforts on Supporting Efficient Writes

Most KV stores support fast writes/updates by using log-based write, such as FAWN [12], FlashStore [20], SkimpyStash [21], SILT [24], LevelDB [4], and bLSM [29]. Though log-appending is efficient for admitting new data, it is not sufficient for high write efficiency. There can be significant writes caused by internal data re-organization and their efficiency can be critical to the write throughput observed by users. A primary objective of the re-organization is to remove garbage from

[7] Note that write performance is not affected by the small memory.

the log. Some systems, such as FAWN, FlashStore, and SkimpyStash, focus mostly on this objective and incurs a relatively small number of additional writes. Though these systems are efficient for serving writes, they leave the data not well organized, and produce a large metadata set leading to slow reads with relatively small memory.

Another group of systems, such as LevelDB, SILT, and bLSM, aim to build a fully organized data structure—one (almost) sorted list of KV items. This is apparently ideal for reducing metadata size and facilitating fast reads. It is also essential for a scalable system. However, it can generate a very large write amplification. The issue quickly deteriorates with the growth of the store. To address the issue, RocksDB compacts more than two contiguous levels at once intending to sort and push data faster to the lower level [11]. However, the improvement is limited as the amplification is fundamentally due to the difference of the data set sizes at different levels. To mitigate the compaction cost, TokuDB uses a Fractal Tree, in which data is pushed to its next level by simply being appended into log files at corresponding tree nodes [23, 15]. Without well sorting its data, TokuDB has to maintain a much larger index, leading to larger memory demand and/or additional disk access for metadata. In contrast, with the support of the trie structure and use of linear growth pattern, LSM-trie minimizes write amplification.

4.2 Efforts on Supporting Efficient Reads

Read efficiency is mostly determined by two factors. One is metadata size and the other is the efficiency of retrieving metadata from the disk. Both determine how many disk reads are needed to locate a requested KV item.

As SILT has a fully sorted list of KV items and uses a highly compact index representation, it produces very small metadata [24]. In contrast, LevelDB's metadata can be much larger as they include both indices and Bloom filters. It may take multiple reads for LevelDB to load its out-of-memory metadata. FAWN [12] and FlashStore [20] have very large metadata as they directly store pointers to the on-disk items, especially when the items are small and the store is large. SkimpyStash stores hash table buckets on the disk, essentially leaving most metadata on the disk and may require many disk reads of metadata to locate the data [21]. In contrast, LSM-trie substantially reduces metadata by removing almost all indices. It requires at most one metadata read for each read request with its well clustered metadata.

4.3 Other Related Works

Sharding (or partitioning), as a technique to distribute heavy system load such as large working sets and intensive I/O requests across nodes in a cluster, has been widely used in database systems and KV stores [6, 5, 2]. It has been proposed as a potential method for reducing merge (or compaction) overhead by maintaining multiple smaller store instances (shards) at a node [24]. However, if the number of shards is moderate (fewer than one hundred) at a node, each shard has to grow into four or larger number of levels when the store becomes large. Accordingly write amplification cannot be substantially reduced. Meanwhile, because memory demand, including MemTables and metadata, is about proportional to the number of shards, using many shards increase pressure on memory. In contrast, LSM-trie fundamentally addresses the issue by improving store growth pattern to minimize compaction cost without concerns of sharding.

Being aware of large compaction cost in LevelDB, VT-Tree opportunistically looks for any block at a level whose key range does not overlap with that of blocks at another level during merge-sorting of the two levels' KV items [30]. Effectiveness of this method relies on probability of having non-overlapping blocks. For workloads with small items, there are a large number of keys in a block, reducing the probability. Though it had been reported that this method can reduce write amplification by about $\frac{1}{3}$ to $\frac{2}{3}$, it is far from enough. In contrast, LSM-trie reduces the amplification by up to an order of magnitude.

While LSM-trie trades some disk space (around 5%) for much improved performance, Yu et al. proposed a method to improve performance of the disk array by trading capacity for performance [32]. They trade 50% of the disk space for a throughput improvement of 160%.

5 Conclusions

In this paper we describe LSM-trie, a key-value store designed to manage a very large data set in terms of both its data volume and KV item count. By introducing linear growth pattern, LSM-trie minimizes compaction cost for LSM-tree-based KV systems. As our extensive experiments demonstrate, LSM-trie can manage billions of KV items with a write amplification of only five. By design it can manage a store of up to 10 TB. LSM-trie can service a read request with only two SSD reads even when over 90% of the bloom-filters is not in the memory. Furthermore, with a second small SSD (only 20 GB) to store the bloom-filters, the overall throughput can reach the peak throughput of the raw device (50 K QPS vs. 52 K IOPS), and 99% of its read latency is below 1 ms.

6 Acknowledgments

This work was supported by US National Science Foundation under CAREER CCF 0845711 and CNS 1217948.

References

[1] Apache cassandra. http://cassandra.apache.org.

[2] Consider the apache cassandra database. http://goo.gl/tX37h3.

[3] How much text versus metadata is in a tweet? http://goo.gl/EBFIFs.

[4] Leveldb: A fast and lightweight key/value database library by google. https://code.google.com/p/leveldb/.

[5] mongodb. http://goo.gl/sQdYfo.

[6] Mysql cluster: Scalability. http://goo.gl/SIfvfe.

[7] Project voldermort: A distributed key-value storage system. http://project-voldemort.com.

[8] Seagate kinetic HDD. http://goo.gl/pS9bs1.

[9] Silt: A memory-efficient, high-performance key-value store. https://github.com/silt/silt.

[10] Storing hundreds of millions of simple key-value pairs in redis. http://goo.gl/ieeU17.

[11] Under the hood: Building and open-sourcing rocksdb. http://goo.gl/9xulVB.

[12] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. *Commun. ACM 54*, 7 (July 2011), 101–109.

[13] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[14] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[15] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2007), SPAA '07, ACM, pp. 81–92.

[16] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.

[17] BRANTNER, M., FLORESCU, D., GRAF, D., KOSSMANN, D., AND KRASKA, T. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 251–264.

[18] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[19] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[20] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1414–1425.

[21] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 25–36.

[22] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[23] KUSZMAUL, B. C. How fractal trees work. http://goo.gl/PG3kr4.

[24] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 1–13.

[25] LOMET, D. Replicated indexes for distributed data. In *In PDIS* (1996), pp. 108–119.

[26] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[27] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Inf. 33*, 4 (June 1996), 351–385.

[28] PIRZADEH, P., TATEMURA, J., AND HACIGÜMÜS, H. Performance evaluation of range queries in key value stores. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings* (2011), pp. 1092–1101.

[29] SEARS, R., AND RAMAKRISHNAN, R. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 217–228.

[30] SHETTY, P., SPILLANE, R., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2013), FAST'13, USENIX Association, pp. 17–30.

[31] VO, H. T., WANG, S., AGRAWAL, D., CHEN, G., AND OOI, B. C. Logbase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow. 5*, 10 (June 2012), 1004–1015.

[32] YU, X., GUM, B., CHEN, Y., WANG, R. Y., LI, K., KRISHNAMURTHY, A., AND ANDERSON, T. E. Trading capacity for performance in a disk array. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000), USENIX Association, pp. 17–17.

[33] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.

MetaSync: File Synchronization Across Multiple Untrusted Storage Services

Seungyeop Han, Haichen Shen, Taesoo Kim[†], Arvind Krishnamurthy, Thomas Anderson, and David Wetherall
University of Washington, [†]Georgia Institute of Technology

Abstract

Cloud-based file synchronization services, such as Dropbox, are a worldwide resource for many millions of users. However, individual services often have tight resource limits, suffer from temporary outages or even shutdowns, and sometimes silently corrupt or leak user data.

We design, implement, and evaluate MetaSync, a secure and reliable file synchronization service that uses multiple cloud synchronization services as untrusted storage providers. To make MetaSync work correctly, we devise a novel variant of Paxos that provides efficient and consistent updates on top of the unmodified APIs exported by existing services. Our system automatically redistributes files upon reconfiguration of providers.

Our evaluation shows that MetaSync provides low update latency and high update throughput while being more trustworthy and available. MetaSync outperforms its underlying cloud services by 1.2-10× on three realistic workloads.

1 Introduction

Cloud-based file synchronization services have become tremendously popular. Dropbox reached 300M users in May 2014, adding 100M customers in six months [15]. Many competing providers offer similar services, including Google Drive, Microsoft OneDrive, Box, and Baidu. These services provide very convenient tools for users, especially given the increasing diversity of user devices needing synchronization. With such resources and tools, mostly available for free, users are likely to upload ever larger amounts of personal and private data.

Unfortunately, not all services are trustworthy or reliable in terms of security and availability. Storage services routinely lose data due to internal faults [6] or bugs [13, 23, 30], leak users' personal data [12, 31], and alter user files by adding metadata [7]. They may block access to content (e.g., DMCA takedowns [38]). From time to time, entire cloud services may go out of business (e.g., Ubuntu One [9]).

Our work is based on the premise that users want file synchronization and the storage that existing cloud providers offer, but without the exposure to fragile, unreliable, or insecure services. In fact, there is no fundamental need for users to trust cloud providers, and given the above incidents our position is that users are best served by *not* trusting them. Clearly, a user may encrypt files before storing them in the cloud for confidentiality. More generally, Depot [27] and SUNDR [26] showed how to design systems from scratch in which users of the cloud storage obtain data confidentiality, integrity, and availability without trusting the underlying storage provider. However, these designs rely on fundamental changes to both client and server; our question was whether we could use existing services for these same ends?

Instead of starting from scratch, MetaSync provides file synchronization on top of multiple existing storage providers. We thus leverage resources that are mostly well-provisioned, normally reliable, and inexpensive. While each service provides unique features, their common purpose is to synchronize a set of files between personal devices and the cloud. By combining multiple providers, MetaSync provides users larger storage capacity, but more importantly a more highly available, trustworthy, and higher performance service.

The key challenge is to maintain a globally consistent view of the synchronized files across multiple clients, using only the service providers' unmodified APIs without any centralized server. We assume no direct client-client or server-server communication. To this end, we devise two novel methods: 1) pPaxos, an efficient client-based Paxos algorithm that maintains globally consistent state among multiple passive storage backends (§3.3), and 2) a stable deterministic replication algorithm that requires minimal reshuffling of replicated objects on service re-configuration, such as increasing capacity or even adding/removing a service (§3.4).

Putting it all together, MetaSync can serve users better in all aspects as a file synchronization service; users need trust only the software that runs on their own computers. Our prototype implementation of MetaSync, a ready-to-use open source project, currently works with five different file synchronization services, and it can be easily extended to work with other services.

2 Goals and Assumptions

The usage model of MetaSync matches that of existing file synchronization services such as Dropbox. A user configures MetaSync with account information for the underlying storage services, sets up one or more directories to be managed by the system, and shares each directory with zero or more other users. Users can connect these directories with multiple devices (we refer to the devices and software running on them as `clients` in this paper), and local updates are reflected to all connected

clients; conflicting updates are flagged for manual resolution. This usage model is supported by a background synchronization daemon (MetaSyncd in Figure 1).

For users desiring explicit control over the merge process, we also provide a manual git-like push/pull interface with a command line client. In this case, the user creates a set of updates and runs a script to apply the set. These sets of updates are atomic with respect to concurrent updates by other clients. The system accepts an update only if it has been merged with the latest version pushed by any client.

Our baseline design assumes the backend services to be curious, as well as potentially unreachable, and unreliable. The storage services may try to discover which files are stored along with their content. Some of the services may be unavailable due to network or system failures; some may accidentally corrupt or delete files. However, we assume that service failures are independent, services implement their own APIs correctly (except for losing and corrupting user data), and communications between client and server machines are protected. We also consider extensions to this baseline model where the services have faulty implementations of their APIs or are actively malicious (§3.6). Finally, we assume that clients sharing a specific directory are trusted, similar to a shared Dropbox directory today.

With this threat model, the goals of MetaSync are:

- **No direct client-client communication:** Clients coordinate through the synchronization services without any direct communication among clients. In particular, they never need to be online at the same time.
- **Availability:** User files are always available for both read and update despite any predefined number of service outages and even if a provider completely stops allowing any access to its previously stored data.
- **Confidentiality:** Neither user data nor the file hierarchy is revealed to any of the storage services. Users may opt out of confidentiality for better performance.
- **Integrity:** The system detects and corrects any corruption of file data by a cloud service, to a configurable level of resilience.
- **Capacity and Performance:** The system should benefit from the combined capacity of the underlying services, while providing faster synchronization and cloning than any individual service.

3 System Design

This section describes the design of MetaSync as illustrated by Figure 1. MetaSync is a distributed, synchronization system that provides a reliable, globally consistent storage abstraction to multiple clients, by using untrusted cloud storage services. The core library defines a generic cloud service API; all components are implemented on top of that abstraction. This makes it

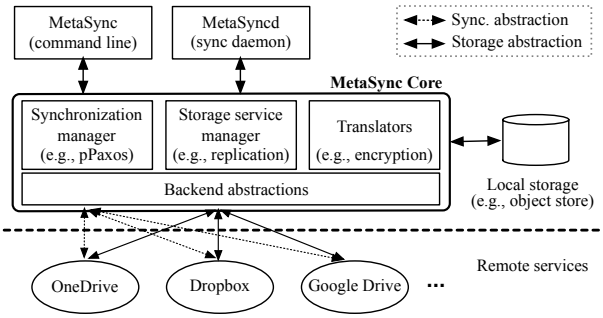


Figure 1: MetaSync has three main components: a storage service manager to coordinate replication; a synchronization manager to orchestrate cloud services; and translators to support data encryption. They are implemented on top of an abstract cloud storage API, which provides a uniform interface to storage backends. MetaSync supports two front-end interfaces: a command line interface and a synchronization daemon for automatic monitoring and check-in.

easy to incorporate a new storage service into our system (§3.7). MetaSync consists of three major components: synchronization manager, storage service manager, and translators. The synchronization manager ensures that every client has a consistent view of the user's synchronized files, by orchestrating storage services using pPaxos (§3.3). The storage service manager implements a deterministic, stable mapping scheme that enables the replication of file objects with minimal shared information, thus making our system resilient to reconfiguration of storage services (§3.4). The translators implement optional modules for encryption and decryption of file objects in services and for integrity checks of retrieved objects, and these modules can be transparently composed to enable flexible extensions (§3.5).

3.1 Data Management

MetaSync has a similar underlying data structure to that of git [20] in managing files and their versions: objects, units of data storage, are identified by the hash of their content to avoid redundancy. Directories form hash trees, similar to Merkle trees [29], where the root directory's hash is the root of the tree. This root hash uniquely defines a snapshot. MetaSync divides and stores each file into chunks, called Blob objects, in order to maintain and synchronize large files efficiently.

Object store. In MetaSync's object store, there are three kinds of objects—Dir, File and Blob—each uniquely identified by the hash of its content (with an object type as a prefix in Figure 2). A File object contains hash values and offsets of Blob objects. A Dir object contains hash values and names of File objects.

In addition to the object store, MetaSync maintains two kinds of metadata to provide a consistent view of the global state: *shared metadata*, which all clients can modify; and *per-client metadata*, which only the single owner (writer) client of the data can modify.

Shared metadata. MetaSync maintains a piece of shared metadata, called `master`, which is the hash value

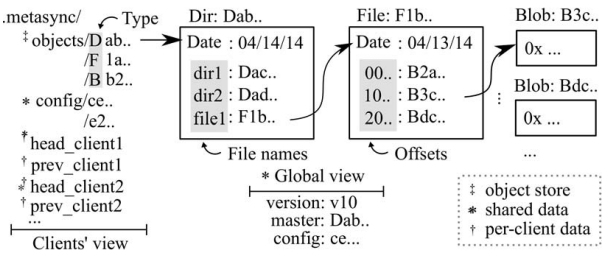


Figure 2: File management in a client's local directory. The object store maintains user files and directories with content-based addressing, in which the name of each object is based on the hash of its content. MetaSync keeps two kinds of metadata: shared, which all clients update; and per-client, for which the owner client is the only writer. The object store and per-client files can be updated without synchronization, while updates to the shared files require coordinated updates of the backend stores; this is done by the synchronization manager (§3.3).

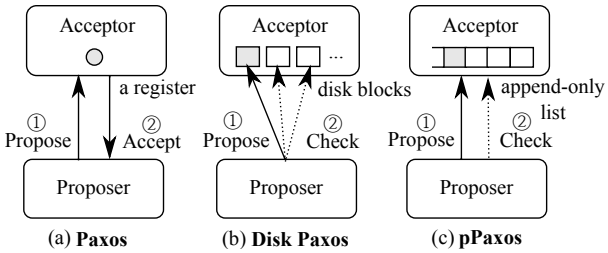


Figure 3: Comparison of operations between a proposer and an acceptor in Paxos [25], Disk Paxos [19], and pPaxos. Each acceptor in Paxos makes a local decision to accept or reject a proposal and then replies with the result. Disk Paxos assumes acceptors are passive; clients write proposals into per-client disk blocks at each acceptor. Proposers need to check every per-client block (at every acceptor) to determine if their proposal was accepted, or preempted by another concurrent proposal. With pPaxos, the append-only log allows clients to efficiently check the outcome at the passive acceptor.

of the root directory in the most advanced snapshot. It represents a consistent view of the global state; every client needs to synchronize its status against the `master`. Another shared piece of metadata is the configuration of the backend services including information regarding the list of backends, their capacities, and authenticators. When updating any of the shared metadata, we invoke a synchronization protocol built from the APIs provided by existing cloud storage providers (§3.3).

Per-client data. MetaSync keeps track of clients' states by maintaining a view of each client's status. The per-client metadata includes the last synchronized value, denoted as `prev_clientID`, and the current value representing the client's recent updates, denoted as `head_clientID`. If a client hasn't changed any files since the previous synchronization, the value of `prev_clientID` is equal to that of `head_clientID`. As this data is updated only by the corresponding client, it does not require any coordinated updates. Each client stores a copy of its per-client metadata into all backends after each update.

3.2 Overview

MetaSync's core library maintains the above data structures and exposes a reliable storage abstraction to applications. The role of the library is to mediate accesses and updates to actual files and metadata, and further interacts with the backend storage services to make file data persistent and reliable. The command line wrapper of the APIs works similarly with version control systems.

Initially, a user sets up a directory to be managed by MetaSync; files and directories under that directory will be synchronized. This is equivalent to creating a repository in typical version control systems. Then, MetaSync creates a metadata directory (`.metasync` as shown in Figure 2) and starts the synchronization of file data to backend services.

Each managed directory has a name (called namespace) in the system to be used in synchronizing with other clients. Upon initiation, MetaSync creates a folder with the name in each backend. The folder at the backend storage service stores the configuration information plus a subset of objects (§3.4). A user can have multiple directories with different configurations and composition of backends.

When files in the system are changed, an update happens as follows: (1) the client updates the local objects and `head_client` to point to the current root (§3.1); (2) stores the updated data blocks on the appropriate backend services (§3.4)); and (3) proposes its `head_client` value as the new value for `master` using pPaxos (§3.3)). The steps (1) and (2) do not require any coordination, as (1) happens locally and (2) proceeds asynchronously. Note that these steps are provided as separate functions to applications, thus each application or user can decide when to run each step; crucially, a client does not have to update global `master` for every local file write.

3.3 Consistent Update of Global View: pPaxos

The file structure described above allows MetaSync to minimize the use of synchronization operations. Each object in the object store can be independently uploaded as it uses content-based addressing. Each per-client data (e.g., `head_client_*`) is also independent since we ensure that only the owning client modifies the file. Thus, synchronization to avoid potential race conditions is necessary only when a client wants to modify shared data (i.e., `master` and configuration).

pPaxos. In a distributed environment, it is not straightforward to coordinate updates to data that can be modified by multiple clients simultaneously. To create a consistent view, clients must agree on the sequence of updates applied to the shared state, by agreeing on the next update applied at a given point.

Clients do not have communication channels between each other (e.g., they may be offline), so they need to rely on the storage services to achieve this consensus. However, these services do not communicate with each other,

nor do they implement consensus primitives. Instead, we devise a variant of Paxos [25], called pPaxos (passive Paxos) that uses the exposed APIs of these services.

We start our overview of pPaxos by relating it to the classic Paxos algorithm (see Figure 3(a)). There, each client works as a proposer and learner; the next state is determined when a majority accepts a given proposal. Acceptors act in concert to prevent inconsistent proposals from being accepted; failing proposals are retried. We cannot assume that the backend services will implement the Paxos acceptor algorithm. Instead, we only require them to provide an *append-only list* that *atomically* appends an incoming message at the end of the list. This abstraction is either readily available or can be layered on top of the interface provided by existing storage service providers (Table 3). With this append-only list abstraction, backend services can act as *passive acceptors*. Clients determine which proposal was "accepted" by examining the log of messages to determine what a normal Paxos acceptor would have done.

Algorithm. With an append-only list, pPaxos becomes a simple adaptation of classic Paxos, where the decision as to what proposal was accepted is performed by proposers. Each client keeps a data structure for each backend service, containing the state it would have if it processed its log as a Paxos acceptor (Figure 4 Lines 1-4). To propose a value, a client sends a `PREPARE` to every storage backend with a proposal number (Lines 7-8); this message is appended to the log at every backend. The proposal number must be unique (e.g., client IDs are used to break ties). The client determines the result of the prepare message by fetching and processing the logs at each backend (Lines 25-29). It aborts its proposal if another client inserted a larger proposal number in the log (Line 10). As in Paxos, the client proposes as the new root the value in the highest numbered proposal "accepted" by any backend server (Lines 12-15), or its own new root if none has been accepted. It sends this value in an `ACCEPT_REQ` message to every backend (Lines 18-19) to be appended to its log; the value is committed if no higher numbered `PREPARE` message intervenes in the log (Lines 20-21, 30-32). When the new root is accepted by a majority, the client can conclude it has committed the new updated value (Line 23). In case it fails, to avoid repeated conflicts the client chooses a random exponential back-off and tries again with an increased proposal number (Lines 33-36).

This setting is similar to the motivation behind Disk Paxos [19]; indeed, pPaxos can be considered as an optimized version of Disk Paxos (Figure 3(b)). Disk Paxos assumes that the storage device provides only a simple block interface. Clients write proposals to their own block on each server, but they must check everyone else's blocks to determine the outcome. Thus, Disk Paxos takes

```
1:  struct Acceptor
2:      round:      promised round number
3:      accepted:   all accepted proposals
4:      backend:    associated backend service
    [Proposer]
5:  procedure ProposeRound(value, round, acceptors)
     prepare:
6:     concurrently
7:     for all a ← acceptors do
8:         Send(⟨PREPARE, round⟩ → a.backend)
9:         Update(a)
10:        if a.round > round then abort
11:    wait until done by a majority of acceptors
     accept:
12:    accepted ← ∪_{a∈acceptors} a.accepted
13:    if |accepted| > 0 then
14:        p ← arg max{p.round | p ∈ accepted}
15:        value ← p.value
16:    proposal ← ⟨round, value⟩
17:    concurrently
18:    for all a ← acceptors do
19:        Send(⟨ACCEPT_REQ, proposal⟩ → a.backend)
20:        Update(a)
21:        if proposal ∉ a.accepted then abort
22:    wait until done by a majority of acceptors
     commit:
23:    return proposal
24: procedure Update(acceptor)
25:    log ← FetchNewLog(acceptor.backend)
26:    for all msg ∈ log do
27:        switch msg do
28:            case ⟨PREPARE, round⟩
29:                acceptor.round ← max(round, acceptor.round)
30:            case ⟨ACCEPT_REQ, proposal⟩
31:                if proposal.round ≥ acceptor.round then
32:                    acceptor.accepted.append(proposal)
33: procedure OnRestartAfterFailure(round)
34:    IncreaseRound
35:    WaitExponentailly
36:    ProposeRound(value, round, acceptors)
    [Passive Acceptor]
37: procedure OnNewMessage(⟨msg, round⟩)
38:    Append(⟨msg, round⟩ → log)
```

Figure 4: pPaxos Algorithm.

time proportional to the product of the number of servers and clients; pPaxos is proportional to number of servers.

pPaxos in action. MetaSync maintains two types of shared metadata: the `master` hash value and service configuration. Unlike a regular file, the configuration is replicated in all backends (in their object stores). Then, MetaSync can uniquely identify the shared data with a three tuple: `(version, master_hash, config_hash)`.

Version is a monotonically increasing number which is uniquely determined for each `master_hash`, `config_hash` pair. This tuple is used in pPaxos to describe the status of a client and is stored in `head_client` and `prev_client`.

The pPaxos algorithm explained above can determine and store the next value of the three tuple. Then, we build the functions listed in Table 1 by using a pPaxos instance per synchronized value. Each client keeps the last value

APIs	Description
propose(prev, next)	Propose a next value of prev. It returns the accepted next value, which could be next or some other value proposed by another client.
get_recent()	Retrieve the most recent value.

Table 1: Abstractions for consistent update.

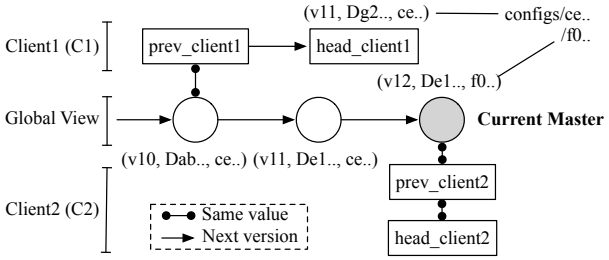


Figure 5: An example snapshot of pPaxos status with two clients. Each circle indicates a pPaxos instance. C1 synchronized against v10. It modified some files but the changes have not been synchronized yet (head_client1). C2 changed some files and the changes were made into v11, then made changes in configuration and synchronized it (v12). Then, it hasn't made any changes. If C1 tries to propose the next value of v10 later, it fails. It needs to merge with v12 and creates v13 head. In addition, C1 can learn configuration changes when getting v12.

with which it synchronized (`prev_client`). To proposes a new value, the client runs pPaxos to update the previous value with the new value. If another value has already been accepted, it can try to update the new value after merging with it. It can repeat this until it successfully updates the master value with its proposed one. This data structure can be logically viewed as a linked list, where each entry points the next hash value, and the tail of the list is the most up-to-date. Figure 5 illustrates an example snapshot of pPaxos status.

Merging. Merging is required when a client synchronizes its local changes (`head`) with the current master that is different from what the client previously synchronized (`prev`). In this case, proposing the current head as the next update to `prev` returns a different value than the proposed head as other clients have already advanced the `master` value. The client has to merge its changes with the current `master` into its head. To do this, MetaSync employs three-way merging as in other version control systems. This allows many conflicts to be automatically resolved. Of course, three-way merging cannot resolve all conflicts, as two clients may change the same parts of a file. In our current implementation, for example, MetaSync generates a new version of the file with `.conflict.N` extension, which allows for users to resolve the conflict manually.

3.4 Replication: Stable Deterministic Mapping

MetaSync replicates objects (in the object store) redundantly across R storage providers (R is configurable, typically $R = 2$) to provide high availability even when a service is temporarily inaccessible. This also provides potentially better performance over wide area networks. Since R is less than the number of services, it is required to maintain information regarding the mapping of objects to services. In our settings, where the storage services passively participate in the coordination protocol, it is particularly expensive to provide a consistent view of this shared information. Not only that, MetaSync requires a mapping scheme that takes into account storage space limits imposed by each storage service; if handled poorly, lack of storage at a single service can block the entire operation of MetaSync, and typical storage services vary in the (free) space they provide, ranging from 2 GB in Dropbox to 2 TB in Baidu. In addition, the mapping scheme should consider a potential reconfiguration of storage services (e.g., increasing storage capacity); upon changes, the re-balancing of distributed objects should be minimal.

Goals. Instead of maintaining the mapping information of each object, we use a stable, deterministic mapping function that locates each object to a group of services over which it is replicated; each client can calculate the same result independently given the same object. Given a hash of an object (mod H), the mapping is: map: $H \rightarrow \{s : |s| = R, s \subset S\}$, where H is the hash space, S is the set of services, and R is the number of replicas. The mapping should meet three requirements:

- R1 Support variations in storage size limits across different services and across different users.
- R2 Share minimal information amongst services.
- R3 Minimize realignment of objects upon removal or addition of a service.

To provide a balanced mapping that takes into account of storage variations of each service (R1), we may use a mapping scheme that represents storage capacity as the number of virtual nodes in a consistent hashing algorithm [24, 36]. Since it deterministically locates each object onto an identifier circle in the consistent hashing scheme, MetaSync can minimize information shared among storage providers (R2).

However, using consistent hashing in this way has two problems: an object can be mapped into a single service over multiple vnodes, which reduces availability even though the object is replicated, and a change in service's capacity—changing the number of virtual nodes, so the size of hash space—requires to reshuffle all the objects distributed across service providers (R3). To solve these problems, we introduce a stable, deterministic mapping scheme that maps an object to a unique set of virtual nodes and also minimizes reshuffling upon any changes to virtual nodes (e.g., changes in configurations). This construction is challenging because our scheme should randomly map each service to a virtual node and balance

```
1: procedure INIT(Services, H)
2:     ▷ H: HashSpace size, bigger values produce better mappings
3:     N ← {(sId, vId) : sId ∈ Services, 0 ≤ vId < Cap(sId)}
4:                         ▷ Cap: normalized capacity of the service
5:     for all i < H do map[i] = Sorted(N, key = md5(i, sId, vId))
6:     return map
7: procedure GETMAPPING(object, R)
8:     i ← hash(object) mod H
9:     return Uniq(map[i], R)    ▷ Uniq: the first R distinct services
```

Figure 6: The deterministic mapping algorithm.

object distribution, but at the same time, be stable enough to minimize remapping of replicated objects upon any change to the hashing space. The key idea is to achieve the random distribution via hashing, and achieve stability of remapping by sorting these hashed values; for example, an increase of storage capacity will change the order of existing hashed values by at most one.

Algorithm. Our stable deterministic mapping scheme is formally described in Figure 6. For each backend storage provider, it utilizes multiple virtual storage nodes, where the number of virtual nodes per provider is proportional to the storage capacity limit imposed by the provider for a given user. (The concept of virtual nodes is similar to that used in systems such as Dynamo [14].) Then it divides the hash space into H partitions. H is configurable, but remains fixed even as the service configuration changes. H can be arbitrary large but need to be larger than the sum of normalized capacity, with larger values producing better-balanced mappings for heterogeneous storage limits. During initialization, the mapping scheme associates differently ordered lists of virtual nodes with each of the H partitions. The ordering of the virtual nodes in the list associated with a partition is determined by hashing the index of the partition, the service ID, and the virtual node ID. Given an object hash n, the mapping returns the first R *distinct* services from the list associated with the (n mod H)th partition, similar to Rendezvous hashing [37].

The mapping function takes as input the set of storage providers, the capacity settings, value of H, and a hash function. Thus, it is necessary to share only these small pieces of information in order to reconstruct this mapping across different clients sharing a set of files. The list of services and the capacity limits are part of the service configuration and shared through the `config` file. The virtual node list is populated proportionally to service capacity, and the ordering in each list is determined by a uniform hash function. Thus, the resulting mapping of objects onto services should be proportional to service capacity limits with large H. Lastly, when N nodes are removed from or added to the service list, an object needs to be newly replicated into at most N nodes.

Example. Figure 7 shows an example of our mapping scheme with four services ($|S| = 4$) providing 1GB or 2GB of free spaces–for example, A(1) means that service A provides 1GB of free space. Given the replication requirement ($R = 2$) and the hash space ($H = 20$), we can populate the initial mapping with `Init` function from Figure 6. Subfigures (a) and (b) illustrate the realignment of objects upon the removal of service B(2) and the inclusion of a new service E(3).

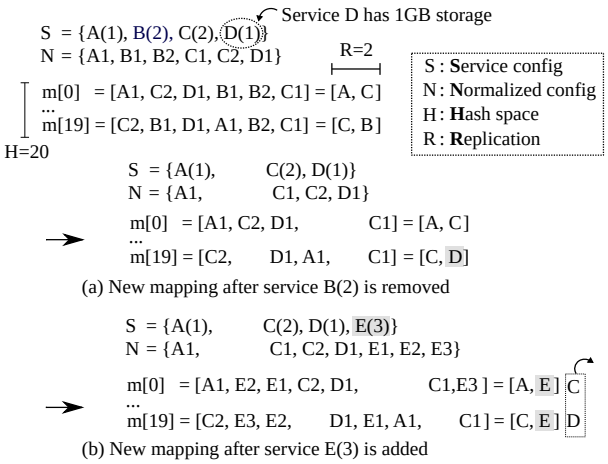


Figure 7: An example of deterministic mapping and its reconfigurations. The initial mapping is deterministically generated by Figure 6, given the configuration of four services, $A(1), B(2), C(2), D(1)$ where the number represents the capacity of each service. (a) and (b) show new mappings after configuration is changed. The grayed mappings indicate the new replication upon reconfiguration, and the dotted rectangle in (b) represents replications that will be garbage collected.

3.5 Translators

MetaSync provides a plugin system, called Translators, for encryption and integrity check. Translators is highly modular so can easily be extended to support a variety of other transformations such as compression. Plugins should implement two interfaces, `put` and `get`, which are invoked before storing objects to and after retrieving them from backend services. Plugins are chained, so that when an object is stored, MetaSync invokes a chain of `put` calls in sequence. Similarly, when an object is retrieved, it goes through the same chain but in reverse.

Encryption translator is currently implemented using a symmetric key encryption (AES-CBC). MetaSync keeps the encryption key locally, but does not store on the backends. When a user clones the directory in another device, the user needs to provide the encryption key. Integrity checker runs hash function over retrieved object and compares the digest against the file name. If it does not match, it drops the object and downloads the object by using other backends from the mapping. It needs to run only in the `get` chain.

3.6 Fault Tolerance

To operate on top of multiple storage services that are often unreliable (they are free!), faulty (they scan and tamper with your files), and insecure (some are outside of your country), MetaSync should tolerate faults.

Data model. By replicating each object into multiple backends (R in §3.4), MetaSync can tolerate loss of file or directory objects, and tolerate temporal unavailability or failures of $R-1$ concurrent services.

File integrity. Similarly with other version control systems [20], the hash tree ensures each object's hash value is valid from the root (`master, head`). Then, each object's integrity can be verified by calculating the hash of the content and comparing with the name when it is retrieved from the backend service. The value of `master` can be signed to protect against tampering. When MetaSync finds an altered object file, it can retrieve the data from another replicated service through the deterministic mapping.

Consistency control. MetaSync runs pPaxos for serializing updates to the shared value for `config` and `master`. The underlying pPaxos protocol requires $2f+1$ acceptors to ensure correctness if f acceptors may fail under the fail-stop model.

Byzantine Fault Tolerant pPaxos. pPaxos can be easily extended to make it resilient to other forms of service failures, e.g., faulty implementations of the storage service APIs and even actively malicious storage services. Note that even with Byzantine failures, each object is protected in the same way through replication and integrity checks. However, updates of global view need to be handled more carefully. We assume that clients are trusted and work correctly, but backend services may have Byzantine behavior. When sending messages for proposing values, a client needs to sign it. This ensures that malicious backends cannot create arbitrary log entries. Instead, the only possible malicious behavior is to break consistency by omitting log entries and reordering them when clients fetch them; a backend server may send any subset of the log entries in any order. Under this setting, pPaxos works similarly with the original algorithm, but it needs $3f+1$ acceptors when f may concurrently fail. Then, for each prepare or accept, a proposing client needs to wait until $2f+1$ acceptors have prepared or accepted, instead of $f+1$. It is easy to verify the correctness of this scheme. When a proposal gets $2f+1$ accepted replies, even if f of the acceptors are Byzantine, the remaining $f+1$ acceptors will not accept a competing proposal. As a consequence, competing proposals will receive at most $2f$ acceptances and will fail to commit. Note that each file object is still replicated at only $f+1$ replicas, as data corruption can be detected and corrected as long as there is a single non-Byzantine service. As a consequence, the only additional overhead of making the system tolerate Byzantine failures is to require a larger quorum ($2f+1$) and a larger number of storage services ($2f+1$) for implementing the synchronization operation associated with updating `master`.

APIs	Description
(a) Storage abstraction	
get(`path`)	Retrieve a file at `path`
put(`path`, `data`)	Store `data` at `path`
delete(`path`)	Delete a file at `path`
list(`path`)	List all files under `path` directory
poll(`path`)	Check if `path` was changed
(b) Synchronization abstraction	
append(`path`, `msg`)	Append `msg` to the list at `path`
fetch(`path`)	Fetch a log from `path`

Table 2: Abstractions for backend storage services.

3.7 Backend abstractions

Storage abstraction. Any storage service having an interface to allow clients to read and write files can be used as a storage backend of MetaSync. More specifically, it needs to provide the basis for the the functions listed in Table 2(a). Many storage services provide a developer toolkit to build a customized client accessing user files [16, 21]; we use these APIs to build MetaSync. Not only cloud services provide these APIs, it is also straightforward to build these functions on user's private servers through SSH or FTP. MetaSync currently supports backends with the following services: Dropbox, GoogleDrive, OneDrive, Box.net, Baidu, and local disk.

Synchronization abstraction. To build the primitive for synchronization, an append-only log, MetaSync can use any services that provide functions listed in Table 2(b). How to utilize the underlying APIs to build the append-only log varies across services. We summarize how MetaSync builds it for each provider in Table 3.

3.8 Other Issues

Sharing. MetaSync allows users to share a folder and work on the folder. While not many backend services have APIs for sharing functions—only Google Drive and Box have it among services that we used—others can be implemented through browser emulation. Once sharing invitation is sent and accepted, synchronization works the same way as in the one-user case. If files are encrypted, we assume that all collaborators share the encryption key.

Collapsing directory. All storage services manage individual files for uploading and downloading. As we see later in Table 4, throughput for uploading and downloading small files are much lower than those for larger files. As an optimization, we collapse all files in a directory into a single object when the total size is small enough.

4 Implementation

We have implemented a prototype of MetaSync in Python, and the total lines of code is about 7.5K. The current prototype supports five backend services including Box, Baidu, Dropbox, Google Drive and OneDrive,

and works on all major OSes including Linux, Mac and Windows. MetaSync provides two front-end interfaces for users, a command line interface similar to git and a synchronization daemon similar to Dropbox.

Abstractions. Storage services provide APIs equivalent to MetaSync's `get()` and `put()` operations defined in Table 2. Since each service varies in its support for the other operations, we summarize the implementation details of each service provider in Table 3. For implementing synchronization abstractions, `append()` and `fetch()`, we utilized the *commenting* features in Box, Google and OneDrive, and *versioning* features in Dropbox. If a service does not provide any efficient ways to support synchronization APIs, MetaSync falls back to the default implementation of those APIs that are built on top of their storage APIs, described for Baidu in Table 3. Note that for some services, there are multiple ways to implement the synchronization abstractions. In that case, we chose to use mechanisms with better performance.

Front-ends. The MetaSync daemon monitors file changes by using `inotify` in Linux, `FSEvents` and `kQueue` in Mac and `ReadDirectoryChangesW` in Windows, all abstracted by the Python library `watchdog`. Upon notification, it automatically uploads detected changes into backend services. It batches consecutive changes by waiting 3 more seconds after notification so that all modified files are checked in as a single commit to reduce synchronization overhead. It also polls to find changes uploaded from other clients; if so, it merges them into the local drive. The command line interface allows users to manually manage and synchronize files. The usage of MetaSync commands is similar to that of version control systems (e.g., `metasync init`, `clone`, `checkin`, `push`).

5 Evaluation

This section answers the following questions:

- What are the performance characteristics of pPaxos?
- How quickly does MetaSync reconfigure mappings as services are added or removed?
- What is the end-to-end performance of MetaSync?

Each evaluation is done on Linux servers connected to campus network except for synchronization performance in §5.3. Since most services do not have native clients for Linux, we compared synchronization time for native clients and MetaSync on Windows desktops.

Before evaluating MetaSync, we measured the performance variance of services in Table 4 via their APIs. One important observation is that all services are slow in handling small files. This provides MetaSync the opportunity to outperform them by combining small objects.

5.1 pPaxos performance

We measure how quickly pPaxos reaches consensus as we vary the number of concurrent proposers. The results of the experiment with 1-5 proposers over 5 storage providers are shown in Figure 8. A single run of pPaxos took about 3.2 sec on average under a single writer model to verify acceptance of the proposal when using all 5 storage providers. This requires at least four round trips: `PREPARE` (Send, FetchNewLog) and `ACCEPT_REQ` (Send, FetchNewLog) (Figure 4) (there could be multiple rounds in FetchNewLog depending on the implementation for each service). It took about 7.4 sec with 5 competing proposers. One important thing to emphasize is that, even with a slow connection to Baidu, pPaxos can quickly be completed with a single winner of that round. Also note that when compared to a single storage provider, the latency doesn't degrade with the increasing number of storage providers—it is slower than using a certain backend service (Google), but it is similar to the median case as the latency depends on the proposer getting responses from the majority.

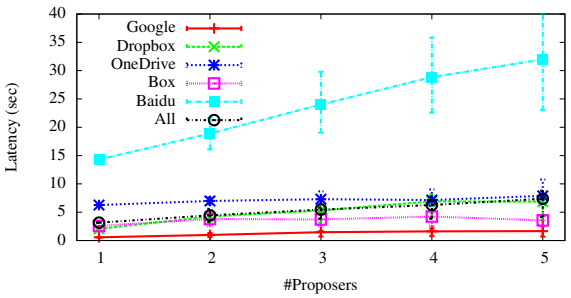


Figure 8: Latency (sec) to run a single pPaxos round with combinations of backend services and competing proposers: when using 5 different storage providers as backend nodes (all), the common path of pPaxos at a single proposer takes 3.2 sec, and the slow path with 5 competing proposers takes 7.4 sec in median. Each measurement is done 5 times, and it shows the average latency.

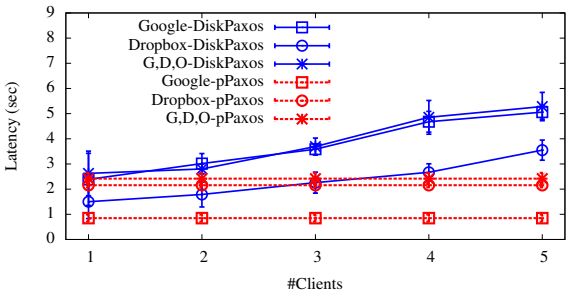


Figure 9: Comparison of latency (sec) to run a single round for Disk Paxos and pPaxos with varying number of clients when only one client proposes a value. Each line represents different backend setting; G,D,O: Google, Dropbox, and Onedrive. While pPaxos is not affected by the number of clients, Disk Paxos latency increases with it. Each measurement is done 5 times, and it shows the average latency.

Next, we compare the latency of a single round for pPaxos with that for Disk Paxos [19]. We build Disk Paxos with APIs by assigning a file as a block for each client. Figure 9 shows the results with varying number of clients when only one client proposes a value. As we explain in §3.3, Disk Paxos gets linearly slower with increasing number of clients even when all other clients are

Service	Synchronization API		Storage API
	`append(path, msg)`	`fetch(path)`	`poll(path)`
Box Google OneDrive	Create an empty `path` file and add `msg` as *comments* to the `path` file.	Download the entire comments attached on the `path` file.	Use `events` API, allowing long polling. (Google, OneDrive: periodically list `pPaxos` directory to see if any changes)
Baidu	Create a `path` directory, and consider each file as a log entry containing `msg`. For each entry, we create a file with an increasing sequence number as its name. If the number is already taken, we will get an exception and try with a next number.	List the `path` directory, and download new log entries since last fetch (all files with subsequent sequence numbers).	Use `diff` API to monitor if there is any change over the user's drive.
Dropbox	Create a `path` file, and overwrite the file with a new log entry containing `msg`, relying on Dropbox's versioning.	Request a list of versions of the `path` file.	Use `longpoll_delta`, a blocked call, that returns if there is a change under `path`.
Disk†	Create a `path` file, and append `msg` at the end of the file.	Read the new entries from the `path` file.	Emulate long polling with a condition variable.

Table 3: Implementation details of synchronization and storage APIs for each service. Note that implementations of other storage APIs (e.g., `put()`) can be directly built with APIs provided by services, with minor changes (e.g., supporting namespace). Disk† is implemented for testing.

Services	1 KB		1 MB		10 MB		100 MB	
	U.S.	China	U.S.	China	U.S.	China	U.S.	China
Baidu	0.7 / 0.8	1.8 / 2.6	0.21 / 0.22	0.12 / 1.48	0.22 / 0.94	0.13 / 2.64	0.24 / 1.07	0.13 / 3.38
Box	1.4 / 0.6	0.8 / 0.2	0.73 / 0.44	0.11 / 0.12	4.79 / 3.38	0.13 / 0.68	17.37 / 15.77	0.13 / 1.08
Dropbox	1.2 / 1.3	0.5 / 0.5	0.59 / 0.69	0.10 / 0.20	2.50 / 3.48	0.09 / 0.41	3.86 / 14.81	0.13 / 0.68
Google	1.4 / 0.8	-	1.00 / 0.77	-	5.80 / 5.50	-	9.43 / 26.90	-
OneDrive	0.8 / 0.5	0.3 / 0.1	0.45 / 0.34	0.01 / 0.05	3.13 / 2.08	0.11 / 0.12	7.89 / 6.33	0.11 / 0.44
	KB/s		MB/s		MB/s		MB/s	

Table 4: Upload and download bandwidths of four different file sizes on each service from U.S. and China. This preliminary experiment explains three design constrains of MetaSync. First, all services are extremely slow in handling small files, 7k/34k times slower in uploading/downloading 1 KB files than 100 MB on Google storage service. Second, the bandwidth of each service approaches its limit at 100 MB. Third, performance varies with locations, 30/22 times faster in uploading/downloading 100 MB when using Dropbox in U.S. compared to China.

inactive, since it must read the current state of all clients.

5.2 Deterministic mapping

We then evaluate how fairly our deterministic mapping distributes objects into storage services with different capacity, in three replication settings ($R = 1, 2$). We test our scheme by synchronizing source tree of Linux kernel 3.10.38, consisting of a large number of small files (464 MB), to five storage services, as detailed in Table 5. We use $H = (5 \times \text{sum of normalized space}) = 10,410$ for this testing. In $R = 1$, where we upload each object once, MetaSync locates objects in balance to all services—it uses 0.02% of each service's capacity consistently. However, since Baidu provides 2TB (98% of MetaSync's capacity in this configuration), most of the objects will be allocated into Baidu. This situation improves for $R = 2$, since objects will be placed into other services beyond Baidu. Baidu gets only 6.2 MB of more storage when increasing $R = 1 \rightarrow 2$, and our mapping scheme preserves the balance for the rest of services (using 1.3%).

The entire mapping plan is deterministically derived from the shared `config`. The size of information to be shared is small (less than 50B for the above example), and the size of the populated mapping is about 3MB.

Reconfiguration	#Objects Added / Removed	Time (sec) Replication / GC
$S = 4, R = 2 \rightarrow 3$	101 / 0	33.7 / 0.0
$S = 4 \rightarrow 3, R = 2$	54 / 54	19.6 / 40.6
$S = 3 \rightarrow 4, R = 2$	54 / 54	29.8 / 14.7

Table 6: Time to relocate 193 MB amount of objects (photo-sharing workloads in Table 7) on increasing the replication ratio, removing an existing service, and adding one more service. MetaSync quickly rebalances its mapping (and replication) based on its new `config`. We used four services, Dropbox, Box, GoogleDrive, and OneDrive ($S = 4$) for experimenting with the replication, including ($S = 3 \rightarrow 4$) and excluding OneDrive ($S = 4 \rightarrow 3$) for re-configuring storage services.

The relocation scheme is resilient to changes as well, meaning that redistribution of objects is minimal. As in Table 6, when we increased the configured replication by one ($R = 2 \rightarrow 3$) with 4 services, MetaSync replicated 193 MB of objects in about half a minute. When we removed a service from the configuration, MetaSync redistributed 96.5 MB of objects in about 20 sec. After adding and removing a storage backend, MetaSync needs to delete redundant objects from the previous configuration, which took 40.6/14.7 sec for removing/adding OneDrive in our experiment. However, the garbage collection will be asynchronously initiated during idle time.

Repl.	Dropbox (2 GB)	Google (15 GB)	Box (10 GB)	OneDrive (7 GB)	Baidu (2048 GB)	Total (2082 GB)
$R = 1$	77 (0.09%) 0.34 MB (0.02%)	660 (0.75%) 2.87 MB (0.02%)	475 (0.54%) 2.53 MB (0.02%)	179 (0.20%) 0.61 MB (0.01%)	86,739 (98.42%) 463.8 MB (0.02%)	88,130 (100%) 470.1 MB (0.02%)
$R = 2$	5,297 (3.01%) 27.4 MB (1.34%)	39,159 (22.22%) 206.4 MB (1.34%)	25,332 (14.37%) 138.2 MB (1.35%)	18,371 (10.42%) 98.3 MB (1.37%)	88,101 (49.98%) 470.0 MB (0.02%)	176,260 (100%) 940.3 MB (0.04%)

Table 5: Replication results by our deterministic mapping scheme (§3.4) for Linux kernel 3.10.38 (Table 7) on 5 different services with various storage space, given for free. We synchronized total 470 MB of files, consisting of 88k objects, and replicated them across all storage backends. Note that for this mapping test, we turned off the optimization of collapsing directories. Our deterministic mapping distributed objects in balance: for example, in $R = 2$, Dropbox, Google, Box and OneDrive used consistently 1.35% of their space, even with 2-15 GB of capacity variation. Also, $R = 1$ approaches to the perfect balance, using 0.02% of storage space in all services.

5.3 End-to-end performance

We selected three workloads to demonstrate performance characteristics. First, Linux kernel source tree (2.6.1) represents the most challenging workload for all storage services due to its large volume of files and directory (920 directories and 15k files, total 166 MB). Second, MetaSync's paper represents a causal use of synchronization service for users (3 directories and 70 files, total 1.6 MB). Third, sharing photos is for maximizing the throughput of each storage service with bigger files (50 files, total 193 MB).

Table 7 summarizes our results for end-to-end performance for all workloads, comparing MetaSync with the native clients provided by each service. Each workload was copied into one client's directory before synchronization is started. The synchronization time was measured as the length of interval between when one desktop starts to upload files and the creation time of the last file synced on the other desktop. We also measured the synchronization time for all workloads by using MetaSync with different settings. MetaSync outperforms any individual service for all workloads. Especially for Linux kernel source, it took only 12 minutes when using 4 services (excluding Baidu located outside of the country) compared to more than 2 hrs with native clients. This improvement is possible due to using concurrent connections to multiple backends, and optimizations like collapsing directories. Although these native clients may not be optimized for the highest possible throughput, considering that they may run as a background service, it would be beneficial for users to have a faster option. It is also worth noting that replication helps sync time, especially when there is a slower service, as shown in the case with $S = 5, R = 1, 2$; a downloading client can use faster services while an uploading client can upload a copy in the background.

Clone. Storage services often limit their download throughput: for example, MetaSync can download at 5.1 MB/s with Dropbox as a backend, and at 3.4 MB/s with Google Drive, shown in Figure 10. Note that downloading is done already by using concurrent connections even to the same service. By using multiple storage services, MetaSync can fully exploit the bandwidth of local connection of users, not limited by the allowed throughput of each service. For example, MetaSync with both services and R=2 took 25.5 sec for downloading 193 MB data, which is at 7.6 MB/s.

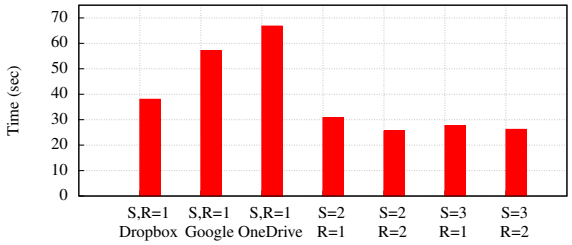


Figure 10: Time to clone 193 MB photos. When using individual services as a backend (Dropbox, Google, and OneDrive), MetaSync took 40-70 sec to clone, but improved the performance, 25-30 sec (30%) by leveraging the distributions of objects across multiple services.

6 Related Work

A major line of related work, starting with Farsite [2] and SUNDR [26] but carrying through SPORC [17], Frientegrity [18], and Depot [27], is how to provide tamper resistance and privacy on untrusted storage server nodes. These systems assume the ability to specify the client-server protocol, and therefore cannot run on unmodified cloud storage services. A further issue is equivocation; servers may tell some users that updates have been made, and not others. Several of these systems detect and resolve equivocations after the fact, resulting in a weaker consistency model than MetaSync's linearizable updates. A MetaSync user knows that when a push completes, that set of updates is visible to all other users and no conflicting updates will be later accepted. Like Farsite, we rely on a stronger assumption about storage system behavior—that failures across multiple storage providers are independent, and this allows us to provide a simpler and more familiar model to applications and users.

Likewise, several systems have explored composing a storage layer on top of existing storage systems. Syndicate [32] is designed as an API for applications; thus, they delegate design choices such as how to manage files and replicate to application policy. SCFS [5] imple-

Workload	Dropbox	Google	Box	OneDrive	Baidu	MetaSync			
						$S=5,R=1$	$S=5,R=2$	$S=4,R=1$	$S=4,R=2$
Linux kernel source	2h 45m	> 3hrs	> 3hrs	2h 03m	> 3hrs	1h 8m	13m 51s	18m 57s	12m 18s
MetaSync paper	48	42	148	54	143	55	50	27	26
Photo sharing	415	143	536	1131	1837	1185	180	137	112

Table 7: Synchronization performance of 5 native clients provided by each storage service, and with four different settings of MetaSync. For $S=5,R=1$, using all of 5 services without replication, MetaSync provides comparable performance to native clients–median speed for MetaSync paper and photo sharing, but outperforming for Linux kernel workloads. However, for $S=5,R=2$ where replicating objects twice, MetaSync outperform >10 times faster than Dropbox in Linux kernel and 2.3 times faster in photo sharing; we can finish the synchronization right after uploading a single replication set (but complete copy) and the rest will be scheduled in background. To understand how slow straggler (Baidu) affects the performance ($R=1$), we also measured synchronization time on $S=4$ without Baidu, where MetaSync vastly outperforms all services.

ments a sharable cloud-backed file system with multiple cloud storage services. Unlike MetaSync, Syndicate and SCFS assume separate services for maintaining metadata and consistency. RACS [1] uses RAID-like redundant striping with erasure coding across multiple cloud storage providers. Erasure coding can also be applied to MetaSync and is part of our future work. SpanStore [39] optimizes storage and computation placement across a set of paid data centers with differing charging models and differing application performance. As they are targeting general-purpose infrastructure like EC2, they assume the ability to run code on the server. BoxLeech [22] argues that aggregating cloud services might abuse them especially given a user may create many free accounts even from one provider, and demonstrates it with a file sharing application. GitTorrent [3] implements a decentralized GitHub hosted on BitTorrent. It uses BitCoin's blockchain as a method of distributed consensus.

Perhaps closest to our intent is DepSky [4]; it proposes a cloud of clouds for secure, byzantine-resilient storage, and it does not require code execution on the servers. However, they assume a more restricted use case. Their basic algorithm assumes at most one concurrent writer. When writers are at the same local network, concurrent writes are coordinated by an external synchronization service like ZooKeeper. Otherwise, it has a possible extension that can support multiple concurrent updates without an external service, but it requires clock synchronization between clients. MetaSync makes no clock assumptions about clients, it is designed to be efficient in the common case where multiple clients are making simultaneous updates, and it is non-blocking in the presence of either client or server failures. DepSky also only provides strong consistency for individual data objects, while MetaSync provides strong consistency across all files in a repository.

Our implementation integrates and builds on the ideas in many earlier systems. Obviously, we are indebted to earlier work on Paxos [25] and Disk Paxos [19]; we earlier provided a detailed evaluation of these different approaches. We maintain file objects in a manner similar to a distributed version control system like git [20]; the Ori file system [28] takes a similar approach. However, MetaSync can combine or split each file object for more efficient storage and retrieval. Content-based addressing has been used in many file systems [8, 11, 26, 28, 35]. MetaSync uses content-based addressing for a unique purpose, allowing us to asynchronously uploading or downloading objects to backend services. While algorithms for distributing or replicating objects have also been proposed and explored by past systems [10, 33, 34], the replication system in MetaSync is designed to minimize the cost of reconfiguration to add or subtract a storage service and also to respect the diverse space restrictions of multiple backends.

7 Conclusion

MetaSync provides a secure, reliable, and performant file synchronization service on top of popular cloud storage providers. By combining multiple existing services, it enables a highly available service during the outage or even shutdown of a service provider. To achieve a consistent update among cloud services, we devised a client-based Paxos, called pPaxos, that can be implemented without modifying any existing APIs. To minimize the redistribution of replicated files upon a reconfiguration of services, we developed a deterministic, stable replication scheme that requires minimal amount of shared information among services (e.g., configuration). MetaSync supports five commercial storage backends (in current open source version), and outperforms the fastest individual service in synchronization and cloning, by 1.2-10× on our benchmarks. MetaSync is publicly available for download and use (`http://uwnetworkslab.github.io/metasync/`).

Acknowledgments

We gratefully acknowledge our shepherd Feng Qin and the anonymous reviewers. This work was supported by the National Science Foundation (CNS-0963754, 1318396, and 1420703) and Google. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Taesoo Kim was partly supported by MSIP/IITP [B0101-15-0644].

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2010.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2002.

[3] C. Ball. Announcing gittorrent: A decentralized github. `http://blog.printf.net/articles/2015/05/29/announcing-gittorrent-a-decentralized-github/`, 2015.

[4] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of ACM EuroSys conference*, pages 31–46, 2011.

[5] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, 2014.

[6] C. Brooks. Cloud Storage Often Results in Data Loss. `http://www.businessnewsdaily.com/1543-cloud-data-storage-problems.html`, October 2011.

[7] S. Byrne. Microsoft OneDrive for business modifies files as it syncs. `http://www.myce.com/news/microsoft-onedrive-for-business-modifies-files-as-it-syncs-71168`, Apr. 2014.

[8] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[9] Canonical Ltd. Ubuntu One: Shutdown notice. `https://one.ubuntu.com/services/shutdown`.

[10] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, pages 37–48, 2013.

[11] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 USENIX Conference on Annual Technical Conference (ATC)*, 2009.

[12] J. Cook. All the different ways that 'icloud' naked celebrity photo leak might have happened. `http://www.businessinsider.com/icloud-naked-celebrity-photo-leak-2014-9`, Sept. 2014.

[13] J. Ćurn. How a bug in dropbox permanently deleted my 8000 photos. `http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html`, 2014.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[15] Dropbox. Thanks for helping us grow. `https://blog.dropbox.com/2014/05/thanks-for-helping-us-grow/`, May 2014.

[16] dropbox-api. Dropbox API. `https://www.dropbox.com/static/developers/dropbox-python-sdk-1.6-docs/index.html`, Apr. 2014.

[17] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[18] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[19] E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, Feb. 2003.

[20] git. Git Internals - Git Objects. `http://git-scm.com/book/en/Git-Internals-Git-Objects`.

[21] google-api. Google Drive API. https://developers.google.com/drive/v2/reference/, Apr. 2014.

[22] R. Gracia-Tinedo, M. S. Artigas, and P. G. López. Cloud-as-a-Gift: Effectively exploiting personal cloud free accounts via REST APIs. In *IEEE 6th International Conference on Cloud Computing (CLOUD)*, 2013.

[23] G. Huntley. Dropbox confirms that a bug within selective sync may have caused data loss. https://news.ycombinator.com/item?id=8440985, Oct. 2014.

[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663. ACM, 1997.

[25] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–9, 2004.

[27] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–12, 2010.

[28] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, history, and grafting in the Ori file system. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP)*, pages 151–166, 2013.

[29] R. C. Merkle. A digital signature based on a conventional encryption function. In *In Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, pages 369–378, Santa Barbara, CA, 1987.

[30] E. Mill. Dropbox Bug Can Permanently Lose Your Files . https://konklone.com/post/dropbox-bug-can-permanently-lose-your-files, October 2012.

[31] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.

[32] J. Nelson and L. Peterson. Syndicate: Democratizing cloud storage and caching through service composition. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 46:1–46:2, 2013.

[33] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2012.

[34] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[35] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001. ISBN 1-58113-411-8.

[37] D. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

[38] Z. Whittaker. Dropbox under fire for 'DMCA takedown' of personal folders, but fears are vastly overblown. http://www.zdnet.com/dropbox-under-fire-for-dmca-takedown-7000027855, Mar. 2014.

[39] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 292–308, 2013.

Pyro: A Spatial-Temporal Big-Data Storage System

Shen Li* Shaohan Hu* Raghu Ganti† Mudhakar Srivatsa† Tarek Abdelzaher*
*University of Illinois at Urbana-Champaign †IBM Research

Abstract

With the rapid growth of mobile devices and applications, geo-tagged data has become a major workload for big data storage systems. In order to achieve scalability, existing solutions build an additional index layer above general purpose distributed data stores. Fulfilling the semantic level need, this approach, however, leaves a lot to be desired for execution efficiency, especially when users query for moving objects within a high resolution geometric area, which we call geometry queries. Such geometry queries translate to a much larger set of range scans, forcing the backend to handle orders of magnitude more requests. Moreover, spatial-temporal applications naturally create dynamic workload hotspots[1], which pushes beyond the design scope of existing solutions. This paper presents Pyro, a spatial-temporal big-data storage system tailored for high resolution geometry queries and dynamic hotspots. Pyro understands geometries internally, which allows range scans of a geometry query to be aggregately optimized. Moreover, Pyro employs a novel replica placement policy in the DFS layer that allows Pyro to split a region without losing data locality benefits. Our evaluations use NYC taxi trace data and an 80-server cluster. Results show that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries compared to the state-of-the-art solutions. Pyro further achieves 10X throughput improvement on $100m \times 100m$ rectangle geometries[2].

1 Introduction

The popularity of mobile devices is growing at an unprecedented rate. According to the report published by the United Nations International Telecommunication Union [1], mobile penetration rates are now about equal to the global population. Thanks to positioning modules in mobile devices, a great amount of information generated today is tagged with geographic locations. For example, users can share tweets and Instagram images with location information with family and friends; taxi companies collect pick-up and drop-off events data with geographic location information as well. The abundances of geo-tagged data give birth to a whole range of applications that issue spatial-temporal queries. These queries, which we call geometry queries, request information about moving objects within a user-defined geometric area. Despite the urgent need, no existing systems manage to meet both the scalability and efficiency requirements for spatial-temporal data. For example, geospatial databases [2] are optimized for spatial data, but usually fall short on scalability on handling big-data applications, whereas distributed data stores [3–6] scale well but quite often yield inefficiencies when dealing with geometry queries.

Distributed data stores, such as HBase [3], Cassandra [4], and DynamoDB [5], have been widely used for big-data storage applications. Their key distribution algorithms can be categorized into two classes: random partitioning and ordered partitioning. The former randomly distributes keys into servers, while the latter divides the key space into subregions such that all keys in the same subregion are hosted by the same server. Compared to random partitioning, ordered partitioning considerably benefits range scans, as querying all servers in the cluster can then be avoided. Therefore, existing solutions for spatial-temporal big-data applications, such as MD-HBase [7], and ST-HBase [8], build index layers above the ordered-partitioned HBase to translate a geometry query into a set of range scans. Then, they submit those range scans to HBase, and aggregate the returned data from HBase to answer the query source, inheriting scalability properties from HBase. Although these solutions fulfill the semantic level requirement of spatial-temporal applications, moving hotspots and large geometry queries still cannot be handled efficiently.

Spatial-temporal applications naturally generate moving workload hotspots. Imagine a million people simultaneously whistle taxis after the New Year's Eve at NYC's Times Square. Or during every morning rush hour, people driving into the city central business district search for the least congested routes. Ordered partitioning data stores usually mitigate hotspots by splitting an overloaded region into multiple daughter regions, which can then be moved into different servers. Nevertheless, as region data may still stay in the parent region's server, the split operation prevents daughter regions from enjoying data locality benefits. Take HBase as an example. Region servers in HBase usually co-locate with HDFS datanodes. Under this deployment, one replica of all region data writes to the region server's storage disks, which allows get/scan requests to be served using local data. Other replicas spread randomly in the entire cluster. Splitting and moving a region into other servers disable data locality benefits, forcing daughter regions to fetch data from remote servers. Therefore, moving hotspots often lead to performance degradation.

In this paper, we present Pyro, a holistic spatial-temporal big-data storage system tailored for high resolution geometry queries and moving hotspots. Pyro con-

[1] The hotspot in this paper refers to a geographic region that receives a large amount of geometry queries within a certain amount of time.

[2] The reason of using small geometries in this experiment is that the baseline solution results in excessively long delay when handling even a single large geometry.

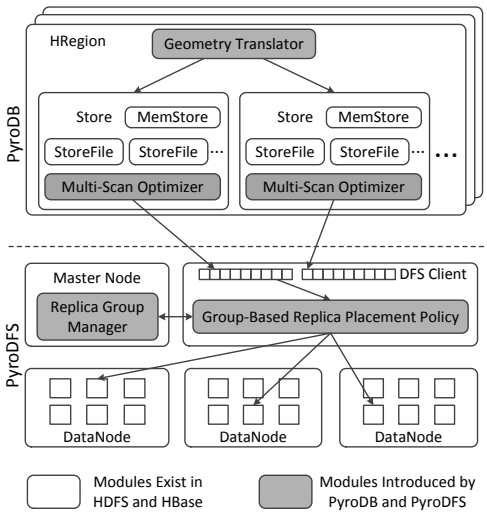


Figure 1: Pyro Architecture

sists of PyroDB and PyroDFS, corresponding to HBase and HDFS respectively. This paper makes three major contributions. First, PyroDB internally implements Moore encoding to efficiently translate geometry queries into range scans. Second, PyroDB aggregately minimizes IO latencies of the multiple range scans generated by the same geometry query using dynamic programming. Third, PyroDFS employs a novel DFS block grouping algorithm that allows Pyro to preserve data locality benefits when PyroDB splits regions during hotspots dynamics. Pyro is implemented by adding 891 lines of code into Hadoop-2.4.1, and another 7344 lines of code into HBase-0.99. Experiments using NYC taxi dataset [9, 10] show that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries. Pyro further achieves 10X throughput improvement on $100m \times 100m$ rectangle geometries.

The remainder of this paper is organized as follows. Section 2 provides background and design overview. Then, major designs are described in Section 3. Implementations and evaluations are presented in Sections 4 and 5 respectively. We survey related work in Section 6. Finally, Section 7 concludes the paper.

2 Design Overview

Pyro consists of PyroDB and PyroDFS. The design of PyroDB and PyroDFS are based on HBase and HDFS respectively. Figure 1 shows the high-level architecture, where shaded modules are introduced by Pyro.

2.1 Background

HDFS [11] is an open source software based on GFS [12]. Due to its prominent fame and universal deployment, we skip the background description.

HBase is a distributed, non-relational database running on top of HDFS. Following the design of BigTable [13], HBase organizes data into a 3D table of rows, columns, and cell versions. Each column belongs to a column family. HBase stores the 3D table as a key-value store. The key consists of row key, column family key, column qualifier, and timestamp. The value contains the data stored in the cell.

In HBase, the entire key space is partitioned into regions, with each region served by an HRegion instance. HRegion manages each column family using a *Store*. Each Store contains one MemStore and multiple StoreFiles. In the write path, the data first stays in the MemStore. When the MemStore reaches some pre-defined flush threshold, all key-value pairs in the MemStore are sorted and flushed into a new StoreFile in HDFS. Each StoreFile wraps an HFile, consisting of a series of data blocks followed by meta blocks. In this paper, we use meta blocks to refer to all blocks that store meta, data index, or meta index. In the read path, a request first determines the right HRegions to query, then it searches all StoreFiles in those regions to find target key-value pairs.

As the number of StoreFiles increases, HBase merges them into larger StoreFiles to reduce the overhead of read operations. When the size of a store increases beyond a threshold, its HRegion splits into two daughter regions, with each region handles roughly half of its parent's keyspace. The two daughter regions initially create reference files pointing back to StoreFiles of their past parent region. This design postpones the overhead of copying region data to daughter region servers at the cost of losing data locality benefits. The next major compaction materializes the reference files into real StoreFiles.

HBase has become a famous big-data storage system for structured data [14], including data for location-based services. Many location-based services share the same request primitive that queries information about moving objects within a given geometry, which we call geometry queries. Unfortunately, HBase suffers inefficiencies when serving geometry queries. All cells in HBase are ordered based on their keys in a one-dimensional space. Casting a geometry into that one-dimensional space inevitably results in multiple disjoint range scans. HBase handles those range scans individually, preventing queries to be aggregately optimized. Moreover, location-based workloads naturally create moving hotspots in the backend, requiring responsive resource elasticity in every HRegion. HBase handles workload hotspots by efficiently splitting regions, which sacrifices data locality benefits for newly created daughter regions. Without data locality, requests will suffer increased response time after splits. Above observations motivate us to design Pyro, a data store specifically tailored for geometry queries.

2.2 Architecture

Figure 1 shows the high-level architecture of Pyro. Pyro internally uses Moore encoding algorithm [15–18] to cast two-dimensional data into one-dimensional Moore

index, which is enclosed as part of the row key. For geometry queries, the *Geometry Translator* module first applies the same Moore encoding algorithm to calculate scan ranges. Then, the *Multi-Scan Optimizer* computes the optimal read strategy such that the IO latency is minimized. Sections 3.1 and 3.2 present more details.

Pyro relies on the group-based replica placement policy in PyroDFS to guarantee data locality during region splits. To achieve that, each StoreFile is divided into multiple shards based on user-defined pre-split keys. Then, Pyro organizes DFS replicas of all shards into elaborately designed groups. Replicas in the same group are stored in the same physical server. After one or multiple splits, each daughter region is guaranteed to find at least one replica of all its region data within one group. To preserve data locality, Pyro just need to move the daughter region into the physical server hosting that group. The details of group-based replica placement are described in section 3.3.

Pyro makes three major contributions:

- Geometry Translation: Apart from previous solutions that build an index layer above HBase, Pyro internally implements efficient geometry translation algorithms based on Moore encoding. This design allows Pyro to optimize a geometry query by globally processing all its range scans together.
- Multi-Scan Optimization: After geometry translation, the multi-scan optimizer aggregately processes the generated range scans to minimize the response time of the geometry query. By using storage media performance profiles as inputs, the multi-scan optimizer employs a dynamic programming algorithm to calculate the optimal HBase blocks to fetch.
- Block Grouping: To deal with moving hotspots, Pyro relies on a novel data block grouping algorithm in the DFS layer to split a region quickly and efficiently, while preserving data locality benefits. Moreover, by treating meta block and data block differently, block grouping helps to improve Pyro's fault tolerance.

3 System Design

We first present the geometry translation and multi-scan optimization in Sections 3.1 and 3.2 respectively. These two solutions help to efficiently process geometry queries. Then, Section 3.3 describes how Pyro handles moving hotspots with the block grouping algorithm.

3.1 Geometry Translation

In order to store spatial-temporal data, Pyro needs to cast 2D coordinates (x, y) into the one-dimensional key space. A straightforward solution is to use a fixed number of bits to represent x, and y, and append x after y to form the spatial key. This leads to the *Strip*-encoding as shown in Figure 2 (a). Another solution is to use ZOrder-encoding [7] that interleaves the bits of x and y. An example is

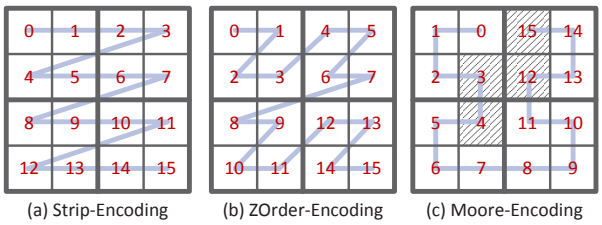


Figure 2: Spatial Encoding Algorithms of Resolution 2

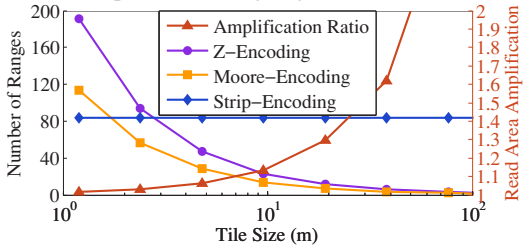
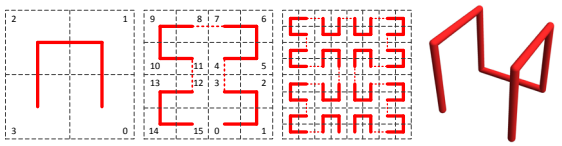


Figure 3: Translate Geometry to Key Ranges

illustrated in Figure 2 (b). These encoding algorithms divide the 2D space into $m \times m$ tiles, and index each tile with a unique ID. The tile is the spatial encoding unit as well as the unit of range scans. We define the resolution as $\log_2(m)$, which is the minimum number of bits required to encode the largest value of x and y.

In most cases, encoding algorithms inevitably break a two-dimensional geometry into multiple key ranges. Therefore, each geometry query may result in multiple range scans. Each range scan requires a few indexing, caching, and disk operations to process. Therefore, it is desired to keep the number of range scans low. We carry out experiments to evaluate the number of range scans that a geometry query may generate. The resolution ranges from 25 to 18 over the same set of randomly generated disk-shaped geometry queries with $100m$ radius in a $40,000,000m \times 40,000,000m$ area. The corresponding tile size ranges from 1.2m to 153m. Figure 3 shows the number of range scans generated by a single geometry query under different resolutions. It turns out that Strip-encoding and ZOrder-encoding translate a single disk geometry to a few tens of range scans when the tile size falls under 20m.

To reduce the number of range scans, we developed the Geometry Translator module. The module employs the *Moore*-Encoding algorithm which is inspired by the Moore curve from the space-filling curve family [15–18]. A simple example is shown in Figure 2 (c). A Moore curve can be developed up to any resolution. As shown in Figure 4 (a), resolutions 1 and 2 of Moore encoding are special cases. The curve of resolution 1 is called a unit component. In order to increase the resolution, the Moore curve expands each unit component according to a fixed strategy as shown in Figure 5. Results plotted in Figure 3 show that Moore-Encoding helps to reduce the number of range scans by 40% when compared to ZOrder-Encoding. Moore curves may generalize to higher dimensions [19], Figure 4 (b) depicts the simplest



(a) (b)

Figure 4: Moore Curve

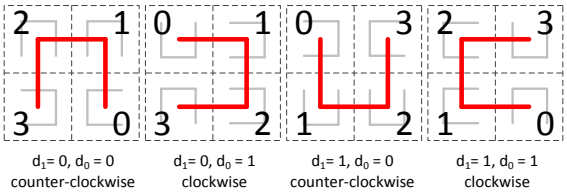


Figure 5: Moore Encoding Unit

3D Moore curve of resolution 1. Implementations of the Moore encoding algorithm are presented in Section 4.

3.2 Multi-Scan Optimization

The purpose of multi-scan optimization is to reduce read amplification. Below, we first describe the phenomenon of read amplification, and then we present our solution to this problem.

3.2.1 Read Amplification

When translating geometry queries, range scans are generated respecting tile boundaries at the given resolution. But, tile boundaries may not align with the geometry query boundary. In order to cover the entire geometry, data from a larger area is fetched. We call this phenomenon *Read Area Amplification*. Figure 3 plots the curve of read area amplification ratio, which is quantitatively defined as the total area of fetched tiles over the area of the geometry query. The curves show that, solely tuning the resolution cannot achieve both a small number of range scans and a low ratio of read area amplification. For example, as shown in Figure 3, restricting each geometry query to generate less than 10 scans forces Pyro to fetch data from a 22% larger area. On the other hand, limiting the area amplification ratio to less than 5% leads to more than 30 range scans per geometry query. The problem gets worse for larger geometries.

Moreover, encoding tiles are stored into fixed-size DB blocks on disks, whereas DB blocks ignore the boundaries of encoding tiles. An entire DB block has to be loaded even when there is only one requested key-value pair fallen in that DB Block, which we call the *Read Volume-Amplification*. Please notice that, DB blocks are different from DFS blocks. DB blocks are the minimum read/write units in PyroDB (similar to HBase). One DB block is usually only a few tens of KiloBytes. In contrast, a DFS block is the minimum replication unit in PyroDFS (similar to HDFS). DFS blocks are orders of magnitudes larger than DB blocks. For example, the default PyroDFS block size is 64MB, which is 1024 times larger than the default PyroDB block size.

Besides read area and volume amplifications, using a third-party indexing layer may also force the data store to unnecessarily visit a DB block multiple times, especially for high resolution queries. We call it the *Redundant Read Phenomenon*. Even though a DB block can be cached to avoid disk operations, the data store still needs to traverse DB block's data structure to fetch the requested key-value pairs. Therefore, Moore encoding algorithm alone is not enough to guarantee the efficiency.

For ease of presentation, we use the term *Read Amplification* to summarize the read area amplification, read volume amplification, and redundant read phenomena. Read amplification may force a geometry query to load a significant amount of unnecessary data as well as visiting the same DB block multiple times, leading to a much longer response time. In the next section, we present techniques to minimize the penalty of read amplification.

3.2.2 An Adaptive Aggregation Algorithm

According to Figure 3, increasing the resolution helps to alleviate read area amplification. Using smaller DB block sizes reduces read volume amplification. However, these changes require Pyro to fetch significantly more DB blocks, pushing disk IO to become a throughput bottleneck. In order to minimize the response time, Pyro optimizes all range scans of the same geometry query aggregately, such that multiple DB blocks can be fetched within fewer disk read operations. There are several reasons for considering IO optimizations in the DB layer rather than relying on asynchronous IO scheduling in the DFS layer or the OS layer. First, issuing a DFS read request is not free. As a geometry query may potentially translate into a large number of read operations, maintaining those reads alone elicits extra overhead in all three layers. Second, performance of existing IO optimizations in lower layers depend on the timing and ordering of request submissions. Enforcing the perfect request submission ordering in the Geometry Translator is not any cheaper than directly performing the IO optimization in PyroDB. Third, as PyroDB servers have the global knowledge about all p-reads from the same geometry request, it is the natural place to implement IO optimizations.

Pyro needs to elaborately tune the trade-off between unnecessarily reading more DB blocks and issuing more disk seeks. Figure 6 shows the profiling results of Hadoop-2.4.1 position read (p-read) performance on a 7,200RPM Seagate hard drive and a Samsung SM0256F Solid State Drive respectively. In the experiment, we load a 20GB file into the HDFS, and measure the latency of p-read operations of varies sizes at random offsets. The disk seek delay dominates the p-read response time when reading less than 1MB data. When the size of p-read surpasses 1MB, the data transmission delay starts to make a difference. A naïve solution calculates the disk seek delay and the per-block transmission delay, and

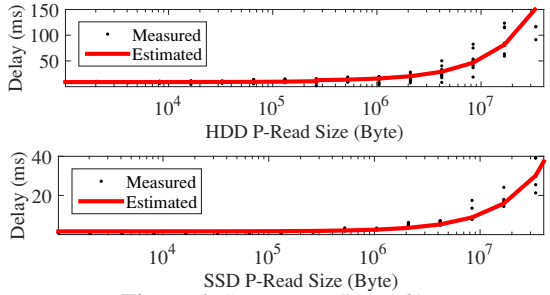


Figure 6: Storage Media Profile

directly compares whether reading the next unnecessary block helps to reduce response time. However, the system may run on different data storage media, including hard disk drives, solid state drives, or even remote cloud drives. There is no guarantee that all media share the same performance profile. Such explicit seek delay and transmission delay may not even exist.

In order to allow the optimized range scan aggregation to work for a broader scenarios, we propose the Adaptive Aggregation Algorithm (A^3). A^3 uses the p-read profiling result to estimate delay of p-read operations. The profiling result contains the p-read response time of various sizes. A^3 applies interpolation to fill in gaps between profiled p-read sizes. This design allows the A^3 algorithm to work for various storage media.

Before diving into algorithm details, we present the abstraction of the block aggregation problem. Suppose a geometry query hits shaded tiles (3, 4, 12, 15) in Fig 2 (c). For the sake of simplicity, assume that DB blocks align perfectly with encoding tiles, one block per tile. Figure 7 shows the block layout in the StoreFile. A^3 needs to determine what block ranges to fetch in order to cover all requested blocks, such that the response time of the geometry query is minimized. In this example, let us further assume each block is 64KB. According to the profiling result shown in Figure 6, reading one block takes about 9 ms, four blocks takes 14 ms, while reading thirteen blocks takes 20 ms. Therefore, the optimal solution reads blocks 3-15 using one p-read operation.

A^3 works as follows. Suppose a geometry query translates to a set $\mathbf{Q}$ of range scans. Block indices help to convert those range scans into another set $\mathbf{B}'$ of blocks, sorted in the ascending order of their offsets. By removing all cached blocks from $\mathbf{B}'$, we get set $\mathbf{B}$ of n requested but not cached blocks. Define $\mathbf{S}[i]$ as the estimated minimum delay of loading the first i blocks. Then, the problem is to solve $\mathbf{S}[n]$. For any optimal solution, there must exist a k, such that blocks k to n are fetched using a single p-read operation. In other words, $\mathbf{S}[n] = \mathbf{S}[k-1] + \textsc{Estimate}(k,n)$, where $\textsc{Estimate}(k,n)$ estimates the delay of fetching blocks from k to n together based on the profiling result. Therefore, starting from $\mathbf{S}[0]$, A^3 calculates $\mathbf{S}[i]$ as $\min\{\mathbf{S}[k-1] + \textsc{Estimate}(k,i) | 1 \le k \le i\}$. The pseudo code of A^3 is presented in Algorithm 1.

Algorithm 1: A^3 Algorithm

```
   Input: blocks to fetch sorted by offset B
   Output: block ranges to fetch R
1  S ← an array of size |B|; initialize to ∞
2  P ← an array of size |B|; S[0] ← 0
3  for i ← 1 ∼ |B| do
4      for j ← 0 ∼ i − 1 do
5          k = i − j;   s ←ESTIMATE(k, i) +S[k − 1]
6          if s < S[i] then
7              S[i] ← s;   P[i] ← k
8  i ← |B|; R ← ∅
9  while i > 0 do
10     R ← R ∪ (P[i], i);   i ← P[i] − 1
11 return R
```

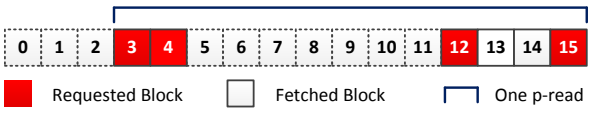


Figure 7: Block Layout in a StoreFile

In Algorithm A^3, the nested loop between line $3-7$ leads to $O(|\mathbf{B}|^2)$ computational complexity. If $\mathbf{B}$ is large, the quadratic computational complexity explosion can be easily mitigated by setting an upper bound on the position read size. For example, for the hard drive profiled in Figure 6, fetching 10^7 bytes result in about the same delay as fetching 5×10^6 bytes twice. Therefore, there is no need to issue position read larger than 5×10^6 bytes. If block size is set to $64KB$, the variable j on the 5^{th} line in Algorithm 1 only needs to loop from 0 to 76, resulting in linear computational complexity.

3.3 Block Grouping

Moore encoding concentrates range scans of one geometry query into fewer servers. This may lead to performance degradation when spatial-temporal hotspots exist. To handle moving hotspots, a region needs to gracefully split itself to multiple daughters to make use of resources on multiple physical servers. Later, those daughter regions may merge back after their workloads shrink.

In HBase, the split operation creates two daughter regions on the same physical server, each owning reference files pointing back to StoreFiles of their parent region. Daughter regions are later moved onto other servers during the next cluster balance operation. Using reference files on one hand helps to keep the split operation light, but on the other hand prevents daughter regions from taking advantage of data locality benefits. Because, after leaving the parent region's server, the two daughter regions can no longer find their region data in their local disks until daughters' reference files are materialized. HBase materializes reference files during the next major compaction, which executes at a very low frequency (*e.g.*, once a day). Forcing earlier materialization does not solve the problem. It could introduce even more overhead to the already-overwhelmed region, as materializa-

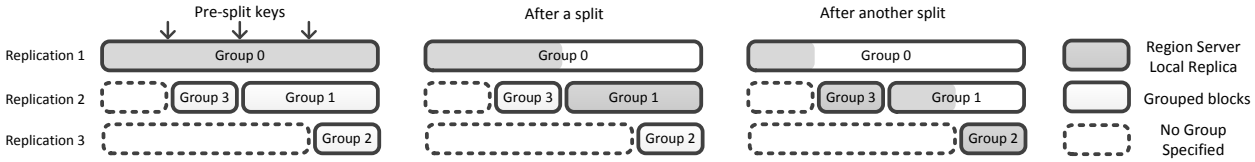


Figure 8: Split Example

tion itself is a heavy operation.

An ideal solution should keep both split and materialization operations light, allowing the system to react quickly when a hotspot emerges. Below, we present our design to achieve such ideal behaviors.

3.3.1 Group Based Replica Placement

Same as HBase, Pyro suggests users to perform pre-split based on expected data distribution to gain initial load balancing among region servers. Pyro relies on the expected data distribution to create more splitting keys for potential future splits. Split keys divide StoreFiles into shards, and help to organize DFS block replicas into replica groups. PyroDFS guarantees that DFS blocks respect predefined split keys. To achieve that, PyroDFS stops writing into the current DFS block and start a new one as soon as it reaches a predefined split key. This design relies on the assumption that, although moving hotspots may emerge in spatial-temporal applications, the long-round popularity of different geographic regions changes slowly. Results presented in evaluation Section 5.1 confirm the validity of this assumption.

Assume blocks are replicated r times and there are $2^{r-1}-1$ predefined split keys within a given region. Split keys divide the region key space into 2^{r-1} shards, resulting in $r \cdot 2^{r-1}$ shard replicas. Group 0 contains one replica from all shards. Other groups can be constructed following a recursive procedure:

1 Let Ψ be the set of all shards. If Ψ contains only one shard, stop. Otherwise, use the median split key κ in Ψ to divide all shards into two sets A and B. Keys of all shards in A are larger than κ, while keys of all shards in B are smaller than κ. Perform step 2, and then perform step 3.

2 Create a new group to contain one replica from all shards in set A. Then, let $\Psi \leftarrow A$, and recursively apply step 1.

3 Let $\Psi \leftarrow B$, and then recursively apply step 1.

Replicas in the same group are stored in the same physical server, whereas different groups of the same region are placed into different physical servers. According to the construction procedure, group 1 starts from the median split key, covering the bottom half of the key space (*i.e.*, 2^{r-2} shards). Group 1 allows half of the regions workload to be moved from group 0's server to group 1's server without sacrificing data locality. Figure 8 demonstrates an example of $r = 3$. PyroDFS is compatible with normal HDFS workload whose replicas can be simply set as no group specified. Section 3.3.2 explains why group 1 and 2 are placed at the end rather than in the beginning of the StoreFile.

Figure 8 also shows how Pyro makes use of DFS block replicas. The shaded area indicates which replica serves workloads falling in that key range. In the beginning, there is only one region server. Replicas in group 0 take care of all workloads. As all replicas in group 0 are stored locally in the region's physical server, data locality is preserved. After one split, the daughter region with smaller keys stays in the same physical server, hence still enjoys data locality. Another daughter region moves to the physical server that hosts replica group 1, which is also able to serve this daughter region using local data. Subsequent splits are carried out under the same fashion.

To distinguish from the original split operation in HBase, we call the above actions the *soft* split operation. Soft splits are designed to mitigate moving hotspots. Daughter regions created by soft splits eventually merge back to form their parent regions. The efficiency of the merge operation is not a concern as it can be performed after the hotspot moves out of that region. Please notice that the original split operation, which we call the *hard* split, is still needed when a region grows too large to fit in one physical server. As this paper focuses on geometry query and moving hotspots, all splits in the following sections refer to soft splits.

3.3.2 Fault Tolerance

As a persistent data store, Pyro needs to preserve high data availability. The block grouping algorithm presented in the previous section affects DFS replica placement schemes, which in turn affects Pyro's fault tolerance properties. In this section, we show that the block grouping algorithm allows Pyro to achieve higher data availability compared to the default random replica placement policy in HDFS.

Pyro inherits the same HFile format [3] from HBase to store key-value pairs. According to HFile Format, meta blocks are stored at the end of the file. Losing any DFS block of the meta may leave the entire HFile unavailable, whereas the availability of key-value DFS blocks are not affected by the availability of other key-value DFS blocks. This property makes the last shard of the file more important than all preceding shards. Therefore, we choose two different objectives for their fault tolerance design.

- Meta shard: Minimize the probability of losing any DFS block.
- Key-value shard: Minimize the expectation of the number of unavailable DFS blocks.

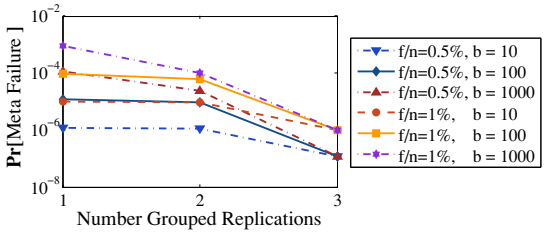


Figure 9: Unavailability Probability

Assume there are n servers in the cluster, and f nodes are unavailable during a cluster failure event. For a given shard, assume it contains b blocks, and replicates r times, where g out of r replications are grouped. PyroDFS randomly distributes the grouped g replications into g physical servers. The remaining $(r-g)b$ block replicas are randomly and exclusively distributed in the cluster. If the meta fails, it must be the case that the g servers hosting the g grouped replications all fail (*i.e.*, $\binom{f}{g}/\binom{n}{g}$), and at least one block in each $r-g$ ungrouped replications fails. Hence, the probability of meta failure is

$$\mathbf{Pr}\left[\text{meta failure}\right] = \frac{\binom{f}{g}}{\binom{n}{g}}\left(1-\left(1-\frac{\binom{f-g}{r-g}}{\binom{n-g}{r-g}}\right)^{b}\right). \quad (1)$$

Figure 9 plots how the number of grouped replications g affects the failure probability. In this experiment, n and r are set to 10,000, and 3 respectively. According to [20–22], after some power outage, 0.5%-1% of the nodes fail to reboot. Hence, we vary f to be 50, and 100. The results show that the meta failure probability decreases when g increases. Pyro sets g to the maximum value for the meta shard, therefore achieves higher fault tolerance compared to default HDFS where g equals 1.

For key-value shards, transient and small-scale failures are tolerable, as they do not affect most queries. It is more important to minimize the scale of the failure (*i.e.*, the number of unavailable DB blocks). The expected failure scale is,

$$\mathbf{E}\left[\text{failure scale}|\text{failure occurs}\right] = \frac{b\binom{f-g}{r-g}}{\binom{n-g}{r-g}}. \quad (2)$$

The failure scale decreases with the increase of grouped replication number g. Therefore, placing replica groups 1 and 3 at the end of the StoreFile minimizes both the meta shard failure probability and the failure scale of key-value shards.

4 Implementation

PyroDFS and PyroDB are implemented based on HDFS-2.4.1 and HBase-0.99 respectively.

4.1 Moore Encoding

As previously shown in Figure 4 and Figure 5, each unit of Moore curve can be uniquely defined by the combination of its orientation (north, east, south, and west) and its rotation (clockwise, counter-clockwise). Encode the orientation with 2 bits, $d1$ and $d0$, such that 00 denotes

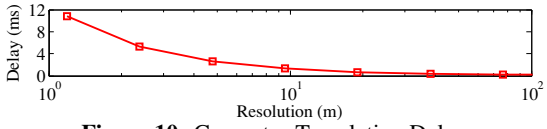


Figure 10: Geometry Translation Delay

north, 01 east, 10 south, and 11 west. With more careful observations, it can be seen that the rotation of a Moore curve component unit completely depends on its orientation. Starting from the direction shown in Figure 4 (a), the encodings in east and west oriented units rotate clockwise, and others rotate counter-clockwise. With a given integer coordinate (x,y), let x_k and y_k denote the k^{th} lowest bits of x and y in the binary presentation. Let $d_{k,1}d_{k,0}$ be the orientation of the component unit defined by the highest $r-k-1$ bits in x, and y. Then, the orientation $d_{k-1,1}d_{k-1,0}$ can be determined based on $d_{k,1}$, $d_{k,0}$, x_k, and y_k [15–18].

$$\begin{aligned}
d_{k-1,0} &= \bar{d}_{k,1}\bar{d}_{k,0}\bar{y}_k \mid \bar{d}_{k,1}d_{k,0}x_k \\
&\mid d_{k,1}\bar{d}_{k,0}y_k \mid d_{k,1}d_{k,0}\bar{x}_k && (3)\\
&= \bar{d}_{k,0}\left(d_{k,1}\oplus\bar{y}_k\right) \mid d_{k,0}\left(d_{k,1}\oplus x\right) && (4)\\
d_{k-1,1} &= \bar{d}_{k,1}\bar{d}_{k,0}x_k\bar{y}_k \mid \bar{d}_{k,1}d_{k,0}\bar{x}_k y_k \\
&\mid d_{k,1}\bar{d}_{k,0}\overline{\bar{x}_k y_k} \mid d_{k,1}d_{k,0}\overline{x_k\bar{y}_k} && (5)\\
&= d_{k,1}\left(\bar{x}_k\oplus y_k\right) \mid \left(x_k\oplus y_k\right)\left(d_0\oplus x_k\right) && (6)
\end{aligned}$$

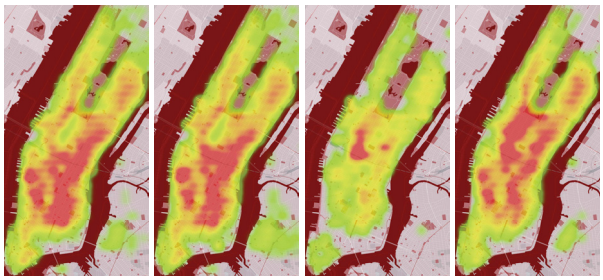
The formula considers all situations where $d_{k-1,0}$ and $d_{k-1,1}$ should equal to 1, and uses a logic *or* to connect them all. For example, the term $\bar{d}_{k,1}\bar{d}_{k,0}\bar{y}_k$ states that when the previous orientation is north ($\bar{d}_{k,1}\bar{d}_{k,0}$), the current unit faces east or west ($d_{k-1,0}=1$) if and only if $y_k=0$. The same technique can be applied to determine the final Moore encoding index ω.

$$\begin{aligned}
\omega_{2k+1} &= \bar{d}_{k,1}\bar{d}_{k,0}\bar{x}_k \mid \bar{d}_{k,1}d_{k,0}\bar{y}_k \\
&\mid d_{k,1}\bar{d}_{k,0}x_k \mid d_{k,1}d_{k,0}y_k && (7)\\
&= \bar{d}_{k,0}\left(d_{k,1}\oplus\bar{x}_k\right)+d_{k,0}\left(d_{k,1}\oplus\bar{y}_k\right) && (8)\\
\omega_{2k} &= \bar{x}_k\oplus y_k && (9)
\end{aligned}$$

Then, each geometry can be translated into range scans using a quad tree. Each level in the quad tree corresponds to a resolution level. Each node in the tree represents a tile, which is further divided into four smaller tiles in the next level. The translating algorithm only traverses deeper if the geometry query partially overlaps with that area. If an area is fully covered by the geometry, there is no need to go further downwards. Figure 10 shows the delay of translating a $5km\times 5km$ square geometry. The delay stays below 11ms even using the finest resolution.

4.2 Multi-Scan Optimization

After converting a geometry query into range scans, the multi-scan optimizer needs two more pieces of information to minimize the response time: 1) storage media performance profiles, and 2) the mapping from key ranges to DB blocks. For the former one, an administrator may specify an HDFS path under the property name *hbase.profile.storage* in the hbase-site.xml configuration



(a) (b) (c) (d)

Figure 11: Manhattan Taxi Pick-up/Drop-off Hotspots

file. This path should point to a file containing multiple lines of (p-read size, p-read delay) items, indicating the storage media performance profile result. Depending on storage media types in physical servers, the administrator may set the property *hbase.profile.storage* to different values for different HRegions. The file will be loaded during HRegion initialization phase. For the latter one, HBase internally keeps indices of DB blocks. Therefore, Pyro can easily translate a range scan into a serious of block starting offsets and block sizes. Then, those information will be provided as inputs for the A^3 algorithm.

4.3 Block Grouping

Distributed file systems usually keep replica placement policies as an internal logic, maintaining a clean separation between the DFS layer and higher layer applications. This design, however, prevents Pyro from exploring opportunities to make use of DFS data replications. Pyro carefully breaks this barrier by exposing a minimum amount of control knobs to higher layer applications. Through these APIs, applications may provide *replica group* information when writing data into DFS. It is important to choose the right set of APIs such that PyroDFS applications do not need to reveal too much about details in the DFS layer. At the same time, applications are able to fully make use of data locality benefits of all block replicas.

In our design, PyroDFS exposes two families of APIs which help to alter its internal behavior.

- *Sealing a DFS Block:* PyroDB may force PyroDFS to seal the current DFS block and start writing into a new DFS block, even if the current DFS block has not reached its size limit yet. This API is useful because DFS block boundaries may not respect splitting keys, especially when there are many StoreFiles in a region and the sizes of StoreFiles are about the same order of magnitude of the DFS block size. The *seal* API family will help StoreFiles to achieve full data locality after splits.
- *Grouping Replicas:* PyroDB may specify *replica namespace* and *replica groups* when calling the *write* API in PyroDFS. This usually happens during MemStore flushes and StoreFile compactions. Under the same namespace, replicas in the same replica group will be placed into the same physical server, and replicas in different groups will be placed into different physical servers. If there are not enough physical servers or disk spaces, PyroDFS works in a best effort manner. The mapping from the replica group to the physical server and corresponding failure recovery is handled within PyroDFS. PyroDB may retrieve a physical server information of a given replica group using *grouping* APIs, which allows PyroDB to make use of data locality benefits.

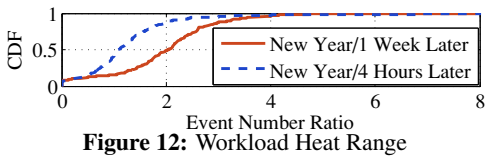


Figure 12: Workload Heat Range

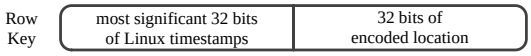


Figure 13: Row Key

5 Evaluation

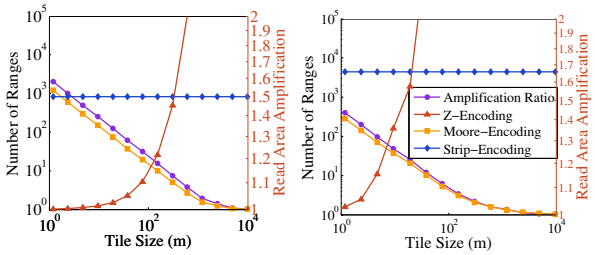
Evaluations use NYC taxi dataset [9, 10] that contains GPS pickup/dropoff location information of 697,622,444 trips from 2010 to 2013. The experiments run on a cluster of 80 Dell servers (40 Dell PowerEdge R620 servers and 40 Dell PowerEdge R610 servers) [23–32]. The HDFS cluster consists of 1 master node and 30 datanodes. The HBase server contains 1 master node, 3 zookeeper [33] nodes, and 30 region servers. Region servers are co-located with data nodes. Remaining nodes follow a central controller to generate geometry queries and log response times, which we call Remote User Emulators (RUE).

We first briefly analyze the NYC taxi dataset. Then, Sections 5.2, 5.3, and 5.4 evaluate the performance improvements contributed by Geometry Translator, Multi-Scan Optimizer, and Group-based Replica Placement respectively. Finally, in Section 5.5, we measure the overall response time and throughput of Pyro.

5.1 NYC Taxi Data Set Analysis

Moving hotspot is an important phenomenon in spatial-temporal data. Figure 11 (a) and (b) illustrate the heat maps of taxi pick-up and drop-off events in the Manhattan area during a 4 hour time slot starting from 8:00PM on December 31, 2010 and December 31, 2012 respectively. The comparison shows that the trip distribution during the same festival does not change much over the years. Figure 11 (c) plots the heat map of the morning (6:00AM-10:00AM) on January 1st, 2013, which drastically differs from the heat map shown in Figure 11 (b). Figure 11 (d) illustrates the trip distribution from 8:00PM to 12:00AM on July 4th, 2013, which also considerably differs from that of the New Year Eve in the same year.

Figures 11 (a)-(d) demonstrate the distribution of spatial-temporal hotspots. It is important to understand by how much hotspots cause event count to increase



(a) radius = $1000m$ (b) $50m \times 628m$

Figure 14: Reducing the Number of Range Scans

in a region. We measure the increase as the ratio, $\frac{\text{event count during peak hours}}{\text{event count during normal hours}}$. The CDF on 16X16 Manhattan area is shown in Figure 12. Although hotspots move over time, the event count of a region changes within a reasonably small range. During New Year midnight, popularity of more than 97% regions grow within four folds.

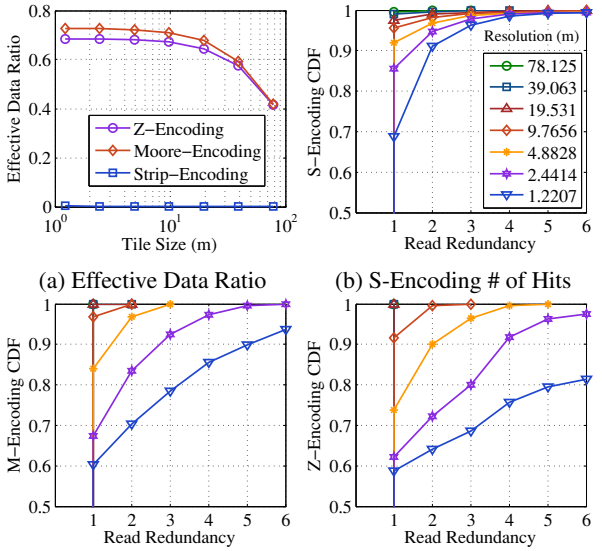
When loading the data into HBase, both spatial and temporal information contribute to the row key. The encoding algorithm translates the 2D location information of an event into a 32-bit spatial-key, which acts as the suffix of the row key. Then, the temporal strings are parsed to Linux 64-bit timestamps. We use the most significant 32 bits as the temporal-key. Each temporal key represents roughly a 50-day time range. Finally, as shown in Figure 13, the temporal-key is concatenated in front of the spatial key to form the complete row key.

5.2 Moore Encoding

Figure 14 shows how much Moore encoding helps to reduce the number of range scans at different resolutions when translating geometry queries in a $40,000,000m \times 40,000,000m$ area. Figures 14 (a) and (b) uses disk geometry and rectangle geometries respectively. The two figures share the same legend. For disk geometries, Moore encoding generates 45% fewer range scans when compared to ZOrder-encoding. When a long rectangle is in use, Moore encoding helps to reduce the number of range scans by 30%.

To quantify the read volume amplification, we encode the dataset coordinates with Moore encoding algorithm using the highest resolution shown in Figure 3, and populate the data using 64KB DB Blocks. Then, the experiment issues $1Km \times 1Km$ rectangle geometries. Figure 15 (a) shows the ratio of fetched key-value pairs volume over the total volume of accessed DB Blocks, which is the inverse of read volume amplification. As the Strip-encoding results in very high read volume amplification, using the inverse helps to limit the result in interval [0, 1]. Therefore, readers can easily distinguish the difference between Moore-encoding and ZOrder-encoding. We call the inverse metric the effective data ratio. As Moore encoding concentrates a geometry query into fewer range scans, and hence fewer range boundaries, it also achieves higher effective data ratio.

Figures 15 (b)-(d) plot the CDFs of redundant read



(a) Effective Data Ratio (b) S-Encoding # of Hits

(c) Z-Encoding # of Hits (d) M-Encoding # of Hits

Figure 15: Read Amplification Phenomenon

counts when processing the same geometry query.It is clear that the number of redundant reads increases when using higher resolutions. Another observation is that, Moore-encoding leads to large read redundancy. Thanks to the multi-scan optimization design, this will not be a problem, as all redundant reads will be accomplished within a single DB block traverse operation.

5.3 Multi-Scan Optimization

In order to measure how A^3 algorithm works, we load data from the NYC taxi cab dataset using Moore encoding algorithm, and force all StoreFiles of the same store to be compacted into one single StoreFile. Then, the RUE generates $1Km \times 1Km$ rectangle geometry queries with the query resolution set to 13. We measure the internal delay of loading requested DB blocks individually versus aggregately.

The evaluation results are presented in Figure 16. The curves convey a few interesting observations. Let us look at the A^3 curve first. In general, this curve rises as the block size increases, which agrees with our intuition as larger blocks lead to more severe *read volume amplification*. The minimum response time is achieved at 8KB. Because the minimum data unit of the disk under test is 4KB, further decreasing block size does not help any more. On the computation side, using smaller block size results in larger input scale for the A^3 algorithm. That explains why the response time below 8KB slightly goes up as the block size decreases. The "*individually*" curve monotonically decreases when the block size grows from 1KB to 100KB. It is because increasing block size significantly reduces the number of disk seeks when the block is small. When the block size reaches between 128KB and 4MB, two facts become true: 1) key-value pairs hit by a geometry query tend to concentrate in less blocks; 2) data transmission time starts to make impacts. The

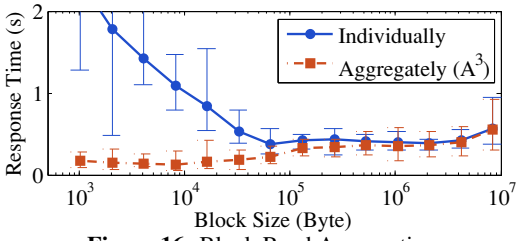


Figure 16: Block Read Aggregation

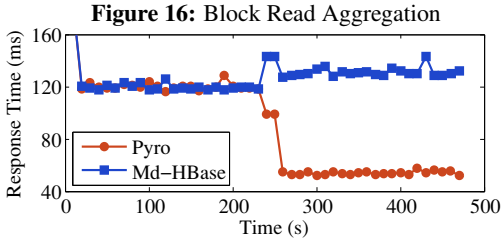


Figure 17: Response Time at Splitting Event

benefits of reducing the number of disk seeks and the penalties of loading DB blocks start to cancel each other, leading to a flat curve. After 4MB, the data transmission delay dominates the response time, and the curve rises again. Comparing the nadirs of the two curves concludes that A^3 helps to reduce the response time by at least 3X.

5.4 Soft Region Split

To measure the performance of *soft* splitting, this experiment uses normal scan queries instead of geometry queries, excluding the benefits of Moore encoding and multi-scan optimization. A table is created for the NYC's taxi data, which initially splits into 4 regions. Each region is assigned to a dedicated server. The HBASE_HEAPSIZE parameter is set to 1GB, and the MemStore flush size is set to 256MB. Automatic region split is disabled to allow us to manually control the timing of splits. Twelve RUE servers generate random-sized small scan queries.

Figure 17 shows the result. The split occurs at the 240^{th} second. After the split operation, HBase suffers from even longer response time. It is because daughter region B does not have its region data in its own physical server, and has to fetch data from remote servers, including the one hosting daughter region A. When the group based replication is enabled, both daughter regions read data from local disks, reducing half of the pressure on disk, cpu, and network resources.

5.5 Response Time and Throughput

We measure the overall response time and throughput improved by Pyro compared to the state-of-the-art solution MD-HBase. Experiments submit rectangle geometry queries of size $1km \times 1km$ and $100m \times 100m$ to Pyro and MD-HBase. The request resolutions are set to 13 and 15 respectively for two types of rectangles. The block sizes vary from 8KB to 512KB. When using MD-HBase, the remote query emulator initiates all scan queries sequentially using one thread. This configuration tries to make the experiment fair, as Pyro uses a single thread to

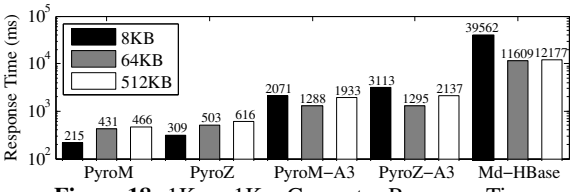


Figure 18: 1Km×1Km Geometry Response Time

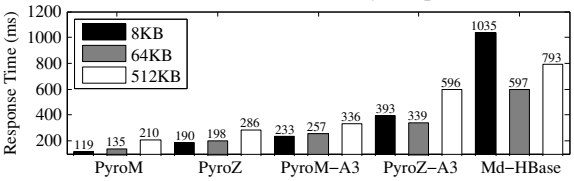


Figure 19: 100m×100m Geometry Response Time

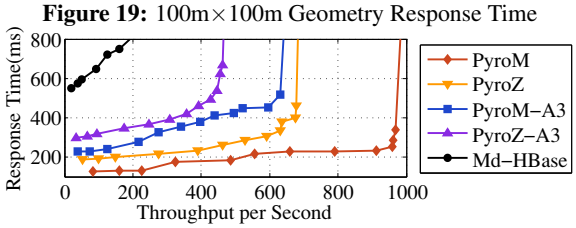


Figure 20: 100m×100m Geometry Throughput

answer each geometry query. Besides, experiments also show how Pyro performs when using ZOrder-encoding or/and A^3 algorithm. Figures 18 and 19 plot experiment results. The legend on the upper-left corner shows the mapping from colors to block sizes. PyroM and PyroZ represent Pyro using Moore- and ZOrder- encoding respectively. PyroM-A^3 and PyroZ-A^3 correspond to the cases with the A^3 algorithm disabled.

When using PyroM and PyroZ, the response times grow with the increase of block size regardless of whether the rectangle geometry is large or small. It is because larger blocks weaken the benefits of block aggregation and force PyroM and PyroZ to read more data from disk. After disabling A^3, the response time rises by 6X for $1km \times 1km$ rectangles, and 2X for $100m \times 100m$ rectangles. MD-HBase achieves the shortest response time when using 64KB DB blocks, which is 60X larger compared to PyroM and PyroZ when handling $1km \times 1km$ rectangle geometries. Reducing the rectangle size to $100m \times 100m$ shrinks the gap to 5X. An interesting phenomenon is that using 512KB DB blocks only increases the response time by 5% compared to using 64KB DB blocks, when the request resolution is set to 13. However, the gap jumps to 33% if the resolution is set to 15. The reason is that, higher resolution leads to more and smaller range scans. In this case, multiple range scans are more likely to hit the same DB block multiple times. According to HFile format, key-value pairs are chained together as a linked-list in each DB block. HBase has to traverse the chain from the very beginning to locate the starting key-value pair for every range scan. Therefore, larger DB block size results in more overhead on iterating through the key-value chain in each DB block.

Figure 20 shows the throughput evaluation results of

the entire cluster. Pyro regions are initially partitioned based on the average pick up/drop off event location distribution over the year of 2013. Literature [9] presents more analysis and visualizations of the dataset. During the evaluation, each RUE server maintains a pool of emulated users who submit randomly located $100m \times 100m$ rectangle geometry queries. The reason of using small geometries in this experiment is that MD-HBase results in excessively long delays when handling even a single large geometry. The distribution of the rectangle geometry queries follows the heat map from 8:00PM to 11:59PM on December 31, 2013. The configuration mimics the situation where an application only knows the long-term data distribution, and is unable to predict hotspot bursts. When setting 600ms to be the maximum tolerable response time, Pyro outperforms MD-HBase by 10X.

6 Related Work

As the volume of spatial-temporal data is growing at an unprecedented rate, pursing a scalable solution for storing spatial-temporal data has become a common goal shared by researchers from both the distributed system community and the database community. Advances on this path will benefit a great amount of spatial-temporal applications and analytic systems.

Traditional relational databases understand high dimensional data well [17, 18, 34, 35] due to extensively studied indexing techniques, such as *R*-Tree [36], *Kd*-Tree [37], *UB*-Tree [38, 39], etc. Therefore, researchers seek approaches to improve the scalability. Wang *et al.* [40] construct a global index and local indices using Content Addressable Network [41]. The space is partitioned into smaller subspaces. Each subspace is handled by a local storage. The global index manages subspaces, and local indices manage data points in their own subspaces. Zhang *et al.* [42] propose a similar architecture using *R*-tree as global index and *Kd*-tree as local indices.

From another direction, distributed system researchers push scalable NoSQL stores [3–6, 13, 43–45] to better understand high dimensional data. Distributed key-value stores can be categorized into two classes. One class uses random partition to organize keys. Such systems include cassandra [4], DynamoDB [5], etc. Due to the randomness on key distribution, these systems are immune to dynamic hotspots concentrated in a small key range. However, spatial-temporal data applications and analytic systems usually issue geometry queries, which translate to range scans. Random partitioning cannot handle range scans efficiently, as it cannot extract all keys within a range with only the range boundaries. Consequently, each range scan needs to query all servers. Other systems, such as BigTable [13], HBase [3], couchDB [46], use ordered partitioning algorithms. In this case, the primary key space is partitioned into regions. The benefits are clear. As data associated with a continuous primary key range are also stored consecutively, sorted partitioning helps to efficiently locate the servers that host the requested key range.

The benefits of ordered partitioning encouraged researchers to mount spatial-temporal application onto HBase. Md-HBase [7] builds an index layer on top of HBase. The index layer encodes spatial information of a data point into a bit series using ZOrder-encoding. Then, a row using that bit series as key is inserted into HBase. The ST-HBase [8] develops a similar technique. However when serving geometry queries, the index layer inevitably translates each geometry query into multiple range scans, and prevents data store from aggregately minimizing the response time.

As summarized above, existing solutions either organize multiple relational databases together using some global index, or build a separate index layer above some general purpose distributed data stores. This paper, however, takes a different path by designing and implementing a holistic solution that is specifically tailored for spatial-temporal data.

7 Conclusion

In this paper, we present the motivation, design, implementation, and evaluation of Pyro. Pyro tailors HDFS and HBase for high resolution spatial-temporal geometry queries. In the DB layer, Pyro employs Moore encoding and multi-scan optimization to efficiently handle geometry queries. In the DFS layer, group-based replica placement policy helps Pyro to preserve data locality benefits during hotspots dynamics. The evaluation shows that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries and improves the throughput by 10X on $100m \times 100m$ rectangle geometries compared to the state-of-the-art solution.

Acknowledgement

We are grateful to Professor Dirk Grunwald and reviewers for their invaluable comments during the revision process of this paper. We would also like to thank Professor Dan Work for sharing with us the NYC taxi dataset. This research was sponsored in part by the National Science Foundation under grants CNS 13-20209, CNS 13-02563, CNS 10-35736 and CNS 09-58314, the Army Research Laboratory, and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] B. Sanou, "The world in 2013: Ict facts and figures," in *International Communication Union, United Nations*, 2013.

[2] S. Steiniger and E. Bocher, "An overview on current free and open source desktop gis developments," *International Journal of Geographical Information Science*, vol. 23, no. 10, pp. 1345–1370.

[3] L. George, *HBase: The Definitive Guide*. O'Reilly, 2011.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP*, 2007.

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *USENIX ATC*, 2013.

[7] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *IEEE International Conference on Mobile Data Management*, 2011.

[8] Y. Ma, Y. Zhang, and X. Meng, "St-hbase: A scalable data management system for massive geo-tagged objects," in *International Conference on Web-Age Information Management*, 2013.

[9] B. Donovan and D. B. Work, "Using coarse gps data to quantify city-scale transportation system resilience to extreme events," *Transportation Research Board 94th Annual Meeting*, 2014.

[10] New York City Taxi & Limousine Commission (NYCT&L), "Nyc taxi dataset 2010-2013," https://publish.illinois.edu/dbwork/open-data/, 2015.

[11] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SOSP*, 2003.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *USENIX OSDI*, 2006.

[14] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *USENIX FAST*, 2014.

[15] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer Publishing Company, Incorporated, 2012.

[16] J. Lawder, "The application of space-flling curves to the storage and retrieval of multi-dimensional data," in *Ph.D. Thesis*, 2000.

[17] K.-L. Chung, Y.-L. Huang, and Y.-W. Liu, "Efficient algorithms for coding hilbert curve of arbitrary-sized image and application to window query," *Inf. Sci.*, vol. 177, no. 10, pp. 2130–2151.

[18] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Halevy, "Hyper-local, directions-based ranking of places," *Proc. VLDB Endow.*, vol. 4, no. 5, pp. 290–301.

[19] R. Dickau, "Hilbert and moore 3d fractal curves," http://demonstrations.wolfram.com/HilbertAndMoore3DFractalCurves/, 2015.

[20] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *USENIX ATC*, 2013.

[21] R. J. Chansler, "Data availability and durability with the hadoop distributed file system," in *The USENIX Magzine*, 2012.

[22] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *USENIX OSDI*, 2010.

[23] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. F. Abdelzaher, "Proteus: Power proportional memory cache cluster in data centers," in *IEEE ICDCS*, 2013.

[24] S. Li, S. Wang, T. Abdelzaher, M. Kihl, and A. Robertsson, "Temperature aware power allocation: An optimization framework and case studies," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 3, pp. 117 – 127, 2012.

[25] S. Li, S. Hu, and T. F. Abdelzaher, "The packing server for real-time scheduling of mapreduce workflows," in *IEEE RTAS*, 2015.

[26] S. Li, T. F. Abdelzaher, and M. Yuan, "TAPA: temperature aware power allocation in data center with map-reduce," in *IEEE International Green Computing Conference and Workshops*, 2011.

[27] S. Li, H. Le, N. Pham, J. Heo, and T. Abdelzaher, "Joint optimization of computing and cooling energy: Analytic model and a machine room case study," in *IEEE ICDCS*, 2012.

[28] S. Li, S. Hu, S. Wang, L. Su, T. F. Abdelzaher, I. Gupta, and R. Pace, "WOHA: deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *IEEE ICDCS*, 2014.

[29] M. M. H. Khan, J. Heo, S. Li, and T. F. Abdelzaher, "Understanding vicious cycles in server clusters," in *IEEE ICDCS*, 2011.

[30] S. Li, S. Hu, S. Wang, S. Gu, C. Pan, and T. F. Abdelzaher, "Wattvalet: Heterogenous energy storage management in data centers for improved power capping," in *USENIX ICAC*, 2014.

[31] S. Li, L. Su, Y. Suleimenov, H. Liu, T. F. Abdelzaher, and G. Chen, "Centaur: Dynamic message dissemination over online social networks," in *IEEE ICCCN*, 2014.

[32] CyPhy Research Group, "UIUC Green Data Center," http://greendatacenters.web.engr.illinois.edu/index.html, 2015.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, 2010.

[34] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," *SIGMOD Rec.*, vol. 30, no. 1, pp. 19–24.

[35] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *VLDB*, 2007.

[36] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984.

[37] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *IEEE Symposium on Interactive Ray Tracing*, 2006.

[38] R. Bayer, "The universal b-tree for multidimensional indexing: General concepts," in *Proceedings of the International Conference on Worldwide Computing and Its Applications*, 1997.

[39] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the ub-tree into a database system kernel," in *VLDB*, 2000.

[40] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *ACM SIGMOD*, 2010.

[41] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *ACM SIGCOMM*, 2001.

[42] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *International Workshop on Cloud Data Management*, 2009.

[43] B. Cho and M. K. Aguilera, "Surviving congestion in geo-distributed storage systems," in *USENIX ATC*, 2012.

[44] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, 2014.

[45] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *USENIX OSDI*, 2010.

[46] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide Time to Relax*, 1st ed. O'Reilly Media, Inc., 2010.

CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal

Mingqiang Li*, Chuan Qin, and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
mingqiangli.cn@gmail.com, {cqin,pclee}@cse.cuhk.edu.hk

Abstract

We present *CDStore*, which disperses users' backup data across multiple clouds and provides a unified multi-cloud storage solution with reliability, security, and cost-efficiency guarantees. CDStore builds on an augmented secret sharing scheme called *convergent dispersal*, which supports deduplication by using deterministic content-derived hashes as inputs to secret sharing. We present the design of CDStore, and in particular, describe how it combines convergent dispersal with two-stage deduplication to achieve both bandwidth and storage savings and be robust against side-channel attacks. We evaluate the performance of our CDStore prototype using real-world workloads on LAN and commercial cloud testbeds. Our cost analysis also demonstrates that CDStore achieves a monetary cost saving of 70% over a baseline cloud storage solution using state-of-the-art secret sharing.

1 Introduction

Cloud storage provides cost-efficient means for organizations to host backups off-site [40]. However, from users' perspectives, putting all data in one cloud raises *reliability* concerns regarding the single point of failure [8] and vendor lock-in [5], especially when cloud storage providers can spontaneously terminate their business [35]. Cloud storage also raises *security* concerns, since data management is now outsourced to third parties. Users often want their outsourced data to be protected with guarantees of *confidentiality* (i.e., data is kept secret from unauthorized parties) and *integrity* (i.e., data is uncorrupted).

Multi-cloud storage coalesces multiple public cloud storage services into a single storage pool, and provides a plausible way to realize both reliability and security in outsourced storage. It disperses data with some form of redundancy across multiple clouds, operated by independent vendors, such that the stored data can be recovered from a subset of clouds even if the remaining clouds are unavailable. Redundancy can be realized through *erasure coding* (e.g., Reed-Solomon codes [51]) or *secret sharing* (e.g., Shamir's scheme [54]). Recent multi-cloud storage systems (e.g., [5, 19, 29, 33, 60]) leverage erasure coding to tolerate cloud failures, but do not address security; DepSky [13] uses secret sharing to further achieve both reliability and security. Secret sharing often comes with high redundancy, yet its variants are shown to reduce the redundancy of secret sharing to be slightly higher than that of erasure coding, while achieving security in the computational sense (see §2). Secret sharing has a side benefit of providing *keyless* security (i.e., eliminating encryption keys), which builds on the difficulty for an attacker to compromise multiple cloud services rather than a secret key. This removes the key management overhead as found in key-based encryption [56].

However, existing secret sharing algorithms prohibit storage savings achieved by *deduplication*. Since backup data carries substantial identical content [58], organizations often use deduplication to save storage costs, by keeping only one physical data copy and having it shared by other copies with identical content. On the other hand, secret sharing uses random pieces as inputs when generating dispersed data. Users embed different random pieces, making the dispersed data different even if the original data is identical.

This paper presents a new multi-cloud storage system called *CDStore*, which makes the first attempt to provide a unified cloud storage solution with reliability, security, and cost efficiency guarantees. CDStore builds on our prior proposal of an enhanced secret sharing scheme called *convergent dispersal* [37], whose core idea is to replace the random inputs of traditional secret sharing with deterministic cryptographic hashes derived from the original data, while the hashes cannot be inferred by attackers without knowing the whole original data. This allows deduplication, while preserving the reliability and keyless security features of secret sharing. Using convergent dispersal, CDStore offsets dispersal-level redundancy due to secret sharing by removing content-level redundancy via deduplication, and hence achieves cost efficiency. To summarize, we extend our prior work [37] and make three new contributions.

First, we propose a new instantiation of convergent dispersal called *CAONT-RS*, which builds on AONT-RS [52]. CAONT-RS maintains the properties of AONT-RS, and makes two enhancements: (i) using OAEP-based

*Mingqiang Li is now with Hong Kong Advanced Technology Center, Ecosystem & Cloud Service Group, Lenovo Group Ltd. This work was done when he was with the Chinese University of Hong Kong.

AONT [20] to improve performance and (ii) replacing random inputs with deterministic hashes to allow deduplication. Our evaluation also shows that CAONT-RS generates dispersed data faster than our prior AONT-RS-based instantiation [37].

Second, we present the design and implementation of CDStore. It adopts *two-stage deduplication*, which first deduplicates data of the same user on the client side to save upload bandwidth, and then deduplicates data of different users on the server side to further save storage. Two-stage deduplication works seamlessly with convergent dispersal, achieves bandwidth and storage savings, and is robust against side-channel attacks [27, 28]. We also carefully implement CDStore to mitigate computation and I/O bottlenecks.

Finally, we thoroughly evaluate our CDStore prototype using both microbenchmarks and trace-driven experiments. We use real-world backup and virtual image workloads, and conduct evaluation on both LAN and commercial cloud testbeds. We show that CAONT-RS encoding achieves around 180MB/s with only two-thread parallelization. We also identify the bottlenecks when CDStore is deployed in a networked environment. Furthermore, we show via cost analysis that CDStore can achieve a monetary cost saving of 70% via deduplication over AONT-RS-based cloud storage.

2 Secret Sharing Algorithms

We conduct a study of the state-of-the-art secret sharing algorithms. A secret sharing algorithm operates by transforming a data input called *secret* into a set of coded outputs called *shares*, with the primary goal of providing both fault tolerance and confidentiality guarantees for the secret. Formally, a secret sharing algorithm is defined based on three parameters (n, k, r): *an (n, k, r) secret sharing algorithm (where $n > k > r \geq 0$) disperses a secret into n shares such that (i) the secret can be reconstructed from any k shares, and (ii) the secret cannot be inferred (even partially) from any r shares.*

The parameters (n, k, r) define the protection strength of a secret sharing algorithm. Specifically, n and k determine the fault tolerance degree of a secret, such that the secret remains available as long as any k out of n shares are accessible. In other words, it can tolerate the loss of $n - k$ shares. The parameter r determines the confidentiality degree of a secret, such that the secret remains confidential as long as no more than r shares are compromised by an attacker. On the other hand, a secret sharing algorithm makes the trade-off of incurring additional storage. We define the *storage blowup* as the ratio of the total size of n shares to the size of the original secret. Note that the storage blowup must be at least $\frac{n}{k}$, as the secret is recoverable from any k out of n shares.

Several secret sharing algorithms have been proposed in the literature. Table 1 compares them in terms of the confidentiality degree and the storage blowup, subject to the same n and k. Two extremes of secret sharing algorithms are Shamir's secret sharing scheme (SSSS) [54] and Rabin's information dispersal algorithm (IDA) [50]. SSSS achieves the highest confidentiality degree (i.e., $r = k - 1$), but its storage blowup is n (same as replication). IDA has the lowest storage blowup $\frac{n}{k}$, but its confidentiality degree is the weakest (i.e., $r = 0$), and any share can reveal the information of the secret. Ramp secret sharing scheme (RSSS) [16] generalizes both IDA and SSSS to make a trade-off between the confidentiality degree and the storage blowup. It evenly divides a secret into $k - r$ pieces, and generates r additional random pieces of the same size. It then transforms the k pieces into n shares using IDA.

Algorithm	Confidentiality degree	Storage blowup†
SSSS [54]	$r = k - 1$	n
IDA [50]	$r = 0$	$\frac{n}{k}$
RSSS [16]	$r \in [0, k - 1]$	$\frac{n}{k-r}$
SSMS [34]	$r = k - 1$	$\frac{n}{k} + n \cdot \frac{S_{key}}{S_{sec}}$
AONT-RS [52]	$r = k - 1$	$\frac{n}{k} + \frac{n}{k} \cdot \frac{S_{key}}{S_{sec}}$

† S_{sec}: size of a secret; S_{key}: size of a random key.

Table 1: Comparison of secret sharing algorithms.

Secret sharing made short (SSMS) [34] combines IDA and SSSS using traditional key-based encryption. It first encrypts the secret with a random key and then disperses the encrypted secret and the key using IDA and SSSS, respectively. Its storage blowup is slightly higher than that of IDA, while it has the highest confidentiality degree $r = k - 1$ as in SSSS. Note that the confidentiality degree is defined in the computational sense, that is, it is computationally infeasible to break the encryption algorithm without knowing the key.

AONT-RS [52] further reduces the storage blowup of SSMS, while preserving the highest confidentiality degree $r = k - 1$ (in the computational sense). It combines Rivest's all-or-nothing transform (AONT) [53] for confidentiality and Reed-Solomon coding [17, 51] for fault tolerance. It first transforms the secret into an AONT package with a random key, such that an attacker cannot infer anything about the AONT package unless the whole package is obtained. Specifically, it splits a secret into a number $s \geq 1$ of words, and adds an extra *canary* word for integrity checking. It masks each of the s words by XOR'ing it with an index value encrypted by a random key. The s masked words are placed at the start of an AONT package. One more word, obtained by XOR'ing the same random key with the hash of the masked words, is added to the end of the AONT package. The final AONT package is then divided into k equal-size shares, which are encoded into n shares using a *system-*

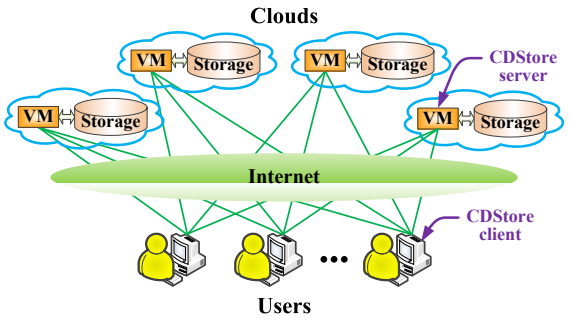


Figure 1: CDStore architecture.

atic Reed-Solomon code (a systematic code means that the n shares include the original k shares).

The security of existing secret sharing algorithms lies in the embedded random inputs (e.g., a random key in AONT-RS). Due to randomness, secrets with identical content lead to distinct sets of shares, thereby prohibiting deduplication. This motivates CDStore, which enables secret sharing with deduplication.

3 CDStore Design

CDStore is designed for an organization to outsource the storage of data of a large group of users to multiple cloud vendors. It builds on the client-server architecture, as shown in Figure 1. Each user of the same organization runs the *CDStore client* to store and access its data in multiple clouds over the Internet. In each cloud, a co-locating virtual machine (VM) instance owned by the organization runs the *CDStore server* between multiple CDStore clients and the cloud storage backend.

CDStore targets backup workloads. We consider a type of backups obtained by snapshotting some applications, file systems, or virtual disk images. Backups generally have significant identical content, and this makes deduplication useful. Field measurements on backup workloads show that deduplication can reduce the storage overhead by 10× on average, and up to 50× in some cases [58]. In CDStore deployment, each user machine submits a series of backup files (e.g., in UNIX `tar` format) to the co-located CDStore client, which then processes the backups and uploads them to all clouds.

3.1 Goals and Assumptions

We state the design goals and assumptions of CDStore in three aspects: reliability, security, and cost efficiency.

Reliability: CDStore tolerates failures of cloud storage providers and even CDStore servers. Outsourced data is accessible if a tolerable number of clouds (and their co-locating CDStore servers) are operational. CDStore also tolerates client-side failures by offloading metadata management to the server side (see §4.3). In the presence of cloud failures, CDStore reconstructs original secrets and then rebuilds the lost shares as in Reed-Solomon codes [51]. We do not consider cost-efficient repair [29].

Security: CDStore exploits multi-cloud diversity to ensure *confidentiality* and *integrity* of outsourced data against outsider attacks, as long as a tolerable number of clouds are uncompromised. Note that the confidentiality guarantee requires that the secrets be drawn from a very large message space, so that brute-force attacks are infeasible [10]. CDStore also uses two-stage deduplication (see §3.3) to avoid insider side-channel attacks [27, 28] launched by malicious users. Here, we do not consider strong attack models, such as Byzantine faults in cloud services [13]. We also assume that the client-server communication over the network is protected, so that an attacker cannot infer the secrets by eavesdropping the transmitted shares.

Cost efficiency: CDStore uses deduplication to reduce both bandwidth and storage costs. It also incurs limited overhead in computation (e.g., VM usage) and storage (e.g., metadata). We assume that there is no billing for the communication between a co-locating VM and the storage backend of the same cloud, based on today's pricing models of most cloud vendors [30].

3.2 Convergent Dispersal

Convergent dispersal enables secret sharing with deduplication by replacing the embedded random input with a deterministic cryptographic hash derived from the secret. Thus, two secrets with identical content must generate identical shares, making deduplication possible. Also, it is computationally infeasible to infer the hash without knowing the whole secret. Our idea is inspired by convergent encryption [24] used in traditional key-based encryption, in which the random key is replaced by the cryptographic hash of the data to be encrypted. Figure 2 shows the main idea of how we augment a secret sharing algorithm with convergent dispersal.

This paper proposes a new instantiation of convergent dispersal called *CAONT-RS*, which inherits the reliability and security properties of the original AONT-RS, and makes two key modifications. First, to improve performance, CAONT-RS replaces Rivest's AONT [53] with another AONT based on *optimal asymmetric encryption padding (OAEP)* [11, 20]. The rationale is that Rivest's AONT performs multiple encryptions on small-size words (see §2), while OAEP-based AONT performs a single encryption on a large-size, constant-value block. Also, OAEP-based AONT provably provides no worse security than any AONT scheme [20]. Second, CAONT-RS replaces the random key in AONT with a deterministic cryptographic hash derived from the secret. Thus, it preserves content similarity in dispersed shares and allows deduplication. Our prior work [37] also proposes instantiations for RSSS [16] and AONT-RS (based on Rivest's AONT) [52]. Our new CAONT-RS shows faster encoding performance than our prior AONT-RS-based instantiation (see §5.3).

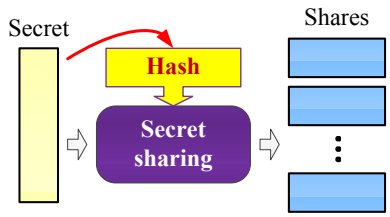


Figure 2: Idea of convergent dispersal.

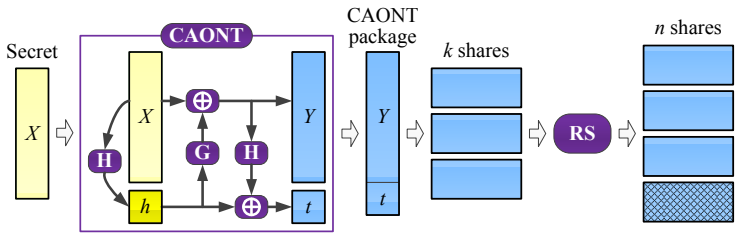


Figure 3: Example of CAONT-RS with $n = 4$ and $k = 3$.

We now elaborate on the encoding and decoding of CAONT-RS, both of which are performed by a CDStore client. Figure 3 shows an example of CAONT-RS with $n = 4$ and $k = 3$ (and hence $r = k - 1 = 2$).

Encoding: We first transform a given secret X into a CAONT package. Specifically, we first generate a hash key h, instead of a random key, derived from X using a (optionally salted) hash function $\mathbf{H}$ (e.g., SHA-256):

$$h = \mathbf{H}(X). \tag{1}$$

To achieve confidentiality, we transform (X, h) into a CAONT package (Y, t) using OAEP-based AONT, where Y and t are the head and tail parts of the CAONT package and have the same size as X and h, respectively. To elaborate, Y is generated by:

$$Y = X \oplus \mathbf{G}(h), \tag{2}$$

where '$\oplus$' is the XOR operator and $\mathbf{G}$ is a generator function that takes h as input and constructs a mask block with the same size as X. Here, we implement the generator $\mathbf{G}$ as:

$$\mathbf{G}(h) = \mathbf{E}(h, C), \tag{3}$$

where C is a constant-value block with the same size as X, and $\mathbf{E}$ is an encryption function (e.g., AES-256) that encrypts C using h as the encryption key.

The tail part t is generated by:

$$t = h \oplus \mathbf{H}(Y). \tag{4}$$

Finally, we divide the CAONT package into k equal-size shares (we pad zeroes to the secret if necessary to ensure that the CAONT package can be evenly divided). We encode them into n shares using the systematic Reed-Solomon codes [17, 46, 47, 51].

To enable deduplication, we ensure that the same share is located in the same cloud. Since the number of clouds for multi-cloud storage is usually small, we simply disperse shares to all clouds. Suppose that CDStore spans n clouds, which we label $0, 1, \cdots, n-1$. After encoding each secret using convergent dispersal, we label the n generated shares $0, 1, \cdots, n-1$ in the order of their positions in the Reed-Solomon encoding result, such that share i is to be stored on cloud i, where $0 \le i \le n-1$. This ensures that the same cloud always receives the same share from the secrets with identical content, either generated by the same user or different users. This also enables us to easily locate the shares during restore.

Decoding: To recover the secret, we retrieve any k out of n shares and use them to reconstruct the original CAONT package (Y, t). Then we deduce hash h by XOR'ing t with $\mathbf{H}(Y)$ (see Equation (4)). Finally, we deduce secret X by XOR'ing Y with $\mathbf{G}(h)$ (see Equation (2)), and remove any padded zeroes introduced in encoding.

We can also verify the integrity of the deduced secret X. We simply generate a hash value from the deduced X as in Equation (1) and compare if it matches h. If the match fails, then the decoded secret is considered to be corrupted. To obtain a correct secret, we can follow a brute-force approach, in which we try a different subset of k shares until the secret is correctly decoded [19].

Remarks: We briefly discuss the security properties of CAONT-RS. CAONT-RS ensures confidentiality against outsider attacks, provided that an attacker cannot gain unauthorized accesses to k out of n clouds, and ensures integrity through the embedded hash in each secret. It leverages AONT to ensure that no information of the original secret can be inferred from fewer than k shares. We note that an attacker can identify the deduplication status of the shares of different users and perform brute-force dictionary attacks [9, 10] inside the clouds, and we require that the secrets be drawn from a large message space (see §3.1). To mitigate brute-force attacks, we may replace the hash key in CAONT-RS with a more sophisticated key generated by a key server [9], with the trade-off of introducing the key management overhead.

3.3 Two-Stage Deduplication

We first overview how deduplication works. Deduplication divides data into fixed-size or variable-size *chunks*. This work assumes variable-size chunking, which defines boundaries based on content and is robust to content shifting. Each chunk is uniquely identified by a *fingerprint* computed by a cryptographic hash of the chunk content. Two chunks are said to be identical if their fingerprints are the same, and fingerprint collisions of two different chunks are very unlikely in practice [15]. Deduplication stores only one copy of a chunk, and refers any

duplicate chunks to the copy via small-size references.

To realize deduplication in cloud storage, a naïve approach is to perform global deduplication on the client side. Specifically, before a user uploads data to a cloud, it first generates fingerprints of the data. It then checks with the cloud by fingerprint for the existence of any duplicate data that has been uploaded by any user. Finally, it uploads only the unique data to the cloud. Although client-side global deduplication saves upload bandwidth and storage overhead, it is susceptible to *side-channel attacks* [27, 28]. One side-channel attack is to infer the existence of data of other users [28]. Specifically, an attacker generates the fingerprints of some possible data of other users and queries the cloud by fingerprint if such data is unique and needs to be uploaded. If no upload is needed, then the attacker infers that other users own the data. Another side-channel attack is to gain unauthorized access to data of other users [27]. Specifically, an attacker uses the fingerprints of some sensitive data of other users to convince the cloud of the data ownership.

To prevent side-channel attacks, CDStore adopts *two-stage deduplication*, which eliminates duplicates first on the client side and then on the server side. We require that each CDStore server maintains a deduplication index that keeps track of which shares have been stored by each user and how shares are deduplicated (see implementation details in §4.4). Then the two deduplication stages are implemented as follows.

Intra-user deduplication: A CDStore client first runs deduplication only on the data owned by the same user, and uploads the unique data of the user to the cloud. Before uploading shares to a cloud, the CDStore client first checks with the CDStore server by fingerprint if it has already uploaded the same shares. Specifically, the CDStore client first sends the fingerprints generated from the shares to the CDStore server. The CDStore server then looks up its deduplication index, and replies to the CDStore client a list of share identifiers that indicate which shares have been uploaded by the CDStore client. Finally, the CDStore client uploads only unique shares to the cloud based on the list.

Inter-user deduplication: A CDStore server runs deduplication on the data of all users and stores the globally unique data in the cloud storage backend. After the CDStore server receives shares from the CDStore client, it generates a fingerprint from each share (instead of using the one generated by the CDStore client for intra-user deduplication), and checks if the share has already been stored by other users by looking up the deduplication index. It stores only the unique shares that are not yet stored at the cloud backend. It also updates the deduplication index to keep track of which user owns the shares. Here, we cannot directly use the fingerprint generated by the CDStore client for intra-user deduplication. Otherwise, an attacker can launch a side-channel attack, by using the fingerprint of a share of other users to gain unauthorized access to the share [27, 43].

Remarks: Two-stage deduplication prevents side-channel attacks by making deduplication patterns independent across users' uploads. Thus, a malicious insider cannot infer the data content of other users through deduplication occurrences.

Both intra-user and inter-user deduplications effectively remove duplicates. Intra-user deduplication eliminates duplicates of the same user's data. This is effective for backup workloads, since the same user often makes repeated backups of the same data as different versions [32]. Inter-user deduplication further removes duplicates of multiple users. For example, multiple users within the same organization may share a large proportion of business files. Some workloads exhibit large proportions of duplicates across different users' data, such as VM images [31], workstation file system snapshots [42], and backups [58]. The removal of duplicates translates to cost savings (see §5.6).

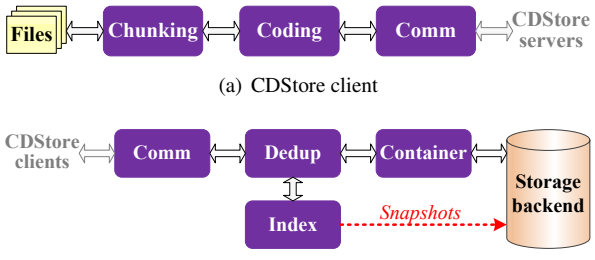
4 CDStore Implementation

We present the implementation details of CDStore. Our CDStore prototype is written in C++ on Linux. We use OpenSSL [4] to implement cryptographic operations: AES-256 and SHA-256 for the encryption and hash algorithms of convergent dispersal, respectively, and SHA-256 for fingerprints in deduplication. We use GF-Complete [48] to accelerate Galois Field arithmetic in the Reed-Solomon coding of CAONT-RS.

4.1 Architectural Overview

We follow a modular approach to implement CDStore, whose client and server architectures are shown in Figure 4. During file uploads, a CDStore client splits the file into a sequence of secrets via the *chunking module*. It then encodes each secret into n shares via the *coding module*. It performs intra-user deduplication, and uploads unique shares to the CDStore servers in n different clouds via both client-side and server-side *communication modules*. To reduce network I/Os, we avoid sending many small-size shares over the Internet. Instead, we first batch the shares to be uploaded to each cloud in a 4MB buffer and upload the buffer when it is full. Upon receiving the shares, each CDStore server performs inter-user deduplication via the *deduplication module* and updates the deduplication metadata via the *index module*. Finally, it packs the unique shares as containers and writes the containers to the cloud storage backend through the internal network via the *container module*.

File downloads work in the reverse way. A CDStore client connects to any k clouds to request to download a file. Each CDStore server retrieves the corresponding containers and metadata, and returns all required shares



(a) CDStore client

(b) CDStore server

Figure 4: Implementation of the CDStore architecture.

and file metadata. The CDStore client decodes the secrets and assembles the secrets back to the file.

4.2 Chunking

We implement both fixed-size chunking and variable-size chunking in the chunking module of a CDStore client, and enable variable-size chunking by default. To make deduplication effective, the size of each secret should be on the order of kilobytes (e.g., 8KB [62]). We implement variable-size chunking based on Rabin fingerprinting [49], in which the average, minimum, and maximum secret (chunk) sizes are configured at 8KB, 2KB, and 16KB, respectively.

4.3 Metadata Offloading

One important reliability requirement is to tolerate client-side failures, as we expect that a CDStore client is deployed in commodity hardware. Thus, our current implementation makes CDStore servers keep and manage all metadata on behalf of CDStore clients.

When uploading a file, a CDStore client collects two types of metadata. First, after chunking, it collects *file metadata* for the upload file, including the full pathname, file size, and number of secrets. Second, after encoding a secret into shares, it collects *share metadata* for each share, including the share size, fingerprint of the share (for intra-user deduplication), sequence number of the input secret, and secret size (for removing padded zeroes when decoding the original secret).

The CDStore client uploads the file and share metadata to the CDStore servers along with the uploaded file. The metadata will serve as input for each CDStore server to maintain index information (see §4.4).

We distribute metadata across all CDStore servers for reliability. For non-sensitive information (e.g., the size and sequence number of each secret), we can simply replicate it, so that each CDStore server can directly use it to manage data transfer and deduplication. However, for sensitive information (e.g., a file's full pathname), we encode and disperse it via secret sharing.

4.4 Index Management

Each CDStore server uses the metadata from CDStore clients to generate index information of the uploaded files and keep it in the index module. There are two types of index structures: the *file index* and the *share index*.

The file index holds the entries for all files uploaded by different users. Each entry describes a file, identified by the full pathname (which has been encoded as described in §4.3) and the user identifier provided by a CDStore client. We hash the full pathname and the user identifier to obtain a unique key for the entry. The entry stores a reference to the *file recipe*, which describes the complete details of the file, including the fingerprint of each share (for retrieving the share) and the size of the corresponding secret (for decoding the original secret). The file recipe will be saved at the cloud backend by the container module (see §4.5).

The share index holds the entries for all unique shares of different files. Each entry describes a share, and is keyed by the share fingerprint. It stores the reference to the container that holds the share. To support intra-user deduplication, each entry also holds a list of user identifiers to distinguish who owns the share, as well as a reference count for each user to support deletion.

Our prototype manages file and share indices using LevelDB [26], an open-source key-value store. LevelDB maintains key-value pairs in a log-structured merge (LSM) tree [44], which supports fast random inserts, updates, and deletes, and uses a Bloom filter [18] and a block cache to speed up lookups. We can also leverage the snapshot feature provided by LevelDB to store periodic snapshots in the cloud backend for reliability. We currently do not consider this feature in our evaluation.

4.5 Container Management

The container module maintains two types of containers in the storage backend: *share containers*, which hold the globally unique shares, and *recipe containers*, which hold the file recipes of different files. We cap the container size at 4MB, except that if a file recipe is very large (due to a particularly large file), we keep the file recipe in a single container and allow the container to go beyond 4MB. We avoid splitting a file recipe in multiple containers to reduce I/Os.

We make two optimizations to reduce the I/O overhead of storing and fetching the containers via the storage backend. First, we maintain in-memory buffers for holding shares and file recipes before writing them into containers. We organize the shares or file recipes by users, so that each container contains only the data of a single user. This retains spatial locality of workloads [62]. Second, we maintain a least-recently-used (LRU) disk cache to hold the most recently accessed containers to reduce I/Os to the storage backend.

4.6 Multi-Threading

Advances of multi-core architectures enable us to exploit multi-threading for parallelization. First, the client-

side coding module uses multi-threading for the CPU-intensive encoding/decoding operations of CAONT-RS. We parallelize encoding/decoding at the secret level: in file uploads, we pass each secret output from the chunking module to one of the threads for encoding; in file downloads, we pass the shares of a secret received by the communication module to a thread for decoding.

Furthermore, both client-side and server-side communication modules use multi-threading to fully utilize the network transfer bandwidth. The client-side communication module creates multiple threads, one for each cloud, to upload/download shares. The server-side communication module also uses multiple threads to send/receive shares for different CDStore clients.

4.7 Open Issues

Our current CDStore prototype implements the basic backup and restore operations. We discuss some open implementation issues.

Storage efficiency: We can reclaim more storage space via different techniques in addition to deduplication. For example, garbage collection can reclaim space of expired backups. By exploiting historical information, we can accelerate garbage collection in deduplication storage [25]. Compression also effectively reduces storage space of both data [58] and metadata (e.g., file recipes [41]). Implementations of garbage collection and compression are posed as future work.

Scalability: We currently deploy one CDStore server per cloud. In large-scale deployment, we can run CDStore servers on multiple VMs per cloud and evenly distribute user backup jobs among them for load balance. Implementing a distributed deduplication system is beyond the scope of this paper.

Consistency: Our prototype is tailored for backup workloads that are immutable. We do not address consistency issues due to concurrent updates as mentioned in [13].

5 Evaluation

We evaluate CDStore under different testbeds and workloads. We also analyze its monetary cost advantages.

5.1 Testbeds

We consider three types of testbeds in our evaluation.

(i) Local machines: We use two machines: *Local-Xeon*, which has a quad-core 2.4GHz Intel Xeon E5530 and 16GB RAM, and *Local-i5*, which has a quad-core 3.4GHz Intel Core i5-3570 and 8GB RAM. Both machines run 64-bit Ubuntu 12.04.2 LTS. We use them to evaluate the encoding performance of CDStore clients.

(ii) LAN: We configure a LAN of multiple machines with the same configuration as Local-i5. All nodes are connected via a 1Gb/s switch. We run CDStore clients and servers on different machines. Each CDStore server mounts the storage backend on a local 7200RPM SATA hard disk. We use the LAN testbed to evaluate the data transfer performance of CDStore.

(iii) Cloud: We deploy a CDStore client on the Local-Xeon machine (in Hong Kong) and connect it via the Internet to four commercial clouds (i.e., $n = 4$): Amazon (in Singapore), Google (in Singapore), Azure (in Hong Kong), and Rackspace (in Hong Kong). We set up the testbed in the same continent to limit the differences among the client-to-server connection bandwidths. Each cloud runs a VM with similar configurations: four CPU cores and 4$\sim$15GB RAM. We use the cloud testbed to evaluate the real deployment performance of CDStore.

5.2 Datasets

We use two real-world datasets to drive our evaluation.

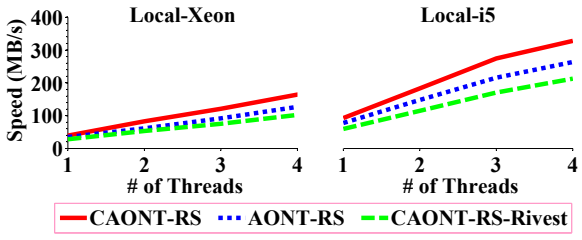
(i) FSL: This dataset is published by the File systems and Storage Lab (FSL) at Stony Brook University [3,57]. Due to the large dataset size, we use the `Fslhomes` dataset in 2013, containing daily snapshots of nine students' home directories from a shared network file system. We select the snapshots every seven days (which are not continuous) to mimic weekly backups. The dataset is represented in 48-bit chunk fingerprints and corresponding chunk sizes obtained from variable-size chunking. Our filtered FSL dataset contains 16 weekly backups of all nine users, covering a total of 8.11TB of data.

(ii) VM: This dataset is collected by ourselves and is unpublished. It consists of weekly snapshots of 156 VM images for students in a university programming course in Spring 2014. We create a 10GB master image with Ubuntu 12.04.2 LTS and clone all VMs. We treat each VM image snapshot as a weekly backup of a user. The dataset is represented in SHA-1 fingerprints on 4KB fixed-size chunks. It spans 16 weeks, totaling 24.38TB of data. For fair comparisons, we remove all zero-filled chunks (which dominate in VM images [31]) from the dataset, and the size reduces to 11.12TB.

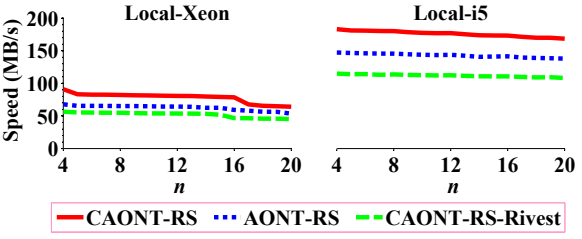
5.3 Encoding Performance

We evaluate the computational overhead of CAONT-RS when encoding secrets into shares. We compare CAONT-RS with two variants: (i) *AONT-RS* [52], which builds on Rivest's AONT [53] and does not support deduplication, and (ii) our prior proposal *CAONT-RS-Rivest* [37], which uses Rivest's AONT as in AONT-RS and replaces the random key in AONT-RS with a SHA-256 hash for convergent dispersal. CAONT-RS uses OAEP-based AONT instead (see §3.2).

We conduct our experiments on the Local-Xeon and Local-i5 machines. We create 2GB of random data in memory (to remove I/O overhead), generate secrets using variable-size chunking with an average chunk size 8KB, and encode them into shares. We measure the *encoding speed*, defined as the ratio of the original data size to the total time of encoding all secrets into shares. Our results

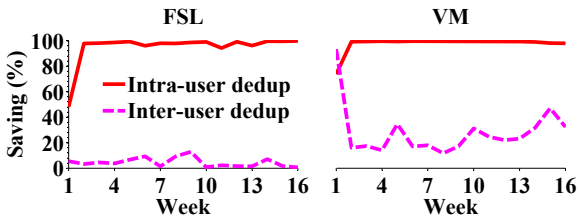


(a) Varying number of threads

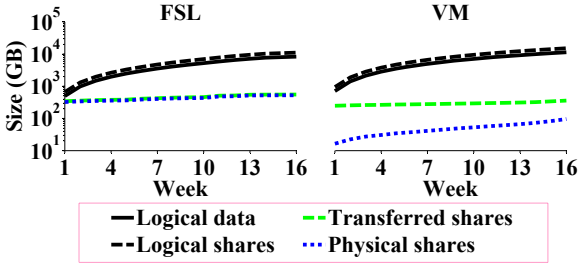


(b) Varying n

Figure 5: Encoding speeds of a CDStore client.



(a) Intra-user and inter-user deduplication savings



(b) Cumulative data and share sizes when $(n, k) = (4, 3)$

Figure 6: Deduplication efficiency of CDStore.

are averaged over 10 runs. We observe similar results for decoding, and omit them here.

We first examine the benefits of multi-threading (see §4.6). Figure 5(a) shows the encoding speeds versus the number of threads, while we fix $(n, k) = (4, 3)$. The encoding speeds of all schemes increase with the number of threads. If two encoding threads are used, the encoding speeds of CAONT-RS are 83MB/s on Local-Xeon and 183MB/s on Local-i5. Also, OAEP-based AONT in CAONT-RS brings remarkable performance gains. Compared to CAONT-RS-Rivest, which performs encryptions on small words based on Rivest's AONT, CAONT-RS improves the encoding speed by 40∼61% on Local-Xeon and 54∼61% on Local-i5; even though compared to AONT-RS, which uses one fewer hash operation, CAONT-RS still increases the encoding speed by 12∼35% on Local-Xeon and 19∼27% on Local-i5.

We next evaluate the impact of n (number of clouds). We vary n from 4 to 20, and fix two encoding threads. We configure k as the largest integer that satisfies $\frac{k}{n} \leq \frac{3}{4}$ (e.g., $n = 4$ implies $k = 3$), so as to maintain a similar storage blowup due to secret sharing. Figure 5(b) shows the encoding speeds versus n. The encoding speeds of all schemes slightly decrease with n (e.g., by 8% from $n = 4$ to 20 for CAONT-RS on Local-i5), since more encoded shares are generated via Reed-Solomon codes for a larger n. However, Reed-Solomon coding only accounts for small overhead compared to AONT, which runs cryptographic operations. We have also tested other ratios of $\frac{k}{n}$ and obtained similar speed results.

The above results only report encoding speeds, while a CDStore client performs both chunking and encoding operations when uploading data to multiple clouds. We measure the combined chunking (using variable-size chunking) and encoding speeds with $(n, k) = (4, 3)$ and two encoding threads, and find that the combined speeds drop by around 16%, to 69MB/s on Local-Xeon and 154MB/s on Local-i5.

5.4 Deduplication Efficiency

We evaluate the effectiveness of both intra-user and inter-user deduplications (see §3.3). We extract the deduplication characteristics of both datasets, assuming that they are stored as weekly backups. We define four types of data: (i) *logical data*, the original user data to be encoded into shares, (ii) *logical shares*, the shares before two-stage deduplication, (iii) *transferred shares*, the shares that are transferred over Internet after intra-user deduplication, and (iv) *physical shares*, the shares that are finally stored after two-stage deduplication. We also define two metrics: (i) *intra-user deduplication saving*, which is one minus the ratio of the size of the transferred shares to that of the logical shares, and (ii) *inter-user deduplication saving*, which is one minus the ratio of the size of the physical shares to that of the transferred shares. We fix $(n, k) = (4, 3)$. Figure 6 summarizes the results.

Figure 6(a) first shows the intra-user and inter-user deduplication savings. The intra-user deduplication savings are very high for both datasets, especially in subsequent backups after the first week (at least 94.2% for FSL and at least 98.0% for VM). The reason is that the users only modify or add a small portion of data. The savings translate to performance gains in file uploads (see §5.5). However, the inter-user deduplication savings differ across datasets. For the FSL dataset, the savings fall to no more than 12.9%. In contrast, for the VM dataset, the saving for the first backup reaches 93.4%, mainly because the VM images are initially installed with the same

operating system. The savings for subsequent backups then drop to the range between 11.8% and 47.0%. Nevertheless, the VM dataset shows higher savings for subsequent backups than the FSL dataset; we conjecture the reason is that students make similar changes to the VM images when doing programming assignments.

Figure 6(b) then shows cumulative data and share sizes before and after intra-user and inter-user deduplications. After 16 weekly backups, for the FSL dataset, the total size of physical shares is only 0.51TB, about 6.3% of the logical data size; for the VM dataset, the total size of physical shares is only 0.09TB, about 0.8% of the logical data size. This shows that dispersal-level redundancy (i.e., $\frac{n}{k} = \frac{4}{3}$) is significantly offset by removing content-level redundancy via two-stage deduplication. Also, if we compare the sizes of transferred shares and physical shares for the VM dataset, we see that inter-user deduplication is crucial for reducing storage space.

5.5 Transfer Speeds

Single-client baseline transfer speeds: We first evaluate the baseline transfer speed of a CDStore client using both LAN and cloud testbeds. Each testbed has one CDStore client and four CDStore servers with $(n, k) = (4, 3)$. We first upload 2GB of unique data (i.e., no duplicates), then upload another 2GB of duplicate data identical to the previous one, and finally download the 2GB data from three CDStore servers (for the cloud testbed, we choose Google, Azure, and Rackspace for downloads). We measure the upload and download speeds, averaged over 10 runs.

Figure 7(a) presents the results. On the LAN testbed, the upload speed for unique data is 77MB/s. Our measurements find that the effective network speed in our LAN testbed is around 110MB/s. Thus, the upload speed for unique data is close to $\frac{k}{n}$ of the effective network speed. Uploading duplicate data has speed 150MB/s. Since it does not transfer actual data after intra-user deduplication, the performance is bounded by the chunking and CAONT-RS encoding operations (see §5.3). The download speed is 99MB/s, about 10% less than the effective network speed. The reason is that the CDStore servers need to retrieve data from the disk backend before returning it to the CDStore client.

On the cloud testbed, the upload and download performance is limited by the Internet bandwidth. For references, we measure the upload and download speeds of each individual cloud when transferring 2GB of unique data divided in 4MB units (see §4.1), and Table 2 presents the averaged results over 10 runs. Since CDStore transfers data through multiple clouds in parallel via multi-threading, its upload speed of unique data and download speed are higher than those of individual clouds (e.g., Amazon and Google). The upload speed for unique data is smaller than the download speed because of sending redundancy and connecting to more clouds. The upload speed for duplicate data is over 9× that for unique data, and this difference is more significant than on the LAN testbed.

Cloud	Upload speed	Download speed
Amazon	5.87 (0.19)	4.45 (0.30)
Google	4.99 (0.23)	4.45 (0.21)
Azure	19.59 (1.20)	13.78 (0.72)
Rackspace	19.42 (1.06)	12.93 (1.47)

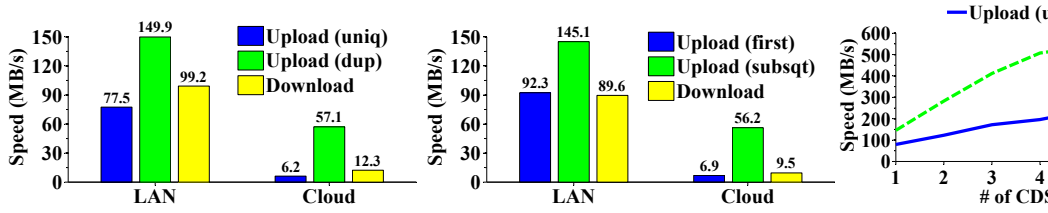
Table 2: Measured speeds (MB/s) of each of four clouds, in terms of the average (standard deviation) over 10 runs.

Single-client trace-driven transfer speeds: We now evaluate the upload and download speeds of a single CDStore client using datasets as opposed to unique and duplicate data above. We focus on the FSL dataset, which allows us to test the effect of variable-size chunking. We again consider both LAN and cloud testbeds with $(n, k) = (4, 3)$. Since the FSL dataset only has chunk fingerprints and chunk sizes, we reconstruct a chunk by writing the fingerprint value repeatedly to a chunk with the specified size, so as to preserve content similarity. Each chunk is treated as a secret, which will be encoded into shares. We first upload all backups to CDStore servers, followed by downloading them. To reduce evaluation time, we only run part of the dataset. On the LAN testbed, we run seven weekly backups for five users (1.06TB data in total). We feed the first week of backups of each user one by one through the CDStore client, followed by the second week of backups, and so on. On the other hand, on the cloud testbed, we run two weekly backups for a single user (21.35GB data in total).

Figure 7(b) presents three results: (i) the average upload speed for the first backup (averaged over five users for the LAN testbed), (ii) the average upload speed for the subsequent backups, and (iii) the average download speed of all backups. The presented results are obtained from a single run, yet the evaluation time is long enough to give steady-state results. We compare the results with those for unique and duplicate data in Figure 7(a).

We see that the upload speed for the first backup exceeds that for unique data (e.g., by 19% on the LAN testbed), mainly because the first backup contains duplicates, which can be removed by intra-user deduplication (see Figure 6(a)). The upload speed for the subsequent backups approximates to that for duplicate data, as most duplicates are again removed by intra-user deduplication.

The trace-driven download speed is lower than the baseline one in Figure 7(a) (e.g., by 10% on the LAN testbed), since deduplication now introduces chunk fragmentation [38] for subsequent backups. Nevertheless, we find that the variance of the download speeds of the backups is very small (not shown in the figure), although



(a) Baseline results (b) Trace-driven results

Figure 7: Upload and download speeds of a CDStore client (the numbers are the speeds in MB/s).

Figure 8: Aggregate upload speeds of multiple CDStore clients.

the number of accessed containers increases for subsequent backups. The download speed will gradually degrade due to fragmentation as we store more backups. We do not explicitly address fragmentation in this work.

Multi-client aggregate upload speeds: We evaluate the aggregate upload speed when multiple CDStore clients connect to multiple CDStore servers. We mainly consider data uploads on the LAN testbed, in which we vary the number of CDStore clients, each hosted on a dedicated machine, and configure four CDStore servers with $(n, k) = (4, 3)$. All CDStore clients perform uploads concurrently, such that each of them first uploads 2GB of unique data, and then uploads another 2GB of duplicate data. We measure the *aggregate upload speed*, defined as the total upload size (i.e., 2GB times the number of clients) divided by the duration when all clients finish uploads. Our results are averaged over 10 runs.

Figure 8 presents the aggregate upload speeds for both unique and duplicate data, which we observe increase with the number of CDStore clients. For unique data, the aggregate upload speed reaches 282MB/s for eight CDStore clients. The speed is limited by the network bandwidth and disk I/O, where the latter is for the CDStore servers to write containers to disk. If we exclude disk I/O (i.e., without writing data), the aggregate upload speed can reach 310MB/s (not shown in the figure), which approximates to the aggregate effective Ethernet speed of $k = 3$ CDStore servers. For duplicate data, there is no actual data transfer, so the aggregate upload speed can reach 572MB/s. Note that the knee point at four CDStore clients is due to the saturation of CPU resources in each CDStore server.

5.6 Cost Analysis

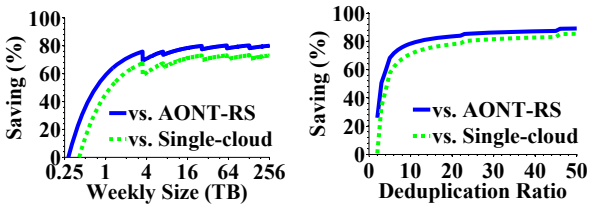
We now analyze the cost saving of CDStore. We compare it with two baseline systems: (i) an AONT-RS-based multi-cloud system that has the same levels of reliability and security as CDStore but does not support deduplication, and (ii) a single-cloud system that incurs zero redundancy for reliability, but encrypts user data with random keys and does not support deduplication. We aim to show that CDStore incurs less cost than AONT-RS through deduplication; even though CDStore incurs redundancy for reliability, it still incurs less cost than the single-cloud system without deduplication.

We develop a tool to estimate the monetary costs using the pricing models of Amazon EC2 [1] and S3 [2] in September 2014. Free charges apply to data transfers between co-locating EC2 instances and S3 storage, and also inbound transfers to both EC2 and S3. We only study backup operations, and do not consider restore operations as they are relatively infrequent in practice. Note that both EC2 and S3 follow tiered pricing, so the exact charges depend on the actual usage. Our tool takes into account tiered pricing in cost calculations. For CDStore, we also consider the storage costs of file recipes.

We briefly describe how we derive the EC2 and S3 costs. For EC2, we consider the category of high-utilization reserved instances, which are priced based on an upfront fee and hourly bills. We focus on two types of instances, namely compute-optimized and storage-optimized, to host CDStore servers on all clouds. Each instance charges around US$60∼1,300 per month, depending on the CPU, memory, and storage settings. Note that both file and share indices (see §4.4) are kept in the local storage of an EC2 instance, and the total index size is determined by how much data is stored and how much data can be deduplicated. Our tool chooses the cheapest instance that can keep the entire indices according to the storage size and deduplication efficiency, both of which can be estimated in practice. On the other hand, S3 storage is mainly priced based on storage size, and it charges around US$30 per TB per month. Note that in backup operations, the costs due to outbound transfer (e.g., a CDStore server replies the intra-user deduplication status to a CDStore client) and storage requests (e.g., PUT) are negligible compared to VM and storage costs.

We consider a case study. An organization schedules weekly backups for its user data, for a retention time of half a year (26 weeks). We fix $(n, k) = (4, 3)$ (i.e., we host four EC2 instances for CDStore servers). We vary the weekly backup size and the deduplication ratio, where the latter is defined as the ratio of the size of logical shares to the size of physical shares (see §5.4).

Figure 9(a) shows the cost savings of CDStore versus different weekly backup sizes, while we fix the deduplication ratio as 10× [58]. The cost savings increase with the weekly backup size. For example, if we



(a) Varying weekly backup size (b) Varying deduplication ratio

Figure 9: Cost savings of CDStore over an AONT-RS-based multi-cloud system and a single-cloud system.

keep a weekly backup size of 16TB, the single-cloud and AONT-RS-based systems incur total storage costs (with tiered pricing) of around US$12,250/month and US$16,400/month, respectively; CDStore incurs additional VM costs of around US$660/month but reduces the storage cost to around US$2,880/month, resulting in around US$3,540/month in total and thus achieving at least 70% of cost savings as a whole. The cost saving of CDStore over AONT-RS is higher than that over a single cloud, as the former introduces dispersal-level redundancy for fault tolerance. The increase slows down as the weekly backup size further increases, since the overhead of file recipes becomes significant when the total backup size is large while the backups have a high deduplication ratio [41]. Note that the jagged curves are due to the switch of the cheapest EC2 instance to fit the indices.

Figure 9(b) shows the cost savings of CDStore versus different deduplication ratios, where the weekly backup size is fixed at 16TB. The cost saving increases with the deduplication ratio. The saving is about 70∼80% when the deduplication ratio is between 10× and 50×.

6 Related Work

Multi-cloud storage: Existing multi-cloud storage systems mainly focus on data availability in the presence of cloud failures and vendor lock-ins. For example, SafeStore [33], RACS [5], Scalia [45], and NCCloud [29] disperse redundancy across multiple clouds using RAID or erasure coding. Some multi-cloud systems additionally address security. HAIL [19] proposes proof of retrievability to support remote integrity checking against data corruptions. MetaStorage [12] and SPANStore [60] provide both availability and integrity guarantees by replicating data across multiple clouds using quorum techniques [39], but do not address confidentiality. Hybris [23] achieves confidentiality by dispersing encrypted data over multiple public clouds via erasure coding and keeping secret keys in a private cloud.

Applications of secret sharing: We discuss several secret sharing algorithms in §2. They have been realized by storage systems. POTSHARDS [56] realizes Shamir's scheme [54] for archival storage. ICStore [21] achieves confidentiality via key-based encryption, where the keys are distributed across multiple clouds via Shamir's scheme. DepSky [13] and SCFS [14] distribute keys across clouds using SSMS [34]. Cleversafe [52] uses AONT-RS to achieve security with reduced storage space. All the above systems rely on random inputs to secret sharing, and do not address deduplication.

Deduplication security: Convergent encryption [24] provides confidentiality guarantees for deduplication storage, and has been adopted in various storage systems [6, 7, 22, 55, 59]. However, the key management overheads of convergent encryption are significant [36]. Bellare *et al.* [10] generalize convergent encryption into Message-locked encryption (MLE) and provide formal security analysis on confidentiality and tag consistency. The same authors also prototype a server-aided MLE system DupLESS [9], which uses more complicated encryption keys to prevent brute-force attacks. DupLESS maintains the keys in a dedicated key server, yet the key server is a single point of failure.

Client-side inter-user deduplication poses new security threats, including the side-channel attack [27,28] and some specific attacks against Dropbox [43]. CDStore addresses this problem through two-stage deduplication. A previous work [61] proposes a similar two-stage deduplication approach (i.e., inner-VM and cross-VM deduplications) to reduce system resources for VM backups, while our approach is mainly to address security.

7 Conclusions

We propose a multi-cloud storage system called CDStore for organizations to outsource backup and archival storage to public cloud vendors, with three goals in mind: reliability, security, and cost efficiency. The core design of CDStore is convergent dispersal, which augments secret sharing with the deduplication capability. CDStore also adopts two-stage deduplication to achieve bandwidth and storage savings and prevent side-channel attacks. We extensively evaluate CDStore via different testbeds and datasets from both performance and cost perspectives. We demonstrate that deduplication enables CDStore to achieve cost savings. The source code of our CDStore prototype is available at **http://ansrlab.cse.cuhk.edu.hk/software/cdstore**.

Acknowledgments

We would like to thank our shepherd, Fred Douglis, and the anonymous reviewers for their valuable comments. This work was supported in part by grants ECS CUHK419212 and GRF CUHK413813 from HKRGC.

References

[1] Amazon EC2. `http://aws.amazon.com/ec2/`, 2014.

[2] Amazon S3. http://aws.amazon.com/s3/, 2014.

[3] FSL traces and snapshots public archive. http://tracer.filesystems.org/, 2014.

[4] OpenSSL Project. http://www.openssl.org, 2014.

[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 229–240, Indianapolis, IN, June 2010.

[6] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 1–14, Boston, MA, Dec. 2002.

[7] P. Anderson and L. Zhang. Fast and secure laptop backups with encrypted de-duplication. In *Proceedings of the 24th Large Installation System Administration Conference (LISA '10)*, pages 1–12, San Jose, CA, Nov. 2010.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaha. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, Apr. 2010.

[9] M. Bellare, S. Keelveedhi, and T. Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proceedings of the 22nd USENIX Security Symposium (Security '13)*, pages 179–194, Washington, DC, Aug. 2013.

[10] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT '13)*, pages 296–312, Athens, Greece, May 2013.

[11] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Proceedings of the 1994 Workshop on the Theory and Application of Cryptographic Techniques (EUROCRYPT '94)*, pages 92–111, Perugia, Italy, May 1994.

[12] D. Bermbach, M. Klems, S. Tai, and M. Menzel. MetaStorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD '11)*, pages 452–459, Washington, DC, July 2011.

[13] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):1–33, Nov. 2013.

[14] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 169–180, Philadelphia, PA, June 2014.

[15] J. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC'06)*, pages 85–90, Boston, MA, June 2006.

[16] G. R. Blakley and C. Meadows. Security of ramp schemes. In *Proceedings of CRYPTO '84 on Advances in Cryptology*, pages 242–268, Santa Barbara, CA, Aug 1984.

[17] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug. 1995.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[19] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 187–198, Chicago, IL, Nov. 2009.

[20] V. Boyko. On the security properties of OAEP as an all-or-nothing transform. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO '99)*, pages 503–518, Santa Barbara, CA, Aug. 1999.

[21] C. Cachin, R. Haas, and M. Vukolić. Dependable storage in the intercloud. IBM Research Report RZ 3783, May 2010.

[22] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 1–14, Boston, MA, Dec. 2002.

[23] D. Dobre, P. Viotti, and M. Vukolić. Hybris: Robust hybrid cloud storage. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SoCC'14)*, pages 1–14, Seattle, WA, Nov. 2014.

[24] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate

files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 617–624, Vienna, Austria, July 2002.

[25] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, pages 181–192, Philadelphia, PA, June 2014.

[26] S. Ghemawat and J. Dean. LevelDB: A fast and lightweight key/value database library by Google. `http://code.google.com/p/leveldb/`, 2014.

[27] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pages 491–500, Chicago, IL, Oct. 2011.

[28] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, Nov-Dec 2010.

[29] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, pages 265–272, San Jose, CA, Feb. 2012.

[30] R. Jellinek, Y. Zhai, T. Ristenpart, and M. Swift. A day late and a dollar short: The case for research on cloud billing systems. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '14)*, pages 1–6, Philadelphia, PA, June 2014.

[31] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the 2009 Israeli Experimental Systems Conference (SYSTOR '09)*, pages 1–12, Haifa, Israel, May 2009.

[32] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12)*, pages 1–12, Haifa, Israel, June 2012.

[33] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC '07)*, pages 129–142, Santa Clara, CA, June 2007.

[34] H. Krawczyk. Secret sharing made short. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93)*, pages 136–146, Santa Barbara, CA, Aug. 1993.

[35] S. Lelii. Nirvanix prepares to close, tells customers to stop using its cloud. `http://itknowledgeexchange.techtarget.com/storage-soup/nirvanix-to-customers-stop-sending-data-to-our-cloud/`, Sept. 2013.

[36] J. Li, X. Chen, M. Li, J. Li, P. P. C. Lee, and W. Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1615–1625, June 2014.

[37] M. Li, C. Qin, P. P. C. Lee, and J. Li. Convergent dispersal: Toward storage-efficient security in a cloud-of-clouds. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*, pages 1–5, Philadelphia, PA, June 2014.

[38] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 183–197, San Jose, CA, Feb. 2013.

[39] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC '97)*, pages 569–578, El Paso, TX, May 1997.

[40] C. McLellan. Storage in 2014: An overview. `http://www.zdnet.com/storage-in-2014-an-overview-7000024712/`, Jan. 2014.

[41] D. Meister, A. Brinkmann, and T. Süß. File recipe compression in data deduplication systems. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 175–182, San Jose, CA, Feb. 2013.

[42] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, pages 1–14, San Jose, CA, Feb. 2011.

[43] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proceedings of the 20th USENIX Security Symposium (Security '11)*, pages 65–76, San Francisco, CA, Aug. 2011.

[44] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[45] T. G. Papaioannou, N. Bonvin, and K. Aberer. Scalia: An adaptive scheme for efficient multi-cloud storage. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 1–10, Salt Lake City, UT, Nov. 2012.

[46] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software — Practice & Experience*, 27(9):995–1012, Sept. 1997.

[47] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software — Practice & Experience*, 35(2):189–194, Feb. 2005.

[48] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*, pages 299–306, San Jose, CA, Feb. 2013.

[49] M. Rabin. Fingerprint by random polynomials. Technical report, Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[50] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, Apr. 1989.

[51] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

[52] J. K. Resch and J. S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, pages 191–202, San Jose, CA, Feb. 2011.

[53] R. L. Rivest. All-or-nothing encryption and the package transform. In *Proceedings of the 4th International Workshop on Fast Software Encryption (FSE '97)*, pages 210–218, Haifa, Israel, Jan. 1997.

[54] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[55] M. W. Storer, K. Greenan, D. D. E. Long, and E. L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS '08)*, pages 1–10, Fairfax, VA, Oct. 2008.

[56] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS — a secure, recoverable, long-term archival storage system. *ACM Transactions on Storage*, 5(2):1–35, June 2009.

[57] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*, pages 261–272, Boston, MA, June 2012.

[58] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, pages 33–48, San Jose, CA, Feb. 2012.

[59] Z. Wilcox-O'Hearn and B. Warner. Tahoe — the least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability (StorageSS '08)*, pages 21–26, Fairfax, VA, Oct. 2008.

[60] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 292–308, Farmington, PA, Nov. 2013.

[61] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng. Multi-level selective deduplication for VM snapshots in cloud storage. In *Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing (CLOUD '12)*, pages 550–557, Honolulu, HI, June 2012.

[62] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 269–282, San Jose, CA, Feb. 2008.

Surviving Peripheral Failures in Embedded Systems

Rebecca Smith
Rice University
rjs@rice.edu

Scott Rixner
Rice University
rixner@rice.edu

Abstract

Peripherals fail. Yet, modern embedded systems largely leave the burden of tolerating peripheral failures to the programmer. This paper presents Phoenix, a semi-automated peripheral recovery system for resource-constrained embedded systems. Phoenix introduces lightweight checkpointing mechanisms that transparently track both the internal program state and the external peripheral state. These mechanisms enable rollback to the precise point at which any failed peripheral access occurred using as little as 6 KB of memory, minimizing both recovery latency and memory utilization.

1 Introduction

Embedded systems drive the modern world, controlling devices all around us from toasters to automobiles. Due to resource constraints, these systems are typically programmed in low-level languages, with the application developer manually managing system resources. To increase programmer productivity, managed run-time systems are gaining popularity within the domain of embedded systems [1, 2, 3, 4]. Run-time systems ease the burden on the programmer by automating resource management and supporting high-level programming languages. The small amount of execution variability that these run-time systems introduce can be tolerated, in order to gain productivity, by many of the soft real-time workloads that are prevalent in modern embedded systems.

Embedded systems are typically event-driven, interacting with the real world by reading sensors, making decisions, and controlling actuators. Since these systems impact their environment, failures can have real-world consequences. Microcontrollers often have built-in support for reliability [19, 31], and could implement additional known processor reliability techniques [5, 41, 43]. However, the sensors and actuators at the heart of these systems are extremely susceptible to failures [9, 20]. Despite this, techniques for fault tolerance in embedded systems primarily focus on handling microcontroller failures [8, 22, 46, 48, 50], leaving management of peripheral failures entirely to the programmer.

This is a complex task for the programmer, as peripherals can fail asynchronously. The programmer would have to anticipate all possible scenarios and determine how to correctly restore the system state after an arbitrary number of instructions and peripheral accesses have executed. Such an application would be difficult to write and even harder to test, and still might have problems if peripherals fail at unanticipated and inopportune times. Therefore, it is crucial to develop mechanisms to facilitate recovery from peripheral failures.

These mechanisms should be largely transparent to the programmer, as is the case in many other domains such as large-scale distributed systems [15, 21, 25, 35, 39]. However, many existing system-level techniques require active participation in the reliability protocols by the elements that can fail; such techniques are ill-suited to handling peripheral failures in embedded systems, as these peripherals are incapable of such participation. One approach that has been effective in embedded systems is checkpointing, which enables the system to roll back to a known valid state, correct failures, and re-execute. However, existing checkpointing schemes for embedded systems do not support peripheral failures [8, 46, 50].

This paper presents Phoenix, a semi-automated peripheral recovery system built around novel checkpointing mechanisms. Phoenix provides fault tolerance for fail-stop peripheral failures, access timeouts, violations in communications protocols, and interrupt storms or other spurious interrupts. The design of Phoenix was motivated by a set of insights into the unique properties of embedded systems and their ramifications on recovery.

In particular, peripherals introduce complexities that must be addressed by any recovery process for correctness. First, many peripherals have external state, which is fundamentally different from internal program state. Internal state can be rolled back simply by resetting the memory. In contrast, the interactions that pe-

ripherals have with the real world may be difficult to undo. Furthermore, some peripherals have transient effects, while others have lasting effects. These differences influence how each peripheral must be handled during recovery. Finally, peripherals may interact with each other. Such dependencies determine the necessary actions to restore the external state. While some peripheral accesses should be replayed, in other cases it is incorrect to redo the access, so it must be skipped.

These complexities, taken in conjunction with stringent resource constraints, introduce several challenges to implementing efficient checkpointing in embedded systems. First, the external peripheral state must be restored along with the internal program state; traditional checkpointing mechanisms neglect this peripheral state. Second, simply taking a snapshot of the memory is intractable. Embedded systems have a small amount of memory, the majority of which is needed by the application itself. It is not practical to keep even a single copy of the memory at any given time. Last, the performance limitations of resource-constrained embedded systems motivate minimizing the latency of recovery. Ideally, the system would roll back to exactly the point before the particular access that failed, minimizing the number of instructions and peripheral accesses to be re-executed.

Based on these insights, the checkpointing mechanisms in Phoenix simultaneously track internal and external state, optimize memory utilization, and enable rollback to any precise point at which a peripheral failure could occur. If a failure occurs, Phoenix automatically rolls back and recovers while keeping the internal and external state consistent. For efficiency, the system only tracks state when there is a chance of failure. Checkpointing is automatically turned on when a peripheral is accessed, and turned off once all past accesses have succeeded. While checkpointing is enabled the system builds an incremental log of the state, maintaining pointers into this log corresponding to each peripheral access. This incremental approach serves two purposes. First, it only checkpoints the minimal amount of state required for rollback. Second, it allows the system to roll back to the point right before any peripheral access that could fail, thereby minimizing re-execution.

Evaluation on a set of microbenchmarks and applications showed that Phoenix is space-efficient enough to operate within the resource constraints of embedded systems. Running on a microcontroller with 96 KB of SRAM, Phoenix used on the order of 5 KB to track both the internal and external system state, leaving the majority of the space free for use by the running program. When there was a failure the overhead rose to 6 KB, as a small amount of extra state is needed during the recovery process. Furthermore, for two of the three applications studied in this paper there was no perceivable change in performance with the addition of Phoenix.

The mechanisms of Phoenix transcend peripheral failures. In particular, any recovery strategy for embedded systems should restore both internal and external state when any asynchronous failure occurs. Phoenix's techniques for logging and restoring these two types of state could be combined with the appropriate failure detection mechanisms to handle additional types of failures.

The rest of the paper proceeds as follows. Section 2 presents the key insights that motivate the design of Phoenix, and Section 3 presents the recovery procedure. Sections 4 and 5 describe the mechanisms used to implement the system. Section 6 shows the experimental evaluation of Phoenix, and Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Key Insights

Two sets of insights into the unique properties of embedded systems shaped the design of Phoenix. The first set consists of broad insights into the implications of peripherals on system behavior and therefore correct system recovery. Supplementing these, a second set of insights influenced the adaptation of a specific reliability technique, checkpointing, to the domain of embedded systems.

2.1 Peripheral State in Embedded Systems

Embedded systems are inherently event-driven, interacting with their surroundings through sensors and actuators. These peripherals have unique properties which manifest themselves in subtle and complex ways, and which must be addressed in order to correctly recover from a failure. Five insights into these peripherals drive the design of the mechanisms presented in this paper.

First, hardware peripherals introduce external state in addition to the internal state. To guarantee correct recovery from failures, Phoenix restores both types of state.

Second, the way that peripherals affect external state varies. Based on this, peripherals can be classified into four categories: stateless, ephemeral, persistent, or historical. The first category includes simple sensors such as accelerometers which cannot affect their surroundings. Ephemeral peripherals do affect the external state, albeit fleetingly; these include, for example, buzzers. In contrast, persistent peripherals have lasting effects. The state of a persistent peripheral is entirely determined by the last write to it. For instance, setting the speed of a motor causes a state change that persists until a new speed is set. Historical peripherals likewise have lasting state, but the state of a historical peripheral is an aggregation of a series of prior writes. Phoenix selects the recovery actions to perform for a given peripheral based on the classification it was registered with during initialization.

Third, peripherals do not operate in isolation; the state of one may impact the behavior of another. Such de-

pendencies are often specific to the context of a particular application. A peripheral P1 has a dependency on a peripheral P2 if P2 failing results in P1 not having its intended effect on the external state. As an example, consider an autonomous car that uses a motor and steering servo to drive. The servo will rotate the car's wheels regardless of whether the motor is functioning. However, if the servo's goal in this application is turning the car itself, the motor failing will prevent it from accomplishing its goal. Thus, the servo has a dependency on the motor.

Fourth, not all peripheral accesses can be replayed. Consider a historical peripheral such as a message passing interface between two devices. If a message is read from this interface, but an unrelated peripheral fails before the message is processed, re-executing the read is incorrect for two reasons. First, the program must process the original message. Moreover, the state of the message passing interface is not only external but shared; depending on the messaging protocol being used, the device on the other end of the connection may not re-send, so a re-read may time out. Instead, this read must be rematerialized: skipped during re-execution, in favor of re-using the original return value. In contrast, accesses to peripherals that depended on the failed peripheral must be re-executed. When a peripheral fails, the Phoenix system uses the dependency information to populate an initial set of peripherals to redo; accesses to peripherals not in this set are rematerialized. As re-execution proceeds, this set is updated as necessary to adapt to changes in state.

Finally, the lasting effects of persistent peripherals demand additional mechanisms for restoring their state. Correct restoration requires three steps. First, prior to recovering from a failure, all persistent peripherals must be put into a safe state; otherwise, they may continue operating in an erroneous state. Next, all persistent peripherals selected for re-execution must be restored to the state that they held prior to the failed access. That way, when re-execution begins, they will be in the same state that they were at this point in the original execution. Last, once re-execution is complete, all persistent peripherals whose accesses were rematerialized must be set to the last state that was rematerialized, so that they are in the correct state going forward. Phoenix automatically performs each of these three steps, ensuring a consistent state throughout the duration of the recovery process.

2.2 Checkpointing in Embedded Systems

Checkpointing is often used to provide fault-tolerance in domains such as mobile and distributed systems [25, 35, 39]. At a high level, the concept of tracking and restoring a known consistent state extends well to other domains. However, the resource-constrained nature of embedded systems presents three unique challenges to adapting existing checkpointing mechanisms.

First, all of the insights introduced in Section 2.1 must be considered when designing *any* recovery mechanism for embedded systems; checkpointing is no exception. In the context of checkpointing, this means that this external peripheral state must be logged and rolled back just like the internal program state. Phoenix therefore logs every peripheral access that is performed. During re-execution, it makes the decision of whether to re-execute or skip on the granularity of an individual peripheral access.

Second, embedded systems face tight memory constraints. For example, the TI Stellaris LM3S9B92 [47], a typical ARM Cortex-M3 microcontroller, has only 96 KB of SRAM and 256 KB of flash memory. Phoenix addresses the lack of memory space by using a logging technique that resembles journaling filesystems [10, 17, 26, 38]. Rather than preemptively checkpointing some or all of the memory, Phoenix only copies memory that has actually been changed. Moreover, Phoenix takes advantage of the fact that when there are no outstanding peripheral accesses, there is no chance of peripheral failure. Thus, Phoenix automatically disables logging once all peripheral accesses have been acked, re-enabling logging only when another peripheral access is issued.

Last, embedded systems also face time constraints. The LM3S9B92 operates at 50 MHz; therefore, minimizing the rollback latency is crucial to maintaining performance. This is magnified by the fact that even soft real-time embedded applications are typically event-driven, and thus require some degree of reactivity. Phoenix guarantees that when a peripheral fails there will be sufficient checkpointing data to roll back to the exact point of failure, avoiding unnecessary re-execution. It achieves this by maintaining one logical checkpoint for each outstanding peripheral access. Each checkpoint is identified by pointers into checkpointing structures that contain only a small subset of the memory at any given time.

3 Recovery Procedure

This section presents a semi-automated recovery procedure for peripheral failures that builds on the insights from Section 2. This procedure was implemented in Owl [3, 7], an embedded run-time system including a Python bytecode interpreter that runs on the bare metal. Peripheral accesses are structured as function calls made through Owl's native function interface. These functions access the hardware through a thin C library. While minor implementation details were tailored to Owl, the procedure and mechanisms of Phoenix transcend Owl and could be implemented in other run-time systems.

When a peripheral access fails, it must be re-executed in order to achieve correct program behavior. However, re-executing this access in the context of the arbitrary execution state in which its failure is detected may be insufficient or incorrect. First, it is likely that many byte-

codes were executed between the peripheral access and the detection of its failure. These bytecodes could have changed the program state. If the peripheral access interacts with the internal program state at all, such as through its parameters, then in order to re-execute it correctly the state of the program must be restored to the point at which the access originally occurred. Second, naively re-executing the same peripheral access may simply result in another failure. Thus, the failed peripheral must be recovered prior to re-execution. Last, some of the operations performed after the peripheral access was issued but before its failure was detected may have depended on the success of the failed access; in this case, those operations must be re-executed as well.

The Phoenix system translates these facts into a three step recovery process which automatically executes upon detection of failure. First, the internal state is rolled back to the point of the failed access. Second, the failed peripheral is recovered. Third, the correct external state is reached via redo mode execution. Within redo mode, the system re-executes the failed access, as well as all accesses to dependent peripherals, but rematerializes accesses to unrelated peripherals. Once execution reaches the point of failure detection, the system seamlessly exits redo mode and resumes normal execution.

As an example, consider Figure 1, which is sample code for an autonomous car that uses a motor and steering servo to drive and an SD card to record its movements. Assume that the peripheral access in line 3 fails, and the system identifies this failure after line 8.

Upon detecting this failure, the system will first put the motor and servo into safe states. It will then restore the internal state as it was immediately prior to line 3, resetting the value of the variable `speed` to 100. The system will recover the motor by invoking a programmer-provided recovery function, and will then enter redo mode. During redo mode, line 3 will be re-executed, since it failed initially. However, line 4 will be rematerialized, as there is no dependency between the motor and the SD card. This rematerialization is a requirement for correctness. Effectively, redo mode restores the peripheral state that would have been reached had there been no failure at all. It would be inaccurate to replay this write

```
1  # Run the motor
2  speed = 100
3  motor.run(speed)
4  SD.write('set motor to 100')
5  speed += 100
6
7  # Turn the wheels
8  servo.set_servo(LEFT)
9  ...
```

Figure 1: Sample Application Code

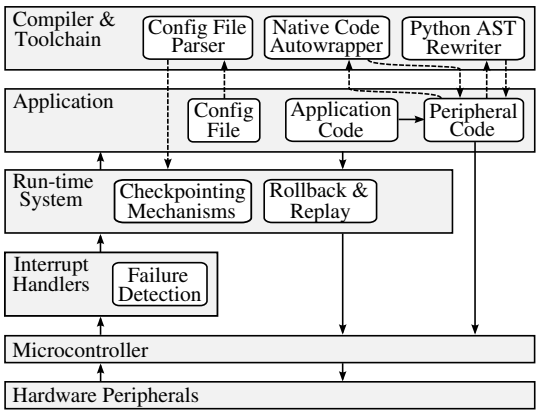


Figure 2: Phoenix System Architecture

and record the motor as having been set to 100 twice, when in fact the first attempt was unsuccessful. Next, line 5 will be re-executed, because it affects the internal state, which was rolled back. Last, the servo depends on the motor, so line 8 will also be re-executed. Finally, normal execution will resume.

4 System Mechanisms

This section presents the interlocking compiler and run-time system mechanisms which work together to carry out the procedure presented in Section 3.

At a high level, the compiler supports recovery in two key ways: it injects code to enable checkpointing prior to each peripheral access, and it auto-generates code to log the arguments and return value of each peripheral access. Then, when a failure is detected at run-time, the run-time system automatically identifies the failed peripheral, puts all persistent peripherals into a safe state, and executes the three step recovery procedure.

None of these mechanisms alone are sufficient for recovery. But, taken as a whole, they form a complete recovery system which facilitates every step from failure detection to rollback, recovery, and re-execution. Figure 2 presents an overview of the Phoenix system architecture, illustrating how the various components of the Phoenix system, in white, fit into the Owl system, in grey.

4.1 Failure Detection

Microcontrollers offer a variety of external peripheral interfaces, which notify the processor of events via interrupts. For example, the LM3S9B92 includes universal asynchronous receiver/transmitters (UART), inter-integrated circuit interfaces (I2C), and other interfaces.

Phoenix provides two system-level mechanisms to interact with these external interfaces: interrupt handlers and light-weight software shims. Since these two mechanisms are the link between the peripherals and the rest of the Phoenix system, Phoenix automatically recovers from all failures that they can detect, primarily consist-

ing of communication failures such as fail-stop peripheral failures, timeouts, and spurious interrupts. Upon detecting a failure, these mechanisms identify the failed peripheral and set the three step recovery procedure in motion by throwing an exception to the interpreter.

Most failures are detected solely via interrupt handlers. However, some external interfaces do not provide the appropriate type of interrupts for failure detection; in these cases, the interrupt handlers are supplemented by the software shims. Further, the existing software detection mechanisms could be augmented to perform additional validation that the peripheral is operating correctly, thereby extending fault coverage without any changes to the rest of the system.

4.1.1 Hardware Detection

The interrupt handlers in Phoenix manage both successful and unsuccessful peripheral accesses. Each interrupt handler begins by pinpointing the peripheral whose access generated the interrupt. During initialization (see Section 5.2), each peripheral is registered with an (interface, interface details) pair that uniquely identifies it to the interrupt handler. As an example, the pair for an I2C device would look like (I2C, (master address, slave address)). Thus, the handler identifies the source of the interrupt by searching the set of registered peripherals.

Next, the interrupt handler checks the status flags to determine whether the access succeeded. If so, the interrupt handler acks it by decrementing the appropriate counter of outstanding reads or writes. Otherwise, it throws a rollback exception to the interpreter. A special block in the interpreter handles this exception by performing the three step procedure discussed in Section 3.

4.1.2 Software Detection

Not all peripheral interfaces provide a convenient interrupt mechanism to detect success and failure. For example, the interrupts for the UART interface on the LM3S9B92 are designed to allow asynchronous transmission and reception of data; thus, they convey whether or not the device is ready for a subsequent transfer, not whether previous transfers have succeeded or failed.

To address this limitation, the system was augmented with a small buffer to hold UART data. Each entry contains a byte that was received by the UART and associated error bits. As the hardware receives data, the interrupt handler transfers the data, and any error bits, to the software buffer. UART reads made by the application then access the software buffer. When a read occurs on a byte that was successfully transferred, the peripheral access is acked, as if there had been a "success" interrupt. When a read occurs on a byte that experienced a transmission error, the recovery procedure is triggered, as if a "failure" interrupt had occurred. This is a general approach that can be used for any peripheral for which the interface does not provide interrupts that enable the system to determine the success or failure of each access.

4.2 Checkpointing

To enable rollback and re-execution, the Phoenix system performs checkpointing whenever there is a chance of peripheral failure. Checkpointing is automatically enabled prior to each peripheral access. While checkpointing is enabled, Phoenix maintains three structures: a journal, a control flow queue, and a set of rematerialization queues. These structures persist past rollback, as they live on a second recovery heap which is not checkpointed. They are freed on a rolling basis as accesses succeed. Further, once all past accesses have succeeded, rollback past the current point is impossible, so Phoenix disables checkpointing until the next access is issued.

4.2.1 Software Journal

In Owl, all of the program state is stored on the Python heap. So, when checkpointing is enabled, the history of this heap must be tracked to enable rollback of the internal program state. When an assembly language store to this heap occurs, Phoenix first records a journal entry containing the memory address to be stored and its old contents. The journal only needs space proportional to the number of stores to the Python heap since checkpointing was last enabled, rather than the entire size of the heap. On failure, the run-time system can roll back the state of the Python heap by restoring the contents of the journal in reverse order.

The prototype implementation of Phoenix keeps the journal using software that exploits existing hardware mechanisms defined by the ARM ISA. In particular, Phoenix uses the ARM Cortex-M3's memory protection unit (MPU), flash remap hardware, and system call instruction (SVC) to keep the journal. These are the only hardware requirements of Phoenix.

The journaling process proceeds as follows. First, when checkpointing is enabled, the regions of memory covering the Python heap are set read-only. This causes the memory management fault handler to run upon each store to this heap. It receives both the address of the instruction that caused the fault and the memory address of the faulting access. Based on the type of instruction (ARM has "store multiple" instructions), it determines how many journal entries to populate. Each entry contains a faulting address and the contents of that address.

If the journal fills, an exception is thrown telling the system to wait for prior peripheral accesses to be acked. Because the failure detection mechanisms only write to the recovery heap — which is not checkpointed — as opposed to the Python heap, there is no chance of deadlock. Once all accesses have succeeded, the system clears the

journal and disables logging. Execution can then resume.

After the store instruction completes, memory protection must be turned back on. The fault handler remaps the instruction that follows the store to become an SVC instruction. Note, however, that the program is executing from flash memory, so the instruction cannot be overwritten. Instead, Phoenix uses the Cortex-M3 *flash patch and breakpoint* support, which allows small regions of flash to be "patched" by remapping them to SRAM. When the system call executes, MPU protection is turned back on. The flash remapping is turned off, and the return from the SVC is routed back to the instruction following the store.

The software journal is effective, but it incurs an overhead for each protected store to the Python heap. Since two exception handlers must run for each protected store, there is a minimum overhead of 48 cycles (12 cycles to enter and 12 cycles to exit each exception handler), plus any pipeline flushes. In practice, the handlers themselves execute for hundreds of additional cycles; during evaluation on a series of microbenchmarks, the overhead per protected store averaged 308 cycles.

4.2.2 Hardware Journal

To reduce the journal overhead, the microcontroller could be augmented with a simple hardware journal. This would involve minimal changes to the ARM ISA: MPU support for an additional "log" mode, a circular buffer with entries that can hold a memory location and a 32-bit value to serve as the journal, and a pair of registers to store head and tail pointers into the journal. If the tail pointer ever becomes equal to the head pointer, the journal is full, and an interrupt would be generated. In response to such an interrupt, the Phoenix system would wait, as described in Section 4.2.1.

When a store instruction writes to memory in a region that has the log attribute set, the hardware would first store the address and the contents at that address in the journal, incrementing the tail pointer modulo the size of the journal. The stage of the store instruction that writes to SRAM would complete in three cycles instead of one. First, the original write to SRAM would be aborted. Second, the contents of SRAM would be read. Last, three writes would occur in parallel: the original write, the journal write, and the tail pointer update. This two cycle overhead is two orders of magnitude lower than that of the software journal, which would yield perceptible improvements for applications with frequent journaling.

4.2.3 Rematerialization Queues

The rematerialization queues checkpoint the external state. Each peripheral has a queue, with one entry per access. These entries serve two purposes: storing the arguments and return value for use during redo mode, and storing the rollback point in case of failure. Since peripheral accesses are structured as native function calls, the return value is not raw data from the peripheral but rather a Python object allocated on the heap. The rematerialization queue holds deep copies of these objects, allocated on the recovery heap in order to persist past rollback.

On failure, Phoenix selects the right checkpoint based on the identity of the failed peripheral. Each checkpoint corresponds to a single access, so metadata for a given checkpoint is stored in the corresponding queue entry: indices into the journal and control flow queue, as well as pointers into the other rematerialization queues.

After rollback, the system constructs a set of peripherals to redo based on programmer-specified dependencies. Each time a peripheral access call is reached during redo mode, Phoenix checks to see if the peripheral is in the redo set. If so, the function is called as usual. Otherwise, Phoenix compares the new arguments against those in the rematerialization queue entry. If they match, the function call is skipped; Phoenix pops the new arguments from the stack and pushes the old return value. If they do not match, the peripheral is added to the redo set and the function call is re-executed with the new arguments.

A given rematerialization queue entry is freed once the peripheral access it corresponds to, plus all earlier accesses, are acked, since these acks guarantee that the system will never roll back to (or past) this entry. The contents of the journal up to the index stored within this rematerialization queue entry are freed at the same time.

4.2.4 Control Flow Queue

When checkpointing is enabled, the control flow queue keeps track of the instruction pointer of each Python bytecode that is executed. During redo mode, this queue is used to determine when to resume normal execution. Redo mode is exited either when the point at which failure was detected is reached once more, or when control flow diverges from the original path. For instance, a re-executed peripheral read may return a different value the second time through and cause a different path to be taken at a branch. Phoenix would then exit redo mode. This allows the system to adapt to changes in the external state that have occurred since the initial execution.

4.3 Compilation

To enable rollback, Phoenix records the journal and control flow queue indices at each checkpoint. Checkpoints correspond to peripheral access function calls — in Python, CALL_FUNCTION bytecodes. However, it is insufficient to re-execute the failed function call with the same arguments, as they may change during recovery. For instance, an I2C device access may take the master address as a parameter. If this address was passed as a variable, and this variable was changed by the recovery function, its new value must be loaded prior to

re-execution. As such, the true checkpoint is not the CALL_FUNCTION bytecode but the first bytecode that loads an argument for the call. These checkpoint locations are difficult to identify at run-time. Thus, Phoenix introduces a new bytecode, JOURNAL_STORE, and adds a custom AST rewriter to the Python compiler which inserts a JOURNAL_STORE prior to loading the arguments for each peripheral access. On reaching this bytecode at run-time, the interpreter stores the current journal and control flow indices and enables checkpointing.

Additionally, Phoenix uses an autowrapping tool to inject code before and after each peripheral access. Prior to a peripheral access, the autowrapper adds code to create a rematerialization queue entry. After the access, it inserts code to increment the number of outstanding reads or writes to this peripheral and set the new rematerialization queue entry's return value. These additions facilitate the run-time system's job in two ways. First, tracking outstanding accesses allows the system to automatically free the checkpointing structures and disable checkpointing as accesses complete. Second, allocating queue entries is a prerequisite for rematerialization.

5 Programmer Mechanisms

Phoenix disentangles the bulk of the application code from the peripheral code. Making this peripheral code recoverable requires that the programmer follow a few simple rules during development, involving minor refactoring and a small amount of new code. For the three applications evaluated in Section 6.2, an additional 9–17 lines of Python and 66–76 lines of C were required.

5.1 Peripheral Classification

Each peripheral must extend one of four provided peripheral classes, which mirror the categories from Section 2.1; an example is shown in Figure 3. Applications using the same hardware peripheral can share the same peripheral class. The class is largely written in Python; the only C code that the programmer must write is that of the low-level access functions. These can then be called from the Python code via Owl's native function interface.

Peripheral access functions should encapsulate an atomic unit of work, since rollback and re-execution occur at the granularity of a single function call. Additionally, each peripheral class must support recovery. All peripheral classes must define a `recover` method, which the system automatically calls after rollback. This gives the programmer flexibility in deciding how to handle a failed peripheral, such as by resetting it, switching in a hot spare, or even signaling for user intervention. Yet, the programmer need never worry about the complex bookkeeping and control flow logic to determine when to call this method. This is the only method that must be defined for stateless and ephemeral peripherals.

```
1  class Motor(PersistentPeripheral):
2      def __init__(self):
3          # Initialize primary device
4          self.init(PRIMARY)
5
6      def recover(self):
7          # Switch to backup device
8          self.init(BACKUP)
9
10     def safe_state(self):
11         self.set_speed(0)
12
13     def last_state(self, *args):
14         native_write(*args)
```

Figure 3: Peripheral Class Excerpt

Because persistent peripherals have lasting state, they require two additional restoration methods, motivated by the final insight in Section 2.1: `safe_state` and `last_state`. Before rollback, Phoenix automatically invokes all `safe_state` methods to put the peripherals into states that will have minimal effects during subsequent recovery steps. For example, a motor might be set to a speed of zero. As with `recover`, the definition of `safe_state` is up to the programmer.

Defining `last_state` is trivial, as it follows a standard template: take a variable number of arguments and perform a write with those arguments. This allows Phoenix to invoke `last_state` by simply passing the arguments from a rematerialization queue entry, regardless of the write function's signature. During redo mode, the system calls `last_state` in two places. Peripherals whose accesses must be replayed are set to their last state at the beginning of redo mode; peripherals whose accesses were not replayed during redo mode are set to their last state prior to resuming normal execution.

Restoring a historical peripheral is challenging, as its state is an aggregation of multiple past writes. The naive solution is to re-execute every past write. However, this would prevent freeing the rematerialization queues, which is untenable given limited memory. While Phoenix does not currently include specialized mechanisms for historical peripherals, this does not preclude their use with Phoenix. Consider an LCD display. Many applications will update a display by completely redrawing its contents, which would work seamlessly with Phoenix. More generally, a finite number of pixels comprise the display's state. Their values could be stored in a buffer programmatically and restored by `recover`, requiring no extra support from Phoenix.

5.2 Peripheral Initialization

To enable failure detection, the peripheral class must do two things upon initialization: register the new instance and enable interrupts. Phoenix provides a

peripheral_register function which must be invoked with the interface and interface details for the peripheral, as described in Section 4.1.1. Since interrupt handlers detect success or failure, and interface information is needed to determine which peripheral generated a given interrupt, registering peripherals and enabling interrupts must occur prior to any peripheral accesses. However, registration and interrupt information can be updated at any time to reflect changes to the hardware configuration. In particular, if the recover function switches to a backup device, it should also re-register the peripheral and enable interrupts for any new interfaces.

5.3 Config File

Last, the programmer must provide a config file for each application containing peripheral metadata. First, the programmer must declare dependencies. Second, the programmer must specify how many interrupts the peripheral access functions generate. This allows Phoenix to treat an access as an atomic unit and determine when it has completed. Writing the config file proved trivial for the three applications studied in this paper.

6 Evaluation

This section will first introduce the microbenchmarks and applications used to evaluate the Phoenix system. It will then assess Phoenix, showing that its space overhead is minimal and its time overhead is completely hidden in realistic workloads.

6.1 Microbenchmarks

The microbenchmarks use two persistent peripherals, a gyroscope and compass, and follow a common structure. They are named in the form <peripherals>_<actions>, where <peripherals> is a subset of {gyro, comp} and <actions> is a subset of {r, w, c}, for read, write, compute. After the peripherals are initialized, the actions are performed in a loop in the order in which they are listed.

Even in the absence of failures, checkpointing necessarily incurs a time overhead, primarily due to journaling. Across all of the benchmarks listed in Table 1, the weighted average cost of adding an entry to the software journal was 6.2 μs. Given a 20 ns cycle length, this means that one journaled store took, on average, 308 cycles. A hardware journal, with an overhead of only 2 cycles per store, would yield dramatic improvements. Introducing a single failure incurred a relatively small additional overhead of 12–143 ms. This overhead did not exceed 45 ms for benchmarks with one peripheral; for gyro_comp_wr the overhead was larger due to additional safe_state and last_state calls.

This overhead is incurred in support of speculative execution. Another approach would be to wait for each access to be acked before proceeding. However, the length of the control flow queue reveals that Phoenix takes advantage of significant opportunities to make progress where a stop-and-wait system would not. As shown in Tables 1 and 5, the control flow queue reached lengths of 9–18 during the period in which a wait-based system would be stalling. This is non-trivial; one Python bytecode may, for example, execute an entire native function.

Table 1: Benchmark Checkpointing Structures, With and Without Failure (max live entries)

Benchmark	JNL		CFQ		RMQ (comp, gyro)	
	Failure?		Failure?		Failure?	
	No	Yes	No	Yes	No	Yes
gyro_r	220	220	9	14	(0, 2)	(0, 3)
gyro_w	165	182	7	14	(0, 2)	(0, 3)
gyro_wr	220	220	9	14	(0, 2)	(0, 3)
comp_r	144	169	6	6	(1, 0)	(2, 0)
comp_w	192	226	9	9	(2, 0)	(3, 0)
comp_wr	192	211	9	9	(2, 0)	(3, 0)
gyro_comp_wr	220	301	9	11	(2, 2)	(3, 2)
comp_wcr	190	219	9	9	(2, 0)	(3, 0)
comp_wrc	190	211	9	9	(2, 0)	(3, 0)
comp_wcrc	192	211	9	9	(2, 0)	(3, 0)

Just as the checkpointing process takes time, the structures storing the state require space. There are three checkpointing structures: the journal (JNL), control flow queue (CFQ), and rematerialization queues (RMQ). Each journal entry requires 6 bytes, as the software journal is optimized to store an offset into the 64 KB heap rather than the full address. A control flow queue entry requires 4 bytes. Phoenix uses a fixed-size journal of 512 entries, and a fixed-size control flow queue of 64 entries. Rematerialization queues are linked lists. Each entry consumes a minimum of (36 + (72 * n)) bytes, where n is the number of hardware peripherals, plus variable space for arguments. Owl's best-fit memory allocator also introduces a small amount of variance.

A long-running program may generate many entries in these structures over the course of its execution. However, since past entries are discarded as acks are received, very few entries are live at the same point in time. Table 1 shows the maximum number of live entries for each of the three checkpointing structures when the benchmarks were run with no failures and with a single failure. No benchmark required more than 301 journal entries, 14 control flow queue entries, or 3 rematerialization queue entries per peripheral. Thus, the fixed sizes for the journal and control flow queue proved more than sufficient, never exceeding 59% or 22% capacity, respectively.

Tables 2 and 3 show the total space overhead with and without failure. The checkpointing structures are the primary source of this overhead. However, Phoenix additionally maintains a small amount of metadata. Peripheral metadata tracks the active peripherals in the system,

Table 2: Benchmark Overhead, Without Failure (bytes)

Benchmark	comp_r	comp_w	comp_wr	comp_wcrc	gyro_r	gyro_w	gyro_wr	gyro_comp_wr
JNL	3120	3120	3120	3120	3120	3120	3120	3120
CFQ	284	284	284	284	284	284	284	284
RMQ	276	472	472	472	484	484	484	752
Peripheral Metadata	268	268	268	268	268	268	268	384
Recovery Metadata	24	24	24	24	24	24	24	24
Total	3972	4168	4168	4168	4180	4180	4180	4564

Table 3: Benchmark Overhead, With Failure (bytes)

Benchmark	comp_r	comp_w	comp_wr	comp_wcrc	gyro_r	gyro_w	gyro_wr	gyro_comp_wr
JNL	3120	3120	3120	3120	3120	3120	3120	3120
CFQ	284	284	284	284	284	284	284	284
RMQ	664	656	656	656	656	656	656	984
Peripheral Metadata	268	268	268	268	268	268	268	384
Recovery Metadata	304	300	300	300	304	304	304	300
Total	4436	4636	4636	4636	4632	4632	4632	5072

growing with the number of peripherals. Some of this metadata is generated at boot-time based on data parsed from the config file; the rest is created when a peripheral is initialized at run-time. The same build of the run-time system was used for all benchmarks; thus, there were two peripherals' worth of boot-time metadata in all of the benchmarks, and the total peripheral metadata size does not double on activating the second peripheral.

Recovery metadata contains information needed to perform rollback and re-execution, including which peripheral failed and which peripherals to redo. Thus, the size of this metadata is initially negligible and grows when a failure occurs. The combined overhead of both types of metadata never exceeded 408 B (gyro_comp_wr) in the benchmarks without failure; with failure, it reached a maximum of only 684 B (gyro_comp_wr).

The total space overhead of Phoenix was relatively consistent across all benchmarks. With no failures, it began at 3.9 KB for the case where a single peripheral was read but not written (comp_r), as this minimized the number of live rematerialization queue entries. Introducing writes (comp_w, comp_wr, comp_wcrc) added an additional 196 B, as a second live rematerialization queue entry is required to support `last_state`. The gyro benchmarks did not show this same increase upon adding writes, as the gyro requires a single write during initialization, and therefore even gyro_r held a second rematerialization queue entry. The largest increase occurs when additional active peripherals are introduced, as in gyro_comp_wr. This is due to two factors: extra rematerialization queue entries and extra peripheral metadata.

These space requirements do not change drastically in the case of failure; only the rematerialization queue entries and the recovery metadata consume additional space. The expansion of the rematerialization queues is due to the fact that the system maintains segments of the queues from the original execution in order to replay accesses. At the same time, replaying these accesses results in additional entries being generated. Thus, during redo mode there are brief periods in which two rematerialization queue entries exist for the same access.

Table 4: Car Interval Length (ms)

Benchmark	Minimum	Maximum	Average
Without Phoenix	30	44	30
Phoenix (No Failure)	30	44	30
Phoenix (Failure)	30	44	30

Overall, the space overhead of the Phoenix system is quite small. Across all benchmarks, the overhead never exceeded 4.5 KB when no failures occurred, nor did it exceed 5.0 KB when one failure occurred. Using a mere 4.0–5.2% of SRAM, Phoenix leaves most of the space available for the running program. At the same time, it maintains multiple simultaneous checkpoints, each of which is positioned at precisely the latest possible location to which Phoenix could roll back in order to recover.

6.2 Applications

Phoenix was also evaluated on three applications, each representative of a different pattern of peripheral accesses. The first application is an autonomous RC car. The microcontroller is attached to three types of peripherals: a motor, a steering servo, and two gyroscopes. An event loop controls the car's movements by reading from the gyro and writing to the motor and steering servo. The second application is an obstacle tracker which periodically logs the distance to the nearest obstacle by reading from one of two range finders and writing to a display.

Table 5: Application Checkpointing Structures, With and Without Failure (max live entries)

Benchmark	JNL		CFQ		RMQ (comp, display, finder, gyro, motor)	
	Failure?		Failure?		Failure?	
	No	Yes	No	Yes	No	Yes
autonomous car	220	220	9	15	(0, 0, 0, 2, 2)	(0, 0, 0, 3, 2)
obstacle tracker	346	346	18	18	(0, 2, 1, 0, 0)	(0, 2, 2, 0, 0)
virtual compass	346	346	18	18	(1, 2, 0, 0, 0)	(2, 2, 0, 0, 0)

Table 6: Application Overhead, No Failure (bytes)

Application	autonomous car	obstacle tracker	virtual compass
JNL	3120	3120	3120
CFQ	284	284	284
RMQ	876	1184	1360
Peripheral Metadata	476	352	368
Recovery Metadata	36	24	24
Total	4792	4964	5156

Table 7: Application Overhead, With Failure (bytes)

Application	autonomous car	obstacle tracker	virtual compass
JNL	3120	3120	3120
CFQ	284	284	284
RMQ	1068	1084	1584
Peripheral Metadata	476	352	368
Recovery Metadata	336	294	312
Total	5280	5134	5668

The final application uses two compasses and a display to draw a virtual compass pointing towards magnetic North. The range finder is a stateless peripheral; the compass, gyro, steering servo, and motor are persistent peripherals; and the display is a historical peripheral. Each application was evaluated on three configurations, for ten seconds each: Owl without Phoenix, Phoenix with no failure, and Phoenix with a single failure.

The autonomous car uses a control loop to query the gyro and steer. It attempts to hit a specific period (30 ms) between updates, where an update consists of reading the gyro and setting the servo. Table 4 shows the minimum, maximum, and average interval lengths, which were identical across all configurations. On average, the car hit its soft real-time deadline of 30 ms. Yet, the maximum was 44 ms. This spike was caused by ill-timed runs of Owl's mark-and-sweep garbage collector.

The obstacle tracker reads the range finder and displays the distance to the nearest obstacle. Instead of aiming for a set period between updates, it sleeps for a fixed amount of time. As a result, sleep dominates the workload, and all configurations completed the same number of iterations in the allotted time.

In contrast, the virtual compass is peripheral access-intensive. It attempts to update as frequently as possible; on each iteration, it reads the compass and draws an arrow on the display. Without Phoenix, each iteration took, on average, 1005 ms. With Phoenix enabled, an iteration averaged 1862 ms. The main reason for this slowdown is that peripheral accesses are so frequent that checkpointing was nearly always enabled. This is compounded by the fact that the display is inherently slow, as a separate native write is required for each pixel.

While the performance of the applications varied due to their disparate update patterns, the space overhead was consistently small. As seen in Table 5, each application fit easily within the 512-slot journal and 64-slot control flow queue, and the maximum rematerialization queue length was three. The total space overhead, presented in Tables 6 and 7, was comparable to that of the benchmarks, requiring a maximum of 5.0 KB (virtual compass) when no failure occurred and a maximum of 5.5 KB (virtual compass) when a single failure occurred. This overhead encompasses multiple checkpoints, one per outstanding peripheral access. In contrast, a traditional checkpointing system would require a complete copy of the heap (64 KB) for each individual checkpoint.

Reliability cannot come for free; it demands tradeoffs in time and space. Phoenix optimizes for both, saving time by enabling rollback to the exact point of failure and saving space by logging only that which is absolutely necessary. Still, a time overhead is perceptible in peripheral-intensive workloads. In such workloads, the costs of improved reliability may outweigh the benefits; yet, if reliability is critical, this tradeoff may well be worthwhile. Further, this overhead could largely be eliminated by employing a hardware journal.

However, Phoenix is well-suited to applications that access peripherals periodically; these applications experienced no observable delays during evaluation. Such access patterns are more characteristic of typical embedded workloads, which periodically monitor their surroundings via sensors and react to discrete events.

7 Related Work

Fault tolerance has been well studied in distributed systems. The elements of these systems collaborate to provide a reliable service to external clients using redun-

dancy and consensus to detect, mask, and recover from failures [15, 21, 27, 28, 36]. However, each element of the system must be able to actively participate in the reliability protocols and communicate its current status. In contrast, peripherals in embedded systems are dedicated to a specific task and are unable to participate in specialized reliability protocols.

There has also been much work on protecting conventional computing systems from device driver failure. In such systems, the operating system has reliability features that allow it to tolerate device failures if the drivers behave properly. So, the focus has been on hardening device drivers and protecting the interface between the operating system and the device driver [18, 23, 24, 29, 40, 44, 45]. Phoenix instead operates at the level of hardware peripheral reads and writes, targeting systems which lack the heavier weight isolation and protection mechanisms of device drivers and conventional operating systems.

The difficulty of exhaustively addressing all possible failure scenarios in embedded applications is widely recognized [9, 11, 30, 49]. Yet, existing techniques leave much of the burden on the programmer, resulting in increased development costs and poor scalability [32]. Efforts have been made to raise the level of abstraction of writing fault-tolerant embedded applications through model- or template-driven development [11, 49] and programming language primitives [9]. However, these application-level approaches rely on the programmer to apply the correct constructs in the right places, a non-trivial task in the face of asynchronous failures.

One recovery methodology with significant traction is rollback and re-execution [6, 12, 16, 25, 35, 37, 39, 42]. State-of-the-art rollback relies on checkpoints, or snapshots, of the system state. Checkpoints are typically either taken periodically by the system, or inserted manually by the programmer [6, 12]. Traditional checkpointing algorithms maintain one or more complete copies of the memory space; while this has been successful in mobile and distributed systems [25, 35, 39], such an overhead is infeasible in a resource-constrained embedded system. Instead of taking snapshots of the entire program state, Phoenix utilizes a logging technique that resembles journaling filesystems [10, 17, 26, 38] and some hardware transactional memory proposals [34]. Though this adds an overhead to some stores, it ensures that the system only copies the subset of memory that has been changed. Further, automated disabling of logging minimizes the number of stores burdened by this overhead.

Moreover, traditional snapshots encapsulate only the internal program state, which can be restored via rollback and re-execution. In the worst case, internal state can be restored by re-executing the entire program. In fact, a full reboot is a common approach to recovery in embedded systems [14, 33]. However, critical application state is likely to be lost on a reboot. To preserve state and minimize the amount of work that is re-executed, algorithms for mobile and distributed systems have been optimized to re-execute only a subset of processes [37, 39] or threads [13, 25] based on dependencies which are either implicitly established via messages or explicitly annotated by the compiler.

Yet, minimal efforts have been made to restore external state. Past work supplementing checkpointing with message logging acknowledged the existence of external state in the form of messages, and attempted to deal with it by skipping all message sends during re-execution [37, 42]. However, this policy leaves no room to adapt to changes during re-execution.

8 Conclusions

As embedded run-time systems grow in popularity, they demand mechanisms for increased reliability. At the same time, they present new opportunities for automating reliability at the system level. This paper has presented the design and implementation of Phoenix, a novel system for surviving peripheral failures in embedded run-time systems. Phoenix is composed of integrated system- and application-level mechanisms, which work together to efficiently record the system state and automatically recover from asynchronous peripheral failures.

Several new insights motivated the design of Phoenix. In particular, peripherals interact with the real world and with each other in ways that are substantively different than internal program interactions. Based on this, one of the key innovations of Phoenix is a novel, lightweight checkpointing system to efficiently track both internal and external state. After a failure, this enables Phoenix not only to reset the internal program state, but also to restore the external peripheral state by determining whether each peripheral access must be re-executed or rematerialized.

The microcontrollers that Phoenix targets are severely resource-constrained. Phoenix guarantees rollback to the precise point at which a failed peripheral access occurred, re-executing the minimal necessary set of actions during recovery. Further, two of the three applications on which Phoenix was evaluated experienced no perceivable overhead during normal system operation. Finally, Phoenix used no more than 6 KB to log both internal and external state for these applications.

Embedded systems interact with the real world by controlling actuators based on sensory inputs. The peripherals enabling these interactions are thus fundamental components of the system and must be reliable. By providing a complete recovery process that addresses the unique challenges of resource-constrained embedded systems, Phoenix is an important step towards improving the future of writing reliable embedded applications.

References

[1] eLua. http://www.eluaproject.net/.

[2] Micro python. http://micropython.org/.

[3] The Owl embedded Python system. http://www.embeddedpython.org/.

[4] python-on-a-chip. http://code.google.com/p/python-on-a-chip/.

[5] AUSTIN, T. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the International Symposium on Microarchitecture* (1999).

[6] BARBOSA, R., AND KARLSSON, J. On the integrity of lightweight checkpoints. In *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium* (Nanjing, China, 2008).

[7] BARR, T. W., SMITH, R., AND RIXNER, S. Design and implementation of an embedded Python run-time system. In *Proceedings of the USENIX Annual Technical Conference* (Boston, Massachusetts, 2012).

[8] BASHIRI, M., MIREMADI, S. G., AND FAZELI, M. A checkpointing technique for rollback error recovery in embedded systems. In *Proceedings of the 18th International Conference on Microelectrics* (2006).

[9] BECKMAN, N., AND ALDRICH, J. A programming model for failure-prone, collaborative robots. In *Proceedings of the 2nd International Workshop on Software Development and Integration in Robotics* (Rome, Italy, 2007).

[10] BEST, S. JFS log: How the journaled file system performs logging. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Berkeley, California, 2000).

[11] BUCKL, C., KNOLL, A., AND SCHROTT, G. Model-based development of fault-tolerant embedded software. In *2nd International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation* (Paphos, Cyrpus, 2007).

[12] CHANDY, K. M., AND RAMAMOORTHY, C. V. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers C-21* (1972).

[13] CHEN, Y., GNAWALI, O., KAZANDJIEVA, M., LEVIS, P., AND REGEHR, J. Surviving sensor network software faults. In *Proceedings of the Symposium on Operating Systems Principles* (2009).

[14] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the International Conference on Embedded Networked Sensor Systems* (2007).

[15] CORBETT, J., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOOKDORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2012).

[16] CUNEI, A., AND VITEK, J. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada, 2006).

[17] CZEZATKE, C., AND ERTL, A. M. LinLogFS: A log-structured file system for Linux. In *Proceedings of the USENIX Annual Technical Conference* (San Diego, California, 2000).

[18] GANAPATHY, V., RENZELMANN, M., BALAKRISHNAN, A., SWIFT, M., AND JHA, S. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[19] GHAHROODI, M. M., OZER, E., AND BULL, D. SEU and SET-tolerant ARM Cortex-R4 CPU for space and avionics applications. In *Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale* (Avignon, France, 2013).

[20] GOPAL, K., RAMALAKSHMI, K., AND PRIYADHARSINI, C. Wide area network fault detection and routing model for wireless sensor networks. *International Journal of Communication and Computer Technologies 2, no. 7* (2014).

[21] HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference* (2010).

[22] IZOSIMOV, V., POP, P., ELES, P., AND PENG, Z. Design optimization of time- and code-constrained fault-tolerant distributed embedded systems. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition* (Munich, Germany, 2005).

[23] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Tolerating hardware device failures in software. In *Proceedings of the Symposium on Operating Systems Principles* (2009).

[24] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Fine-grained fault tolerance using device checkpoints. In *Proceeedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[25] KASBEKAR, M., AND DAS, C. R. Selective checkpointing and rollbacks in multithreaded distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems* (Mesa, Arizona, 2001).

[26] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review 40* (2006).

[27] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16* (1998).

[28] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32* (2001).

[29] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (2004).

[30] LUTZ, R. R. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering* (San Diego, California, 1992).

[31] LYONS, W. Enabling increased safety with fault robustness in microcontroller applications.

[32] MARIANI, R., FUHRMANN, P., AND VITTORELLI, B. Fault-robust microcontrollers for automotive applications. In *Proceedings of the 12th IEEE International On-Line Testing Symposium* (Como, Italy, 2006).

[33] MATIJEVIC, J., AND DEWELL, E. Anomaly recovery and the Mars exploration rovers. In *Proceedings of SpaceOps* (2006).

[34] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture* (2006).

[35] NEOGY, S., SINHA, A., AND DAS, P. K. Selective recovery in distributed systems. In *TENCON* (Chiang Mai, Thailand, 2004).

[36] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference* (2014).

[37] PARK, T., WOO, N., AND YEOM, H. Y. An efficient optimistic message logging scheme for recoverable mobile computing systems. *IEEE Transactions on Mobile Computing 1* (2002).

[38] PIERNAS, J., CORTES, T., AND GARCIA, J. M. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing* (New York, New York, 2006).

[39] PRAKASH, R., AND SINGHAL, M. Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems 7* (1996).

[40] RENZELMANN, M. J., AND SWIFT, M. M. Decaf: Moving device drivers to a modern language. In *Proceedings of the USENIX Annual Technical Conference* (2009).

[41] ROTENBERG, E. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the International Symposium on Fault Tolerant Computing* (1999).

[42] SARIDAKIS, T. Design patterns for log-based rollback recovery. In *Proceedings of the 2nd Nordic Conference on Pattern Languages of Programs* (Boahtteigi, Saariselk, Finland, 2003).

[43] SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2000).

[44] SWIFT, M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Transactions on Computer Systems 24*, 4 (2006).

[45] SWIFT, M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems 23*, 1 (2005).

[46] TABKHI, H., MIREMADI, S. G., AND EJLALI, A. An asymmetric checkpointing and rollback error recovery scheme for embedded processors. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems* (Boston, Massachusetts, 2008).

[47] TEXAS INSTRUMENTS. *Stellaris LM3S9B92 Microcontroller Data Sheet*, Oct. 5, 2012.

[48] WATTANAPONGSKORN, N., AND COIT, D. W. Fault-tolerant embedded system design and optimization consdering reliability estimation uncertainty. In *Reliability Engineering and System Safety* (2007).

[49] WILLIAMS, B. C., INGHAM, M. D., CHUNG, S. H., AND ELLIOTT, P. H. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE 91* (2003).

[50] ZHANG, Y., AND CHAKRABARTY, K. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems* (2003).

Log^2: A Cost-Aware Logging Mechanism for Performance Diagnosis

Rui Ding[1], Hucheng Zhou[1], Jian-Guang Lou[1], Hongyu Zhang[1], Qingwei Lin[1],
Qiang Fu[2], Dongmei Zhang[1], Tao Xie[3]
[1]*Microsoft Research*
[2]*Microsoft*
[3]*University of Illinois at Urbana-Champaign*

Abstract

Logging has been a common practice for monitoring and diagnosing performance issues. However, logging comes at a cost, especially for large-scale online service systems. First, the overhead incurred by intensive logging is non-negligible. Second, it is costly to diagnose a performance issue if there are a tremendous amount of redundant logs. Therefore, we believe that it is important to limit the overhead incurred by logging, without sacrificing the logging effectiveness. In this paper we propose Log^2, a cost-aware logging mechanism. Given a "budget" (defined as the maximum volume of logs allowed to be output in a time interval), Log^2 makes the "whether to log" decision through a two-phase filtering mechanism. In the first phase, a large number of irrelevant logs are discarded efficiently. In the second phase, useful logs are cached and output while complying with logging budget. In this way, Log^2 keeps the useful logs and discards the less useful ones. We have implemented Log^2 and evaluated it on an open source system as well as a real-world online service system from Microsoft. The experimental results show that Log^2 can control logging overhead while preserving logging effectiveness.

1 Introduction

Logging has been commonly adopted for monitoring and diagnosing performance issues of online service systems, such as web search engines and online banking systems. Typically, performance logs record the end-to-end execution time of a service request as well as the execution time of a component of the service system. Logging is usually achieved by instrumenting source code with logging statements and the resultant logs are stored on disks. In practice, performance logs constitute a large proportion of total logs. For example, our study of a Microsoft online service system (described in Section 6) shows that around 20%-40% of the total logs are performance logs.

Although logging is effective for performance diagnosis, it comes at a cost. Logging introduces overhead, such as disk I/O bandwidth as well as CPU and memory consumption. Intensive logging could further interfere with the service's normal execution. For example, web search engines are sensitive to performance interference from the logging system, which tends to generate huge volume of logs. Empirical results [20] show that if logging is fully conducted, the average execution time of requests in a search engine could increase by 16.3% and the average throughput could decrease by 1.48%. Therefore, it is critical to reduce the performance interference by reducing the logging overhead. In addition, our survey (See Section 2 for more details) of Microsoft engineers confirms this finding. About 80% of the survey participants confirmed that they had experienced non-negligible performance overhead caused by logging. Furthermore, intensive logging could introduce a large amount of less "useful" logs (i.e., the logs that are not useful for helping diagnose the performance issue under investigation). A study [9] on one large-scale online service system in Microsoft indicates that a high proportion of logs are useless for diagnostic purposes. Our survey of Microsoft engineers also confirms this observation.

Existing techniques for reducing logging overhead include manually removing some logging statements, changing the logging level (e.g., from "Verbose" to "Medium"), and outputting logs in a sampling fashion [20][6]. These techniques aim to reduce the number of logs to be output. However, these techniques are insufficient for several reasons. First, they cannot guarantee to preserve logging effectiveness (i.e., preserving the useful logs for diagnosis purposes). For instance, the sampling technique could miss important events due to randomness of the sampling. Second, there is no control mechanism on "whether to log" (whether or not the executed logging statement should be output) over the existing logging systems. Therefore, once developers decide "where to log", the logging system must strictly

output the logs after the execution of the placed logging statements. The resultant logs could still contain many useless ones. Finally, most of these existing techniques do not consider the dynamic properties of a running system. For a running system, the changes of workload and throughput can influence the load of its logging system. Simply using a single logging level or a sampling rate may not be able to control the logging overhead during workload spikes. Therefore, it is desirable to have a new, overhead-constrained logging system for performance diagnosis.

In this paper, we propose a cost-aware logging mechanism called *Log*^{2}. Using *Log*^{2}, developers predefine a resource *budget* allowed for logging. At runtime, the logging system decides "whether to log" such that the logging overhead is constrained under the budget while the logging effectiveness is maximized. The budget for logging overhead is defined as *logging bandwidth*, which is the maximum volume of logs allowed to be output in a time interval (such as 1KB per second). There are two reasons for choosing logging bandwidth as the budget. First, according to our survey, I/O bandwidth is the most concerning overhead in practice. Second, in general, most logging overhead such as disk storage, network I/O and CPU are directly or indirectly affected by I/O bandwidth. The logging effectiveness is measured as the percentage of performance issues that can be captured by the resultant logs.

There are three challenges for realizing such a cost-aware logging mechanism:

- It should be able to control logging overhead while preserving logging effectiveness.
- It should incur low additional overhead such as CPU and memory consumption.
- It should provide flexibility for developers to configure it for different service scenarios, and should be able to adapt to environmental changes dynamically.

To address the above challenges, *Log*^{2} introduces a two-phase filtering mechanism. In the first phase, a large number of irrelevant logs are discarded efficiently. In the second phase, useful logs are cached and output while complying with the logging budget. The two-phase mechanism is updated dynamically to address all the challenges.

We evaluate *Log*^{2} on BlogEngine, which is a popular open source blogging platform. Furthermore, we perform an evaluation of *Log*^{2} using real logs of ServiceX, which is a large-scale online service system from Microsoft. The evaluation results confirm that *Log*^{2} is effective and practical in real-world scenarios.

This paper makes the following main contributions:

- We propose a novel cost-aware logging mechanism *Log*^{2}, which helps achieve a balance between logging overhead and effectiveness. Such a mechanism incurs low additional overhead and is flexible.
- We design and implement *Log*^{2}. We also evaluate *Log*^{2} on both a open source system and a large-scale online service system from Microsoft.

The rest of the paper is organized as follows. Section 2 describes a survey of logging practice in Microsoft, which motivates the design goals of *Log*^{2} described in Section 3. Section 4 describes the design and implementation of *Log*^{2}. Section 5 provides the detailed evaluation of *Log*^{2} on an open source system. Section 6 describes a case study on Microsoft ServiceX system. We discuss the limitations and future work in Section 7. Section 8 introduces the related work, and Section 9 concludes the paper.

2 A Survey of Logging Practice in Microsoft

To better understand the current logging practice, we conducted a comprehensive survey among hundreds of engineers from five product teams in Microsoft. We received responses from 84 engineers. According to the survey, 81 out of 84 respondents are "expert" or "knowledgeable" to logging systems. The survey aims to understand the participants' experience in logging systems and logging overhead. The details of survey questions are available online [4].

In general, the logging systems used by Microsoft engineers fall into three categories, including (1) internally developed systems that directly output the executed logging statements via a language-intrinsic component or a wrapped API; (2) ETW logging [2], which writes the buffered logs in a batch fashion, and (3) sampling-based logging tools that are mainly designed for large-scale online services sensitive to logging overhead.

2.1 Logging Overhead

According to our survey, 80% of the participants agreed that logging overhead is a non-negligible issue. The top three most commonly concerned types of overhead are storage (60%), I/O bandwidth (58%), and CPU usage (56%). Among the participants, 59% of them have suffered from the consequences incurred by the logging overhead. Table 1 shows some of the experiences reported by the surveyed engineers.

The top three most widely used approaches to control the logging overhead include adjusting the logging level (93%), manually removing unnecessary logs (64%), and

Table 1: Some of the experiences of the logging overhead

Category	Reported Experiences
Disk I/O bandwidth	Overuse of I/O *caused perception of interference with core functionality.* The bandwidth requirement by *enabling all logs is 8MB/s, which however should be* $\leq$ *200KB/s.*
Storage	*OS slows down*, other process that needs disk space *may crash and even logging system could crash.* Storage is a *critical component that may cause system crash, but it is often overlooked.*
CPU	*Service is slowed down significantly once the CPU usage of logging is increased to double digits.* CPU usage of logging is *very sensitive to our super-efficient system.* *3%-5% is the upper bound for CPU usage of logging.*
Memory	Unexpected increases of memory usage of logging system *was the root cause of one service incident.* Memory leak of logging system *caused days of efforts on debugging.*

archiving log files periodically (43%). However, about 65% of the participants replied that they are not satisfied with the existing approaches. For instance, removing logs by changing source code requires extra efforts on re-compiling, testing, and re-deployment. Archiving log files is often expensive because a large volume of data needs to be transformed via network. All these existing approaches are considered to be after-thoughts, and are applied only when logging overhead starts to compromise the system quality.

About 83% of the survey participants also agreed that many log messages are redundant for diagnosing performance issues, implying the feasibility to reduce logging overhead while preserving sufficient logging effectiveness. In addition, about 43% of all participants agreed that logging overhead needs to be controlled, and they considered resource budget for logging in their work.

2.2 Other Limitations of Existing Logging Systems

A number of participants also shared with us additional limitations of the existing logging systems and expressed the needs for a cost-aware logging mechanism. These comments and suggestions strongly motivated the design of Log^2:

Lack of cost-awareness during log instrumentation. One participant complained about the lack of cost-awareness during log instrumentation. He noticed that some developers often had little idea about the resulting logging overhead when they planned to instrument source code with new logging statements. A typical bad logging practice is to insert logging statements in tight loops (i.e., the loops which iterates intensively), which could cause high overhead, especially in I/O throughput and storage. He suggested a logging system for controlling the logging overhead transparently, so that developers can perform log instrumentation without worrying about the overhead incurred.

Burden in log analysis. One participant commented that too many logs make it challenging to analyze logs via manual inspection. It would be helpful if a logging system can collect all possible logs but do not flush all of them. He also suggested a potential solution: logging system should flush the logs only when some predefined rules are violated.

In summary, the survey results motivate a new overhead-constrained logging mechanism as we propose in this paper.

3 The Design Goals of Log^2

3.1 Cost-Aware Logging Mechanism

In this paper, we propose Log^2, a cost-aware logging mechanism that constraints logging overhead. Using this mechanism, developer can perform logging by instrumenting their programs, and predefine a resource *budget* for logging. With the given *budget*, the logging mechanism decides "whether to log" for each logging request at runtime, makes sure that the logging overhead complies with the predefined *budget*, and maximizes the logging effectiveness at the same time. In addition, the logging mechanism can support on-the-fly budget setting. Therefore, the logging mechanism not only provides developers with the flexibility to strike the balance between logging overhead and effectiveness, but also provides the flexibility to configure different logging budgets for different service scenarios, or even the flexibility to dynamically configure the logging budget. Furthermore, such a cost-aware logging mechanism enables better planning of maintenance resources [5], as the logging budget can be determined in advance.

3.2 Design Goals

Log^2 is designed to realize such a cost-aware logging mechanism. The budget for logging overhead in Log^2 is defined as *logging bandwidth*, which is the maximum volume of logs allowed to be output in a time interval. Logging bandwidth is the most concerning logging overhead according to engineers' feedback. It is also the most

representative logging overhead, because other types of logging overhead such as disk storage, network I/O and CPU are often directly or indirectly affected by the logging bandwidth.

We have identified four design goals for Log^2, which are listed below:

Cost-effectiveness: Log^2 should be able to achieve an optimal balance between logging overhead and effectiveness. The logging overhead, defined in terms of log bandwidth, should be constrained under the budget. Although the logging budget is under constraint, logging effectiveness cannot be compromised, i.e., with respect to performance diagnosis, the number of performance issues detected by the reduced number of logs should be similar to the number of issues detected by the total number of logs. In Log^2, a ranking score named *utility score* is defined to measure how much utility each logging request contributes to performance diagnosis. Log^2 then selects the top-ranked logging requests and outputs them. Other logging requests are filtered away. More details are described in Section 4.3.

Low additional overhead. Log^2 should incur low additional overhead. The additional overhead brought by runtime decision on "whether to log" (i.e., CPU usage and memory consumption) should be negligible. The design choices of Log^2 for minimizing CPU usage and memory consumption are described in detail in Section 4.4.

Scalable. Log^2 should be scalable to the number of logging requests. It is very common that thousands of requests are processed per second, and considering that many logging statements are executed when serving one single request, the scale of the *logging requests* per second is large. A traditional logging system, which makes centralized decision, suffers since such centralized decision can delay the logging time as well as increasing the corresponding memory buffer usage. In contrast, Log^2 includes a two-phase filtering design to avoid the potential bottleneck. The details are described in Section 4.

Flexible. Log^2 should provide developers with the flexibility to configure the system. First, Log^2 provides several types of predefined *utility scores*, which are designed for the most common diagnostic scenarios (to be described in Section 4.3.1). It also allows developers to configure a user-defined function for computing utility scores. Such flexibility enables Log^2 to tackle various types of performance issues. Second, the budget can be configured on-the-fly. Such on-the-fly configuration enables developers to select a proper *logging bandwidth* according to the different resource plans in different scenarios. Since there is no one-fit-for-all configuration for all kinds of services, such flexibility is crucially important for wide adoption in different scenarios. More details are described in Section 4.3.2 and Section 4.4.2.

```
1  Log2.Begin(string McrName, ...); //begin
2  DoSomething();
3  Log2.End(string McrName, ...);   //end
```

Figure 1: Logging API in Log^2.

4 Design and Implementation of Log^2

This section illustrates the detailed design and implementation of Log^2. We first discuss the high level workflow of Log^2, and then illustrate its two core components, namely local filter and global filter. These core components are essential for achieving the goals of Log^2.

4.1 Logging Requests

For performance diagnosis, developers can specify an area of code that should be monitored and logged. We call such an area of code Monitored Code Region (MCR). Examples of typical MCR include:

- Expensive system-level APIs, such as operations on I/O, database, networking, etc.
- Loop blocks. Previous work [13] found that a significant portion of real-world performance issues are caused by inefficient loops.
- Function calls cross application-level component boundaries, such as RPC or the connection between GUI and backend services.

Performance logs should record two timestamps at the beginning and end of a MCR, which are sufficient to compute the execution time of the MCR. Log^2 provides two logging APIs, *Begin* and *End*, to denote the beginning and end of an MCR, respectively. The APIs compute the execution time of an MCR and also record the unique ID of the MCR. Figure 1 depicts the logging API usage in Log^2, where the execution time of *DoSomething* is recorded. A pair of logs *Begin* and *End* form a *logging request*, which will be further processed by Log^2 to decide whether they should be filtered or output.

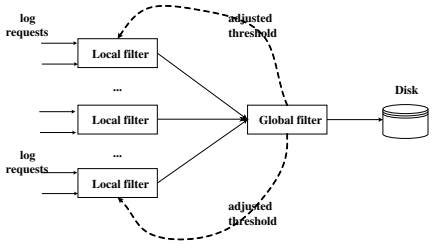


Figure 2: The workflow of Log^2.

4.2 Overall Workflow

The workflow of Log^2 is depicted in Figure 2. Two filtering phases, local filter and global filter, are adopted to decide whether or not the incoming logging requests should be logged (whether to log). Such a two-phase filtering mechanism is used to avoid the potential bottleneck of a single centralized filter, when a huge number of log requests come in simultaneously. The local filters are responsible for discarding the trivial logging requests, which are logging requests that have low utility scores. The global filter is responsible for flushing the top ranked logging requests to disk and in the meantime complying with the logging budget.

Each thread of logging requests has a local filter. Only the logging requests with *utility scores* (which are calculated dynamically) higher than a global threshold can pass through the local filter to a memory buffer in the global filter. Other logging requests are discarded.

The global threshold is adjusted dynamically, to adapt to environment dynamics, while optimizing the effectiveness and efficiency of Log^2. Usually, a significant high portion of logging requests are discarded in the first phase. In the global filter, the final decision on log outputting is made periodically to make sure that the budget constraint is compliant. The logging requests from all local filters during the last time window are cached in memory. When a periodic event is triggered, the cached logging requests are sorted according to their utility scores. Only the top-ranked requests with total volume equal to the logging budget are flushed to disk. Meanwhile, the global threshold for utility scores is updated by the global filter by considering the volume of logging requests in recent time intervals. Lastly, the global filter feeds the new threshold back to each local filter.

Details about each component are described in the following subsections.

4.3 Local Filter

The major task of the local filter component is to compute the utility score for each logging request. The utility score measures the usefulness of a logging request for performance diagnosis. Note that a local filter is executed in the same service thread being monitored. The overhead for computing utility score should be kept low to reduce the impact on the service.

4.3.1 Formula of utility score

To compute the utility score for each logging request, we analyze the histogram of the execution time of the corresponding MCR. The intuition is that the utility score should be higher if the execution time of a MCR deviates further away from its past behavior. For each MCR, we can measure the degree of performance deviation based on the histogram of the execution time of the MCR. However, it is inefficient to maintain the complete history of execution time for each MCR and compute the histogram. In our work, we adopt the concept of *method of moments* [15], which can be efficiently computed. According to statistical theory, *moments* can well approximate histogram [10]. The 1-order of *moment* is mean, and the 2-order of *moment* (σ^2) is the square of *standard deviation* (σ).

Based on the *mean* (μ) and the *standard deviation* (σ) of execution time of an MCR, we propose three forms of utility scores, given the current execution time t of the MCR:

$$utility = \frac{t-\mu-\tau}{\sigma} \tag{1}$$

$$utility = t \tag{2}$$

$$utility = t-\mu-\tau \tag{3}$$

In Equation (1), a constant value τ is a tolerance factor, which is used to further reduce false-positives for MCRs. For example, execution time of $5ms$ is significantly abnormal compared to $1ms$ as the average execution time, but is ignorable for performance diagnosis. The default value of τ is $25ms$.

Equation (2) simply uses the execution time as the utility score, which is suitable when the users would like to identify performance hotspots (e.g., those components with the longest execution time). Equation (3) computes utility score based on the mean execution time. Compared with Equations (1) and (2), it considers the abnormality (t-μ) while ignores the fluctuation.

Besides the predefined utility formulas, we also allow users to specify their own utility functions to cater for their own scenarios.

4.3.2 Updating the utility scores dynamically

During performance monitoring, the execution time t of each MCR varies at runtime. Therefore, the *mean* and *standard deviation* of t should be updated dynamically over time. *moments* can be updated incrementally, with the time complexity of $O(1)$:

$$\mu_n = (1-\frac{1}{n})\mu_{n-1} + \frac{1}{n}t_n \tag{4}$$

$$\sigma_n^2 = (1-\frac{1}{n})[\sigma_{n-1}^2 + \frac{1}{n}(t_n-\mu_{n-1})^2] \tag{5}$$

where n denotes the n^{th} update; t_n is the n^{th} execution time.

We also modify the Equations (4) and (5) in a manner similar to *Exponential Smoothing*[11]. Exponential Smoothing can better capture the slow-varying system dynamics. The corresponding formulas are as follows:

$$\mu_n = (1-\alpha)\mu_{n-1} + \alpha t_n \tag{6}$$

$$\sigma_n^2 = (1-\alpha)[\sigma_{n-1}^2 + \alpha(t_n - \mu_{n-1})^2] \quad (7)$$

where α is a weighting factor, which is empirically set to 0.01.

4.4 Global Filter

In *Log*^{2}, the global filter component performs two major tasks: log flushing and utility-threshold adjusting.

4.4.1 Log flushing

Log flushing is triggered periodically, and such period is called *flush interval*. When the timer is triggered, *Log*^{2} first sorts the buffered logs according to the utility score, and then flushes the top ranked logs so that the total flushed log volume does not exceed the logging budget. All selected logs are packed together and are flushed once in a batched fashion.

Buffer design. Proper buffer design is important for reducing logging overhead, especially for reducing CPU usage. Note that the buffer will be accessed by multiple local filters with fast inserting operation, as well as the global filter thread with slow sorting and flushing operations. To make sure that the latter one does not affect the inserting performance and thus does not block working threads, *Log*^{2} includes a data structure called *swap buffer*, which has two buffers: one serves for inserting operation, and the other serves for sorting and flushing operations. These two buffers are swapped periodically after a flush interval. A 0/1 flag is used to indicate which buffer is currently used for insertion, and which one is for flushing. Such mechanism guarantees that the two threads work on different buffers without lock contention except swapping the global flag.

Flush-interval selection. Long flush interval would result in larger swap buffer, and thus more memory consumption; while shorter interval benefits less from batched flushing, and incurs frequent overhead in swapping buffers. *Log*^{2} currently sets the default flush interval to 30 seconds, which works well in our experiments and practice. Users are also allowed to configure the flush interval on-the-fly.

4.4.2 Utility threshold adjustment

The utility threshold is used to control the volume of logs to be inserted into the *swap buffer*. Because only the logging requests with utility scores larger than the threshold is cached, setting a proper threshold is very important for *Log*^{2}. Specifically, if the threshold is set too low, massive logs could be inserted into the *swap buffer*, the consequence is larger overhead. On the other hand, if the threshold is set too high, only a small amount of logs could be cached in the buffer, thus the important logs could be missed, leading to unacceptable logging effectiveness.

The optimal objective is to cache just budget-volume logs by selecting a proper threshold. Choosing such an optimal threshold value in one-shot in unrealistic, because either the environment dynamics or the frequency of different utility scores is unknown. To address this challenge, we design an iterative way for adjusting the threshold by 'learning from history'. The duration of each iteration is called *adjust interval*. Intuitively, when the volume of logs in the previous *adjust interval* is higher than the budget, then the threshold should be increased. The threshold should be decreased when the volume of logs in the previous *adjust interval* is lower than the budget. From both effectiveness and efficiency perspectives, it is desirable that the adjusting algorithm should converge quickly, and the volume of logs in the buffer should not be too large (low overshoot [19]) in any interval. We next illustrate the details of *Log*^{2}'s threshold-adjustment algorithm, which is agile and has low overshoot.

Adjustment mechanism. Let us denote the threshold and log volume as T_n and V_n, respectively. Here n is the index of the *adjust interval*. Let us denote B as the logging budget. The threshold adjusting mechanism used in *Log*^{2} is as follows (in the form of Secant Method [18]):

$$T_n = T_{n-1} + (V_{n-1} - B) \times \frac{T_{n-1} - T_{n-2}}{V_{n-1} - V_{n-2}} \quad (8)$$

Mathematically, the convergence of our algorithm is super-linear, with an order of 1.618 [18]. More details about the mathematical deduction of our method are available at our project website [4]. The interpretation is that the 'gain' $T_n - T_{n-1}$ on the threshold is proportional to 'error' $V_{n-1} - B$, and coefficient $\frac{T_{n-1}-T_{n-2}}{V_{n-1}-V_{n-2}}$ approximates the reciprocal of the derivative, if we treat V as a function of T.

In our implementation, to avoid a divide-by-zero error, we add 1 if $V_{n-1} - V_{n-2}$ is close to 0. When $T_{n-1} - T_{n-2}$ is equal to zero, threshold updating can trap to a certain number and never changes. To avoid such issue, we add a very small value (0.01) under such situation.

Adjustment interval. To make the threshold adjustment mechanism more effective, a properly chosen *adjustment interval* is needed. The adjustment interval should mitigate the fluctuation of environment change, i.e., the workload varies slowly under the granularity of the chosen adjustment interval. Therefore, the adjust interval cannot be too short; otherwise, the transient random variation of workload will be significant, On the other hand, a too long interval indicates longer time for convergence, making *Log*^{2} less agile. In our implementation, *Log*^{2} sets the adjust interval to 30 seconds, which is the same as the *flush interval*.

4.5 Implementation Details

We have implemented Log^2 using the C# language. Some details about the implementation are as follows.

Bounded memory usage. The maximum memory usage of Log^2 is set to 50MB in configuration, so that Log^2 has negligible memory contention with normal service operations. In our implementation, when the maximum memory usage is reached, new *logging requests* will be dropped in the same flushing interval. In fact, 50MB is rarely reached in most cases. Specifically, two components in Log^2 consume most memory usage. One is the cache for maintaining μ and σ for all the MCRs. For a large-scale online service, the number of MCRs is in a magnitude of 100,000, so the corresponding memory usage is 100,000 $\times$ 2 $\times$ 8B = 1.6MB. The other component that consumes most memory usage is the swap buffer. Its size depends on both the budget size and flush interval. The I/O bandwidth of logging is 200KB/s (which is 20GB per day!) per machine for a typical large-scale online service. Because the budget size does not exceed the overall throughput, a much loose upper bound of memory usage on the swap buffer is 200KB/s $\times$ 60s $\times$ 2 = 24MB. In addition, the 50MB threshold has not been reached in all of our experiments.

Handling system idle time. System idling is a special circumstance that needs to be handled. Specifically, when *logging requests* are rare, the budget will not be reached no matter how the utility threshold is adjusted. The consequence is that the utility threshold could become extremely low, and thus the system will overshoot dramatically (i.e., there will be a burst of flushing) when the intensity of *logging requests* turns back to normal. In order to avoid such circumstances, a lower bound on the *adjust interval* is set. In our implementation, we set the lower bound to 0. Such mechanism is commonly used in the area of control engineering [8].

Nested instrumentation. To support nested instrumentation, it is noteworthy that each local filter actually maintains a timestamp stack to match the logging begin-end pair, When a *Begin* is invoked, the corresponding timestamp is pushed into the stack; and when an *End* is invoked, the top element in the stack is popped, and is matched as the *Begin* corresponding to the current *End* invocation. As illustrated in Section 4.3.2, the historical information of each MCR is maintained separately, therefore, dropping the outer log request will not directly lead to the dropping of the inner log request.

5 Evaluation

In our evaluation, we intend to evaluate Log^2 from the following three aspects:

Logging throughput: How much I/O throughput (the volume of logs flushed to disk within a time interval) can be reduced by Log^2, compared with the existing logging system?

Logging effectiveness: How effective is Log^2 in diagnosing performance issues? The effectiveness is measured as the percentage of performance issues that can be captured by the flushed logs.

Additional overhead: How much additional CPU and memory overhead is incurred by Log^2?

5.1 Experimental Subject and Setup

To evaluate Log^2, we design experiments on BlogEngine [1], which is a popular open-source, ASP.NET based blogging platform. BlogEngine has received more than 1,000,000 downloads as of January 30, 2015. It supports various blogging activities, such as writing blogs, adding comments, sharing, and following. We choose the version 2.8, as it is a recent stable version.

To evaluate Log^2 on BlogEngine, we run the BlogEngine as a service, and we simulate concurrent access to the service via multiple synthetic users. We then analyze the logs generated by Log^2 as well as the runtime performance. We set up the experiment on BlogEngine with four steps: instrumentation, deployment, performance issue injection, and overhead monitoring. Below are the detailed setup procedures.

Instrumentation. We perform program instrumentation guided by previous work [14] [13]. Specifically, three types of code regions in BlogEngine are marked as MCRs and logged, since they have relatively high potential to cause performance issues. These three types of MCRs include expensive system-level APIs, loop blocks, and function calls. In summary, about 1000 MCRs are identified and instrumented.

Deployment. We use one physical machine to deploy the BlogEngine service, and two other physical machines are configured as client nodes. Each machine runs Windows Server 2012 R2, with CPU Intel(R) Xeon(R) E5-2650 v2 @ 2.60GHz (2 processors) and 192GB Memory.

We adopt a tool named WebTest [3] to simulate high workload from multiple synthetic users to access the BlogEngine service. WebTest is a new testing tool released with Visual Studio 2012. It can be configured to generate mixed types of requests with user-specified loads. In our experiment, we generate five typical types of requests in WebTest - *read blogs, write comments, search, download files* and *upload files*. These requests cover the most common usage scenarios of BlogEngine.

Performance Issue Injection. In order to evaluate the logging effectiveness of Log^2, we inject three types of performance issues, namely *upload an extremely large file*, *search a strange term*, and *exhaust CPU by other process*. Specifically, when uploading a file with size

larger than 100MB, the GUI on the client side starts to hang (a possible fix is to put the uploading job in a back-end thread). The response to the search operation becomes significantly slow when entering a strange query term that is long and contains special characters (a possible fix is to pre-process the query term). Both of these two performance issues can be directly pinpointed by the corresponding logs.

We write a program named *ResourceEater* to consume high CPU usage in a certain period to mimic the third type of performance issues. When ResourceEater is launched, it occupies CPU intensively. The runtime performance of BlogEngine degrades significantly. Such performance issues can be reflected in the corresponding logs (e.g., the logs that mark the loop blocks).

Overhead monitoring. To measure the I/O throughput, we record the number of logs flushed to disk per time interval. To measure the additional CPU/Memory overhead of Log^2, we write a program named *PerfMonitor* to periodically monitor the CPU and memory usage of BlogEngine at every second. The CPU overhead is measured as the percentage of total CPU cycles Log^2 occupies, and the memory overhead is measured as the bytes of memory space Log^2 consumes.

5.2 Experimental Design

We design an experiment to evaluate Log^2. We use the WebTest tool [3] to simulate 101 synthetic users concurrently accessing BlogEngine. The experiment runs for two hours. Among the 101 users, 100 users mimic the normal user behaviors, which fall into the five aforementioned groups (*read blogs, write comments, search, download files* and *upload files*). One user mimics the abnormal usage to inject two types of performance issues (*upload an extremely large file* and *search a strange term*), which are generated 78 times during the 2-hour experiment.

To inject the issues caused by *exhausting CPU by other process*, the ResourceEater is triggered on the service machine one hour after start, and lasts for 10 minutes.

We also evaluate the logging effectiveness of Log^2 using three utility scores: t, $(t-\mu-\tau)$, $(t-\mu-\tau)/\sigma$, respectively.

In the experiment, we compare Log^2 with the baseline approach, which directly outputs all executed logs without considering cost-effectiveness. As we instrument all the interested MCRs, the baseline approach is able to detect all injected performance issues. We are interested in knowing how Log^2 can detect similar number of issues using fewer amount of logs.

In addition, we compare Log^2 with two sampling-based logging approaches, named Sampling-counter and Sampling-time, respectively. Sampling-counter is counter-based, which uses a global counter to record how many logging requests are processed. Only the logs whose corresponding counter is divisible by the reciprocal of the sampling rate are flushed to disk. Sampling-time is time-interval based, which uses a timer to control when the logs are flushed to disk. Only the logs executed when the timer is triggered are flushed.

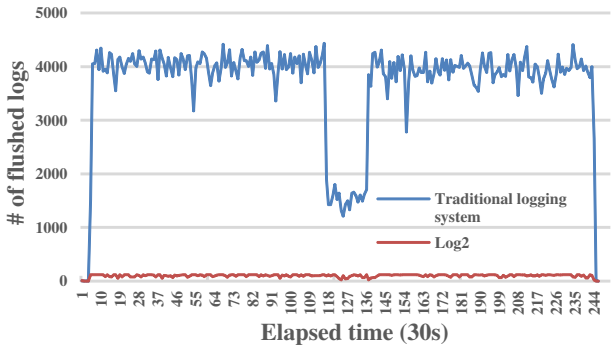


Figure 3: Comparison of logging throughput. (budget = 120 logs/interval)

5.3 Experimental Results

Logging throughput. Figure 3 shows the number of logs flushed per time interval (30s) using Log^2 and the baseline logging approach, respectively. The budget is set to 120 logs/interval. The big drop on the number of logging requests (around interval 118-136) is due to the launching of *ResourceEater*.

Figure 3 shows that the logging throughput is significantly reduced using Log^2. The average number of logs flushed per interval is 104 for Log^2, while it is 3,800 for the baseline logging approach. The reduction on logging throughput is over 97%. In addition, the logging throughput of Log^2 strictly complies with the budget constraint ($<$ 120 logs/interval).

Logging effectiveness. The logging effectiveness is inherently associated with the budget size, i.e., the logging bandwidth. Higher logging bandwidth would induce higher logging effectiveness. We evaluate the logging effectiveness by varying the budget size. In addition, we also evaluate three alternative formulas of utility scores (t, $t-\mu-\tau$, and $(t-\mu-\tau)/\sigma$).

Figure 4 illustrates how the logging effectiveness increases as the budget size increases. All the three proposed utility scores help achieve high effectiveness, i.e., the coverage of marked logs increases quickly to almost 100% when the budget size starts to increase. The results indicate that Log^2 has strong ability to preserve high logging effectiveness while reducing a significant amount of logs.

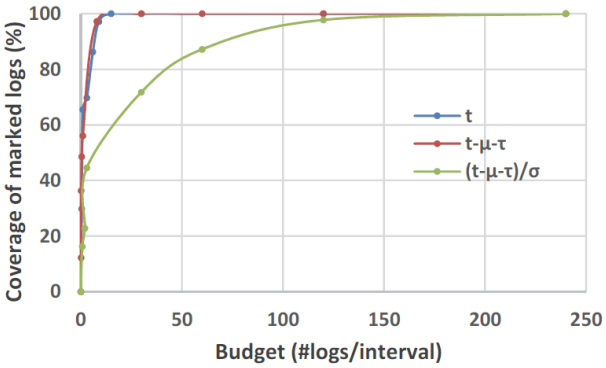


Figure 4: Logging effectiveness vs. budget size

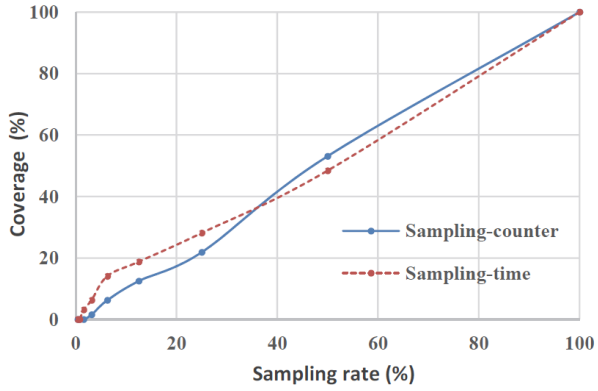


Figure 5: Logging effectiveness of two sampling-based approaches

The results of two sampling-based logging systems, Sampling-counter and Sampling-time, are illustrated in Figure 5. The effectiveness of either Sampling-counter or Sampling-time is approximately proportional to the sampling rate, which is much lower than what Log^2 achieves. It is worth noting that the budget size of 120 logs/interval is equivalent to the sampling rate of 3%. While Log^2 achieves almost 100% coverage with such budget size, Sampling-counter and Sampling-time achieve only 2% and 6% coverage, respectively.

For the issues injected by *exhausting CPU by other processes*, there are in total 690,000 individual calls on 6 instrumented loop blocks during the experiment (note that each loop block is one MCR). By using Log^2, only 22,000 (97% reduction) calls on loop blocks are recorded (budget size = 120 logs/interval), with the average execution time of 160ms. By inspecting the loop-related logs, we found that the average execution time is 423ms when ResourceEater is launched, which is significantly larger than the average value (160ms) without the impact of ResourceEater. Our inspection shows that the logs reflecting loops with long execution time are recorded, which demonstrates the capability of Log^2 to detect the performance issue due to exhausted CPU usage.

In summary, the experimental results show that Log^2 is effective in detecting performance issues, while keeping the volume of logs low.

Additional overhead. Log^2 works in the same process of the BlogEngine service, hence its own CPU/Memory usage cannot be measured directly. In order to evaluate the overhead of Log^2, we measure the overall CPU/Memory usage of the BlogEngine system integrated with Log^2, and compare it with the overall usage of the BlogEngine system integrated with the baseline logging approach (outputting all logs). We run the experiment with each setting 7 times to overcome random variations.

Table 2: Comparison on overall resource usage

Logging system	Memory(GB)	CPU(%)
Log^2	4.74±0.21	63.4±3.0
Baseline	4.70±0.25	70.6±4.1

According to Table 2, the additional memory usage of Log^2 over the baseline approach is not noticeable. When integrated with Log^2, the average CPU usage of BlogEngine is slightly lower than that with the baseline logging system. This is because using Log^2, a large number of logging requests are discarded at early stage, therefore a significant amount of processing (such as logging state extraction or string conversions) as well as lock contention are avoided, leading to reduced CPU usage.

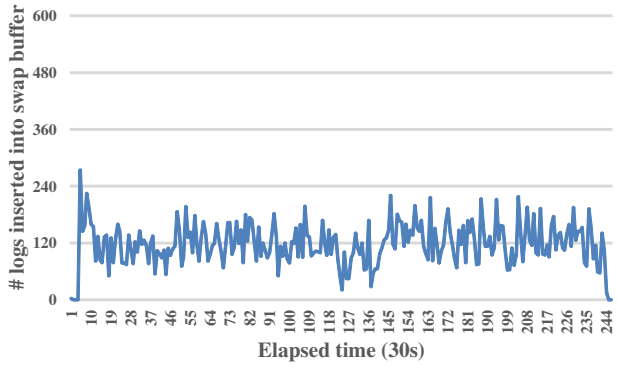


Figure 6: Dynamics of swap buffer size

In order to evaluate the memory usage of Log^2, we monitor the size of the swap buffer over time. Figure 6 shows the number of logs inserted into the swap buffer per flush interval. There is one peak at the beginning, when the threshold for the utility score is not converged. The peak is about 1.3 times higher than average, which is far from the default maximum memory limit set in Log^2. In addition, it takes only five iterations to converge, which shows that the small memory peak disappears quickly. The variation of the curve is mostly caused by the randomness in the workload.

6 An Application to Microsoft ServiceX

To further evaluate Log^2, we have applied it to analyze the performance logs of Microsoft ServiceX (the service name is anonymized due to confidentiality). ServiceX is a large-scale online service system, serving millions of users globally.

Designed with a 3-tier architecture, ServiceX is run on a large number of machines, each of which continuously generates huge amount of logs. A typical front-end machine usually generates logs with a speed of 30MB per minute. Log aggregation from all the machines is a heavy task, since each machine generates about 40GB logs every day. ServiceX provides a logging API called MoS for performance diagnosis. The corresponding logs are called MoS logs (i.e., performance logs), which take up 20%-40% of the total logs. Engineers of ServiceX would like to reduce the large volume of MoS logs, since most of them are not useful for performance diagnosis and they simply incur overhead.

We apply Log^2 to evaluate its ability to reduce the volume of MoS logs.

Setup. Each MoS log entry contains the following information: log time, execution time of the MCR, code region ID, and thread ID. Such information is sufficient to re-construct the execution flows of all the MoS logs. We randomly select 12 different datasets. Each dataset contains logs generated during one continuous hour.

We focus on evaluating logging bandwidth and effectiveness in our study. To do so, we identify performance hotspots, which are the code regions that take most time to execute. We choose the MoS logs having the top 0.3% (i.e., 1 - 99.7%, which is a 3-sigma rule of thumb [22]) longest execution time as the performance hotspots. We then apply Log^2 to see how many of these performance hotspots can be successfully identified. We choose t as the utility formula. We evaluate logging effectiveness as the coverage of the performance hotspots by varying the budget size. Additionally, we also evaluate how the flush interval affects the effectiveness.

Results. Figure 7 shows the logging effectiveness of Log^2 by varying the budget size. Since we conduct experiments on 12 datasets, the effectiveness on each budget is represented by a range. As shown in Figure 7, the coverage of performance hotspots quickly comes up to 100% when the budget size increases. Particularly, when the budget is set to 100 logs/interval, which is equivalent to the sampling rate of 0.77%, the coverage is already 98%. On the other hand, only 4.5 MB logs are recorded, while the size of original MoS logs is 500MB for each dataset.

Figure 8 shows the effectiveness of Log^2 under different flush interval values. Here the budget is set to 120 logs/interval, which is equivalent to the sampling rate of 1.0% . When the flush interval is very small, the coverage rate is relatively low, mainly due to the significance of randomness on the workload. Setting the flush interval to 30 seconds is satisfactory, since the coverage rate here is almost 100%.

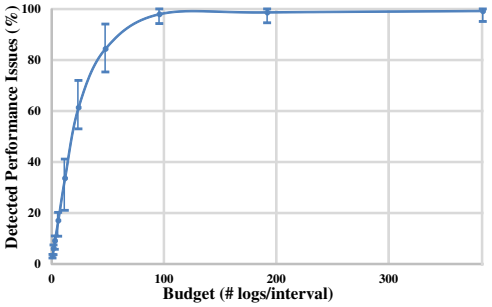


Figure 7: Logging effectiveness vs. budget

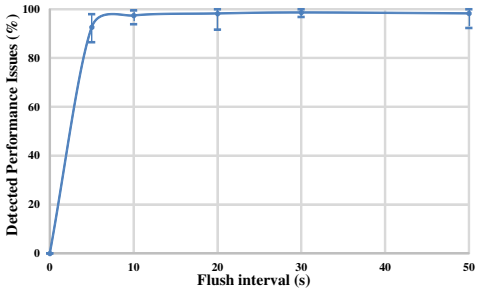


Figure 8: Logging effectiveness vs. flush interval

In summary, our case study on ServiceX has confirmed the applicability of Log^2 to real-world systems.

7 Discussion

Budget control for multiple services. In our current design, Log^2 is implemented as a runtime logging library and can be dynamically linked to a service system under monitoring. It controls the budget for only one single service. As budget can be changed dynamically, it is possible to make Log^2 a standalone process, which manages a set of budgets for multiple services. Such a centralized budget control system can further enable dynamic budget re-allocation to different services.

Supporting more types of performance analysis. Log^2 is very effective for capturing performance hotspots on-the-fly. In practice, there are other commonly required types of performance analysis. For example, to understand the overall latency status of the system under monitoring, the total number of times the latency hits the 3-sigma threshold, the average latency of a component, and so on. Log^2 has the ability to provide such information. For example, Log^2 maintains the *mean*

and *standard deviation* values for each MCR. In addition, Log^2 can record how many times each MCR is updated. Hence, many other measures of performance status (such as the 3-sigma measures) can be easily derived from these basic statistics. Additionally, all the data of Log^2 can be dumped periodically and used by other performance analysis tools. Analytical reports based on the off-line processing of the data can produce comprehensive information for postmortem analysis.

Multiple objectives. Currently we only define budget in terms of I/O bandwidth, as it is mostly concerned by our surveyed participants. It is possible to consider more objectives, such as CPU and memory usage, and to control logging overhead by performing multi-objective optimization. We will address it in our future work.

Where to log. As described in Section 4.1, we identify MCRs for performance diagnosis. In this paper, we focus on the problem of "whether to log". Another important topic is "where to log", i.e., the automatic identification of code regions that should be logged and monitored. The two problems, "whether to log" and "where to log" are closely related to each other. For example, the logging mechanism we proposed enables "conservative logging", i.e., developers can instrument a large amount of logging statements without concerning about the logging cost. This is an important topic of our future work.

Leveraging non-performance logs. Although performance logs are common in practice, there are also other types of logs such as those for failure diagnostics. Two adjacent log entries indicate the time spent on executing code between the two log entries. It would be interesting to leverage those logs for performance diagnosis.

Extension to failure diagnosis. Our current work focuses on analyzing performance logs for effective and efficient monitoring and diagnosis of performance issues. Apart from performance logs, there are other types of logs such as logs recording error and failure information. These logs are mainly for diagnosing software failures in production environment [24, 26, 27]. How to extend our work to support failure diagnosis is important future work.

8 Related Work

Performance monitoring and diagnosis has becoming increasingly important, especially in the era of Internet-based services and cloud computing. A large amount of research has been conducted to characterize [13, 28, 16] and improve system performance [14, 21, 23, 12, 7].

In production environment, logging is still the most commonly used technique for performance monitoring and diagnosis. Dapper [20] is a large-scale distributed tracing infrastructure widely adopted by Google for ubiquitous and continuous monitoring. Dapper is designed to have low overhead, application-level transparency and scalability. Log^2 shares the same design goals with Dapper, and goes one-step forward with finer-grained and more accurate control on logging overhead to comply with the resource budget. Dapper flushes only a fraction of all traces using a sampling (with a manually configured sampling rate) approach such that interesting traces could be missed. Log^2 preserves useful logs with significantly higher effectiveness. At the same time, Log^2 guarantees the resource budget constraints, which can be violated in Dapper.

ETW (Event Tracing for Windows) [2] is a framework that can log Windows kernel or application-specific events to a log file. It has a buffering mechanism that reduces the number of disk accesses for logging. However, ETW is not cost-aware: it cannot selectively record a number of logs based on a given budget.

Paradyn [17] also controls its instrumentation overhead dynamically. However, it depends on users to explicitly configure *where to log*, and predict *whether to log*. Log^2 instead is user-transparent in that *whether to log* decisions are dynamically made by the logging mechanism. Excessive instrumentation is commonly adopted in the profiling domain. Matthew [6] presents sampling based low-cost instrumentation to enable feedback-guided just-in-time optimization. Like Dapper, logging based on random sampling would miss interesting traces.

Yuan et al. [25, 26, 27] have pioneered the work on log-based failure diagnosis. LogEnhancer [27] aims to enhance the recorded contents in existing logging statements by automatically identifying and inserting critical variable values into them. ErrLog [26] utilizes a number of exception patterns that potentially cause system failures, and then adds proactive logging code to automatically log all of them. These work mainly address the problems of "what to log" and "where to log". Our work, instead, focuses on "whether to log".

9 Conclusion

In this paper, we have presented Log^2, a cost-aware logging system for making the optimal "whether to log" decisions. Log^2 adopts a two-phase filtering mechanism to selectively record useful logs based on a given logging bandwidth. The experimental results on both BlogEngine and ServiceX demonstrate the capability of Log^2 to control logging overhead while preserving effectiveness.

Currently, Log^2 analyzes performance logs for performance monitoring and diagnosis. As we discussed in Section 7, in the future we will extend Log^2 to support more type of analysis, such as supporting other kinds of logs for failure diagnosis.

References

[1] Blogengine, 2007. http://www.dotnetblogengine.net/.

[2] Etw tracing, 2007. https://msdn.microsoft.com/en-us/library/ms751538(v=vs.110).aspx.

[3] Record and run a web performance test, 2013. http://msdn.microsoft.com/en-us/library/ms182539.aspx.

[4] Log2, an overhead-constrained logging system, 2014. http://research.microsoft.com/en-us/projects/log2/default.aspx.

[5] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[6] ARNOLD, M., AND RYDER, B. G. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation* (2001), ACM Press, pp. 168–179.

[7] ARULRAJ, J., CHANG, P., JIN, G., AND LU, S. Production-run software failure diagnosis via hardware performance counters. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, USA* (2013), pp. 101–112.

[8] CHONG, K. H., Y., G. C., AND Y, L. Pid control system analysis, design, and technology. In *IEEE Trans Control Systems Tech* (2005).

[9] DING, R., FU, Q., LOU, J., LIN, Q., ZHANG, D., AND XIE, T. Mining historical issue repositories to heal large-scale online service systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA* (2014), pp. 311–322.

[10] ERLING, A. Sufficiency and exponential families for discrete sample spaces. In *Journal of the American Statistical Association* (1970).

[11] GOODELL, B. R. *Smoothing Forecasting and Prediction of Discrete Time Series*. Englewood Cliffs, NJ: Prentice-Hall, 1963.

[12] HAN, S., DANG, Y., GE, S., ZHANG, D., AND XIE, T. Performance debugging in the large via mining millions of stack traces. In *34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland* (2012), pp. 145–155.

[13] JIN, G., SONG, L., SHI, X., SCHERPELZ, J., AND LU, S. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12* (2012), pp. 77–88.

[14] JOVIC, M., ADAMOLI, A., AND HAUSWIRTH, M. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 155–170.

[15] L, W. *All of statistics: A concise course in statistical inference*. New York: Springer, 2004.

[16] LIU, Y., XU, C., AND CHEUNG, S. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India* (2014), pp. 1013–1024.

[17] MILLER, B., CALLAGHAN, M., CARGILLE, J., HOLLINGSWORTH, J., IRVIN, R., KARAVANIC, K., KUNCHITHAPADAM, K., AND NEWHALL, T. The paradyn parallel performance measurement tool. In *IEEE Computer* (1995).

[18] MYRON, A., AND ELI, I. *Numerical analysis for applied science*. John Wiley, Sons, 1998.

[19] OGATA, K. *Discrete-time control systems*. Prentice-Hall, 1987.

[20] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. In *Google technical report* (2010).

[21] SONG, L., AND LU, S. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA,* (2014), pp. 561–578.

[22] WHEELER, D. J., AND CHAMBERS, D. S. *Understanding Statistical Process Control*. SPC Press, 1992.

[23] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 117–132.

[24] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2010).

[25] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

[26] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., ZHOU, Y., AND SAVAGE, S. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association.

[27] YUAN, D., ZHENG, J., PARK, S., ZHOU, Y., AND SAVAGE, S. Improving software diagnosability via log enhancement. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, March 2011).

[28] ZAMAN, S., ADAMS, B., AND HASSAN, A. E. A qualitative study on performance bugs. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, Zurich, Switzerland* (2012), pp. 199–208.

Identifying Trends in Enterprise Data Protection Systems

George Amvrosiadis
Dept. of Computer Science, University of Toronto
gamvrosi@cs.toronto.edu

Medha Bhadkamkar
Symantec Research Labs
medha_bhadkamkar@symantec.com

Abstract

Enterprises routinely use data protection techniques to achieve business continuity in the event of failures. To ensure that backup and recovery goals are met in the face of the steep data growth rates of modern workloads, data protection systems need to constantly evolve. Recent studies show that these systems routinely miss their goals today. However, there is little work in the literature to understand why this is the case.

In this paper, we present a study of 40,000 enterprise data protection systems deploying Symantec NetBackup, a commercial backup product. In total, we analyze over a million weekly reports which have been collected over a period of three years. We discover that the main reason behind inefficiencies in data protection systems is misconfigurations. Furthermore, our analysis shows that these systems grow in bursts, leaving clients unprotected at times, and are often configured using the default parameter values. As a result, we believe there is potential in developing automated, self-healing data protection systems that achieve higher efficiency standards. To aid researchers in the development of such systems, we use our dataset to identify trends characterizing data protection systems with regards to configuration, job scheduling, and data growth.

1 Introduction

Studies analyzing the characteristics of storage systems are an important aid in the design and implementation of techniques that can improve the performance and robustness of these systems. In the past 30 years, numerous file system studies have investigated different aspects of desktop and enterprise systems [2, 6, 7, 19, 30, 39, 47, 51, 55, 56]. However, little work has been published to provide insight in the characteristics of backup systems, focusing on deduplication rates [52], and the characteristics of the file systems storing the backup images [66]. With this study, we look into the backup application generating these images, their internal structure, and the characteristics of the jobs that created them.

Modern data growth rates and shorter recovery windows are driving the need for innovation in the area of data protection. Recent surveys of CIOs and IT professionals indicate that 90% of businesses use more than two backup products [18], and only 28% of backup jobs complete within their scheduled window [34, 65]. The goal of this study is to investigate how data protection systems are configured and operate. Our analysis shows that the inefficiency of backup systems is largely attributed to misconfigurations. We believe automating configuration management can help alleviate these configuration issues significantly. Our findings motivate and support research on automated data protection [22, 27], by identifying trends in data protection systems, and related directions for future research.

Our study is based on a million weekly reports collected in a span of three years, from $40,000$ enterprise backup systems, also referred to as *domains* in the rest of the paper. Each domain is a multi-tiered network of backup servers deploying Symantec NetBackup [61], an enterprise backup product. To the best of our knowledge, this dataset is the largest in existing literature in terms of both the number of domains, and the time span covered. As a result, we are able to analyze the characteristics of a diverse domain population, and its evolution over time.

First, we investigate how backup domains are configured. Identifying common growth trends is useful for provisioning system resources, such as network or storage bandwidth, to accommodate future growth. We find that the population of protected client machines grows in bursts and rarely shrinks. Furthermore, domains protect data of a single type, such as database files or virtual machines, regardless of domain size. Overall, our findings suggest that automated configuration is an important and feasible direction for future research to accommodate growth bursts in the number of protected clients.

The configuration of a backup system, with regards to job frequency and scheduling, is also an important contributor to resource consumption. Understanding common practices employed by systems in the field can give us better insight in the load that these systems face, and the characteristics of that load. To derive these trends, we analyzed 210 million jobs performing a variety of tasks, ranging from data backup and recovery, to management

Characteristic	Observation	Section	Previous work
System setup	The initial configuration period of backup domains is at least 3 weeks.	4.1	None
Protected clients	Clients tend to be added to a domain in groups, on a monthly basis.	4.2	None
Backup policies	82% of backup domains protect one type of data.	4.3	None
	The number of backup job policies in a domain remains mostly fixed. Also, 79% of clients subscribe to a single policy.	4.4	None
Job frequency	Full backups tend to occur every few days, while incremental ones occur daily. Recovery operations occur for few domains, on a weekly or monthly basis.	5.2	None
	Users prefer default scheduling windows during weekdays, resulting in nightly bursts of activity.	5.3	None
Job sizes	Incremental and full backups tend to be similar to each other in terms of size and number of files. Recovery jobs restore either few files and bytes, or entire volumes.	6.1	Considers file sizes instead [66]
Deduplication ratios	Deduplication can result in the reduction of backup image sizes by more than 88%, despite average job sizes ranging in the tens of gigabytes.	6.2	We confirm their findings [66]
Data retention	Incremental backups are retained for weeks, while full backups are retained for months and retention depends on their scheduling frequency.	6.3	We confirm their findings [66]

Table 1: A summary of the most important observations of our study.

of backup archives. We find that jobs occur in bursts, due to the preference of default scheduling parameters by users. Moreover, job types are strongly correlated to specific days and times of the week. To avoid these bursts of activity, we expect future backup systems to follow more flexible scheduling plans based on data protection guarantees and resource availability [4, 26, 48].

Finally, successful resource provisioning for backup storage capacity requires data growth rate knowledge. Our results show that jobs in the order of tens of GBs are the norm, even with deduplication ratios of 88%. Also, retention periods for these jobs are selected as a function of backup frequency, and backups are performed at intervals significantly shorter than the periods for which they are retained. Thus, future data protection offering faster backup and recovery times through the use of snapshots [1, 22], will have to be designed to handle significant data churn, or employ these mechanisms selectively.

We summarize the most important observations of our study in Table 1. Note that a *policy* (see Section 2.2) refers to a predefined set of configuration parameters specific to an application. The rest of the paper is organized as follows. In Section 2, we provide an overview of the evolution of backup systems. Section 3 describes the dataset used in this study. Sections 4 through 6 present our analysis results on backup domain configuration, job scheduling, and data growth, respectively. Finally, we discuss directions for research on next-generation data protection systems, supported by our findings, in Section 7, and conclude in Section 8.

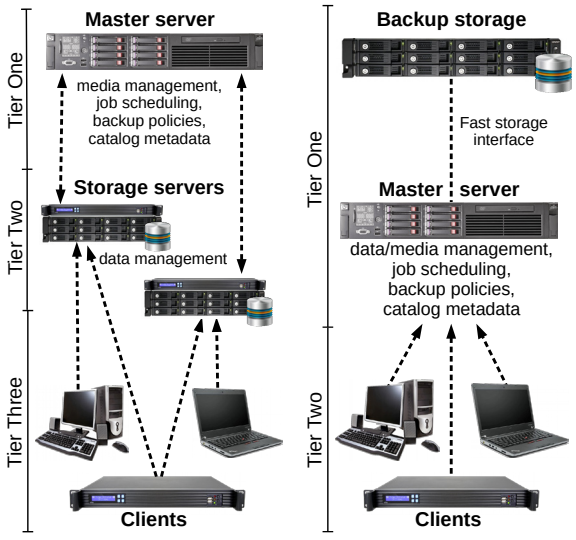
2 Background

Formally, *backup* is the process of making redundant copies of data, so that it can be retrieved if the original copy becomes unavailable. In the past 30 years, however, data growth coupled with capacity and bandwidth limitations have triggered a number of paradigm shifts in the way backup is performed. Recently, data growth trends have once again prompted efforts to rethink backup [1, 9, 20, 22, 27]. This section underlines the importance of field studies in this process (Section 2.1), putting our study in context, and describes the architecture of modern backup systems (Section 2.2).

2.1 Evolution of backup and field studies

In the early 1990s, backup consisted of using simple command-line tools to copy data to/from tape. A number of studies tested and outlined the shortcomings of these contemporary backup methods [38, 54, 69, 70]. The limitations of this approach, which included scaling, archive management, operating on online systems, and completion time, were subsequently addressed sufficiently by moving to a client-server backup model [8, 11, 15, 16]. In this model, job scheduling, policy configuration, and archive cataloging were all unified at the server side.

In the early 2000s, deduplicating storage systems were developed [53, 67], which removed data redundancy, lowering the cost of backup storage. Subsequently, Wallace et al. [66] published a study that aims to characterize backup storage characteristics by looking at the contents and workload of file systems that store images produced by backup applications such as NetBackup. A large body of work used their results to simulate deduplicating backup systems more realistically [41, 43, 44, 57, 62], and was built on the motivation provided by the study's results [40, 42, 46, 58]. The authors analyze weekly reports from appliances, while we analyze reports from the backup application, which has visibility within the archives and the jobs that created them. However, the two studies overlap in three points. First, the deduplication ratios reported for backups confirm our findings. Second, we report backup data retention as a configuration parameter, while they report on file age, two distri-



(a) 3-tier architecture (b) 2-tier architecture

Figure 1: Architecture of a modern backup domain.

butions that overlap for popular values. Third, the average job sizes we report are 5-8 times smaller than the file sizes reported in their study, likely because they take into account all files in the file system storing the backup images. Overlaps between our study and previous work are summarized in Table 1.

Recently, an ongoing effort has been initiated in the industry to redefine enterprise data protection as a response to modern data growth rates and shorter backup windows [12, 18, 65]. Proposed deviations from the traditional model rely on data snapshots, trading management complexity for faster job completion rates [22], and a paradigm shift from backup to data protection policies, in which users specify constraints on data availability as opposed to backup frequency and scheduling [1]. The latter paradigm allows the system to make decisions on individual policy parameters that can increase global efficiency, while keeping misconfigurations to a minimum. In this direction, previous work leverages predictive analytics to configure backup systems [9, 20, 25]. We believe that all this work is promising, and that a study characterizing the configuration and evolution of backup systems over time could aid in developing new approaches and predictive models that ensure backup systems meet their goals timely, while efficiently utilizing their resources.

2.2 Anatomy of modern backup systems

Modern *backup domains* typically consist of three tiers of operation: a master server, one or more storage servers, and several clients, as shown in Figure 1a. The domain's *master server* maintains information on backup images and backup policies. It is also responsible for scheduling and monitoring backup jobs, and assigning them to storage servers. *Storage servers* manage storage media, such as tapes and hard drives, used to archive backup images. By abstracting storage media management in this way, clients can send data directly to their corresponding storage server, avoiding a bandwidth bottleneck at the master server. Finally, domain *clients* can be desktops, servers, or virtual machines generating data that is protected by the backup system against failures. In an alternative 2-tiered architecture model (Figure 1b), the storage servers are absent and the storage media are directly managed by the master server. The majority of enterprise backup software today, including Symantec NetBackup, support the 3-tiered model [3, 5, 13, 17, 21, 28, 32, 60, 68].

Performing a backup generally consists of a sequence of operations, each of which is executed as an independent *job*. Such jobs include: *snapshots* of the state of data at a given point in time, copying data into a backup image as part of a *full backup*, copying modified data since the last backup as part of an *incremental backup*, restoring data from a backup image as part of a *recovery operation*, and managing backup images or backing up the domain's configuration as part of a *management operation*. These jobs are typically employed in a predefined order. For example, a full backup may be followed by a management operation that deletes backup images past their retention periods.

To be consistently backed up, or provide point-in-time recovery guarantees, business applications may require specific operations to take place. In these scenarios, backup products offer predefined *policies* that are specific to individual applications. For instance, a Microsoft Exchange Server policy will also backup the transaction log, to capture any updates since the backup was initiated. Users can further configure policies to specify the characteristics of backups jobs, such as their frequency and retention rate.

3 Dataset Information

Our analysis is based on *telemetry reports* collected from customer installations of a commercial backup product, Symantec NetBackup [61], in enterprise and regular production environments. Reports are only collected from customers who opted to participate in the telemetry program, so our dataset represents a fraction of the customer base. The reports contain no personal identifiable information, or details about the data being backed up.

Report types. Each report in our dataset belongs to exactly one of three types: installation, runtime, or domain report. Reports of different types are collected at distinct points in the lifetime of a backup domain. *Installation*

Report type	Metrics used in study
Installation	Installation time
Runtime report	Job information: starting time, type, size, number of files, client policy, deduplication ratio, retention period
Domain report	Number and type of policies, number of clients, number of storage media, number of storage servers and appliances

Table 2: Telemetry report metrics used in the study.

reports are generated when the backup software is successfully installed on a server, and can be used to determine the time each server of a domain first came online. *Runtime reports* are generated and transmitted on a weekly basis from online domains, and contain daily aggregate data about the backup jobs running on the system. *Domain reports* are also generated and transmitted on a weekly basis, and report daily aggregate metrics that describe the configuration of the backup domain. The telemetry report metrics used in this study are summarized in Table 2.

Dataset size. The telemetry reports in our dataset were collected over the span of 3 years (January 2012 to December 2014), across two major versions of the NetBackup software. We collected 1 million reports from over 40,000 server installations deployed in 124 countries, on most modern operating systems.

Monitoring duration. The backup domains included in our study were each monitored for 5.5 months on average, and up to 32 months. We elaborate on our strategy for excluding some of the domains from our analysis in Section 4.1. Note that the monitoring time is not always equivalent to the total lifetime of the domain, as many of these domains were still online at the time of this writing.

Architecture. While NetBackup supports the 3-tiered architecture model, only 35% of domains in our dataset use dedicated storage servers. The remaining domains omit that layer, opting for a 2-tier system instead. Additionally, while backup software can be installed on any server, storage companies also offer Purpose-Built Backup Appliances (PBBAs) [33]. 31% of domains in our dataset represent this market by deploying NetBackup on Symantec PBBAs.

4 Domain configuration

This section analyzes the way backup domains are configured with regards to their clients and backup policies. We use the periodic telemetry reports to quantify the growth rate of the number of clients and policies across domains, and characterize the diversity of policy types based on the type of data and applications they protect.

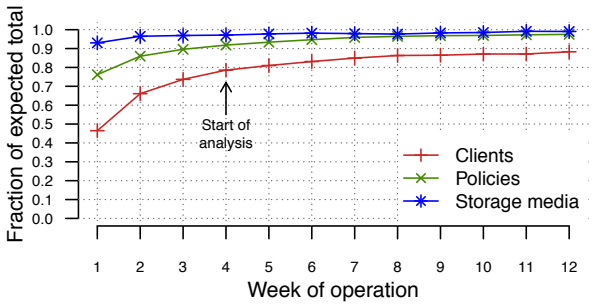


Figure 2: The average number of clients, policies, and storage media for a given week of operation, as a fraction of the expected total, i.e. the overall mean. We begin our analysis on the fourth week of operation, when these quantities become relatively stable.

4.1 Initial configuration period

Observation 1: *Backup domains take at least 3 weeks to reach a stable configuration after installation.*

The number of clients, policies, and storage media are three characteristic factors of a backup domain's configuration. These numbers fluctuate as resources are added to, or removed from the domain. As we monitor domains since their creation, we find the number of clients, policies, and storage media to be initially close to zero, and then increase rapidly until the domain is properly configured. After this initial configuration period, variability for these numbers tends to be low over the lifetime of each domain, with standard deviations less than 16% of the corresponding mean.

To avoid having the initial weeks of operation affect our results, we exclude them from our analysis. To estimate the average configuration period length, we analyze the number of clients, policies, and storage media in a backup domain as a fraction of the overall mean, i.e. the expected total. In Figure 2, we report the average fractions for all domains that have been monitored for more than 16 weeks. For example, a fraction of 0.47 for the number of clients during the first week of operation, implies that the number of clients at that time is 47% of the domain's expected total. With the exception of storage media, which seem to be added to backup domains from their first week of operation, we find that the number of clients and policies tends to be significantly lower for the first 3 weeks of operation. As a result, we choose to start our analysis from the fourth week of operation.

4.2 Client growth rate

Observation 2: *The number of clients in a domain increases by an average of 7 clients every 3.7 months.*

Clients are the producers of backup data, and the con-

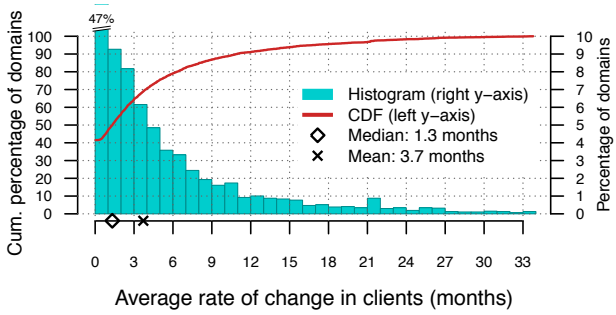


Figure 3: Distribution of the average rate at which the number of clients changes, across all domains in our dataset. On average, 93% of client population changes are attributed to the addition of clients.

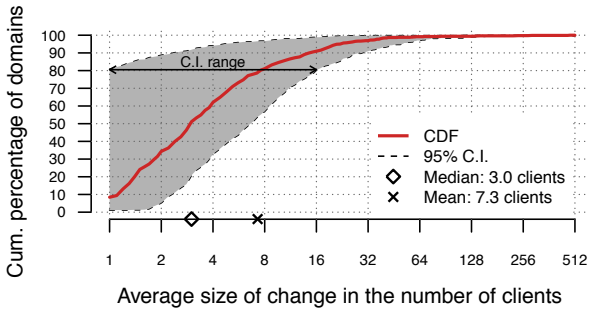


Figure 4: Distribution of the average number of clients involved in each change of a domain's client population, across all domains in our dataset. The 95% confidence intervals (C.I.) for each domain's average are also shown.

sumers of said data during recovery. As a result, the number of jobs running on a backup domain is directly proportional to the number of clients in the domain, deeming it important to quantify the rate at which their population grows over time.

Once the initial configuration period for a backup domain has elapsed, we find that clients tend to be added to, or removed from the domain in groups. Therefore, we characterize a domain's client population growth by quantifying the average rate of change in the client population, the sign indicating an increase or decrease in the population, and size of each change.

To estimate the rate at which the number of clients change, we extract inter-arrival times between changes through change-point analysis [37], a cost-benefit approach for detecting changes in time series. Then, we estimate the average rate of change for a domain as the average of these inter-arrival times. In Figure 3, we show the distribution of the average rates of change, i.e. the average number of months between changes in the number of clients across domains. For 42% of backup domains, the number of clients remains fixed after the first 3 weeks of operation, while on average the number of clients in a domain changes every 3.7 months. Overall, we find no strong correlation between the rate of change in the number of clients, and the domain's lifetime.

We further analyze the sign and size of each population change. Of all events in which a domain's client population changes, 93% are attributed to the addition of clients. However, 78% of domains never remove clients. Regarding the size of each change, Figure 4 shows the distribution of the average number of clients involved in each change, across all domains in our study. On average, a domain's population changes by 7.3 clients at a time. The average standard deviation of the number of clients over time is 13.1% of the corresponding expected value, indicating low variation overall. However, the 95% confidence intervals (C.I.) for each mean (Figure 4), suggest that growth spurts as large as 2.16 times the average value are possible, as this is the width of the average 95% confidence interval.

Policy category	Domains with at least 1 policy
File and block policy	61.24%
Database policy	20.34%
Virtual machine policy	15.13%
Application policy	13.52%
Metadata backup policy	**31.93%**

Table 3: Percentage of backup domains with at least one policy of a given category. Less than a third of domains protect the master server using a metadata backup policy.

4.3 Diversity of protected data

Observation 3: *82% of backup domains protect one type of data, and only 32% of domains effectively protect the master server's state and metadata.*

To provide consistent online backups, backup products offer optimizations for different application types, implemented as dedicated policy types [14, 23, 59]. For our analysis, we partitioned these policy types into four categories. *File and block policies* are specifically tailored for backing up raw device data blocks, or file and operating system data and metadata, e.g. from NTFS, AFS, or Windows volumes. *Database policies* are designed to provide consistent online backups for specific database management systems, such as DB2 and Oracle. *Virtual machine policies* are tuned to backup and restore VM images, from virtual environments such as VMware or Hyper-V. *Application policies* specialize in backing up state for client-server applications, such as Microsoft Exchange and Lotus Notes. Finally, a *metadata backup policy* can be setup to backup the master server's state.

In Table 3, we show the probability that at least one policy of a given category will be present in a backup domain. Since domains may deploy policies from multiple categories, these percentages add up to more than 100%.

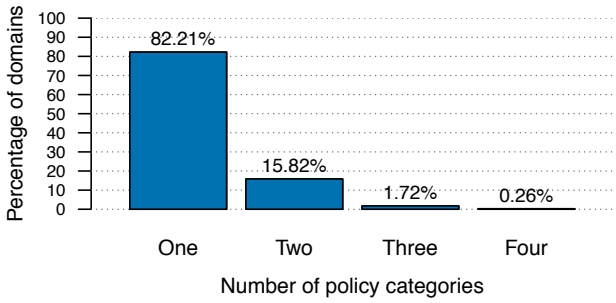


Figure 5: Distribution of the number of policy categories per backup domain. The metadata backup policy category is not accounted for in these numbers.

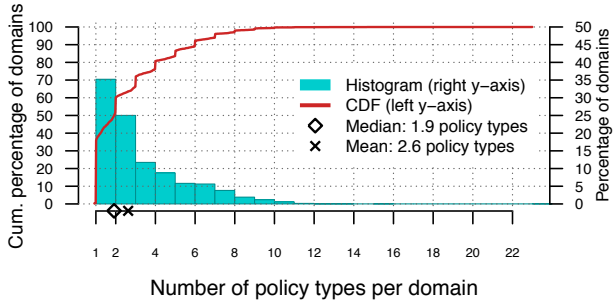


Figure 6: Distribution of the number of policy types per backup domain, across all domains in the study. More than 25 distinct NetBackup policy types are present in the telemetry data.

Surprisingly, we find that only 32% of backup domains register a metadata backup policy to protect the master server's data. While the remaining domains may employ a different mechanism to backup the master server, guaranteeing no data inconsistencies while doing so is challenging. In any case, this result suggests that automatically configured metadata backup policies should be a priority for future backup systems.

We also look into the number of policy categories represented by each domain's policies, to gauge the diversity in the types of protected data. Interestingly, Figure 5 shows that 82% of domains deploy policies of a single category (excluding metadata backup policies), and the remaining domains mostly use policies of two distinct categories. We further examine the number of distinct policy types that are deployed in each domain. As shown in Figure 6, domains tend to make use of a small number of policy types. Specifically, 61% of the domains deploy policies of only one, or two distinct types.

4.4 Backup policies

Observation 4: *After the initial configuration period, the number of policies in a domain remains mostly fixed and 79% of clients subscribe to a single policy each.*

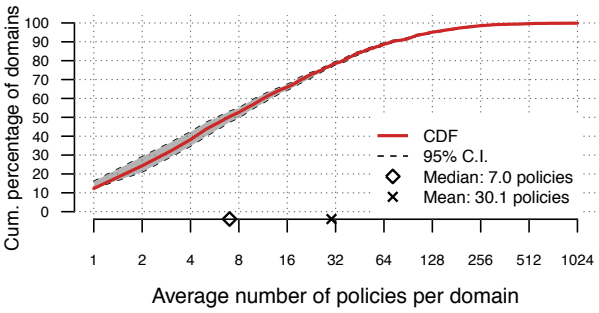


Figure 7: Distribution of the average number of policies per backup domain. The 95% confidence intervals for each average are also shown. Overall, the number of policies remains stable over the lifetime of a domain.

Following from Section 4.2, the policies in a backup domain, along with the number of clients, are indicative of the domain's load. Recall from Section 2.2, that clients subscribe to policies which determine the characteristics of backup jobs. Therefore, it is important to quantify both the number of policies in a domain and the characteristics of each, to effectively characterize the domain's workload. We defer an analysis of job characteristics to the remainder of the paper, and focus here on the number of policies in each domain.

In Figure 7, we show the distribution of the average number of policies in a given backup domain, across all domains in our dataset. Overall, we find that once the initial configuration period is complete, the number of backup policies in a domain remains mostly stable. Specifically, the expected width of the 95% confidence interval is 2.5% of the average number of policies.

Figure 7 also shows that the average backup domain carries 30 backup policies, while 5% of domains carry over 128. While each policy may represent a group of clients with specific data protection needs, we find that individual clients usually subscribe to a single policy. In Figure 8, we show the distribution of the average number of policies that each client subscribes to. More than 79% of clients belong to only one policy, while 16% spend some or most of their time unprotected (less than one policy on average). The latter result, coupled with the large number of policies in backup domains and the fact that clients are added to a domain in groups (Section 4.2), suggests that manual policy configuration might not be ideal as a domain's client population inflates over time.

5 Job scheduling

While the master server can reorder policy jobs to increase overall system efficiency, it adheres to user preferences that dictate when, and how often a job should be scheduled. This section looks into the way that these pa-

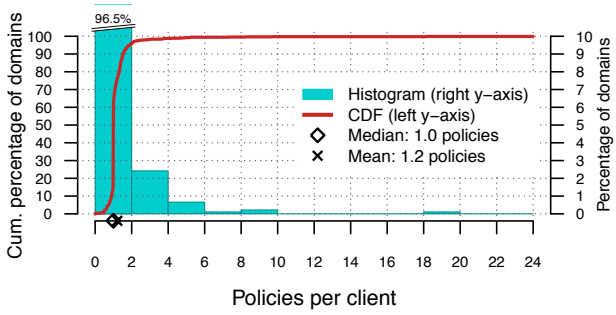


Figure 8: Distribution of the average number of policies that a domain client subscribes to. Overall, 79% of clients subscribe to one policy, while 16% spend some or most time unprotected by a policy ($x < 1$).

Job type	Percentage of jobs
Incremental Backups	45.27%
Full Backups	31.20%
Snapshot Operations	12.61%
Management Operations	10.12%
Recovery Operations	0.80%

Table 4: Breakdown of all jobs in the dataset by type.

rameters are configured by users across backup domains, and the workload generated in the domain as a result.

5.1 Job types

Recall from Section 2.2 that policies consist of a predefined series of operations, each carried out by a separate job. We collected data from 209.5 million jobs, and we group them in five distinct categories: full and incremental backups, snapshots, recovery, and management operations. In Table 4, we show a breakdown of all jobs in our dataset by job type. Across all monitored backup domains, we find that 76% of jobs perform data backups, having processed a total of 1.64 Exabytes of data, while 13% of jobs take snapshots of data. On the other hand, less than 1% of jobs are tasked with data recovery, having restored a total of 5.12 Petabytes of data. Finally, 10% of jobs are used to manage backup images, e.g. migrate, duplicate, or delete them. Due to the data transfer of backup images, these jobs processed 4.88 Exabytes of data. We analyze individual job sizes in Section 6.

5.2 Scheduling frequency

Observation 5: *Full backups tend to occur every 5 days or fewer. Recovery operations occur for few domains, on a weekly or monthly basis.*

A factor indicative of data churn in a backup domain is the rate at which jobs are scheduled to backup, restore, or manage backed-up data. To quantify the scheduling frequency of different job types for a given domain, we

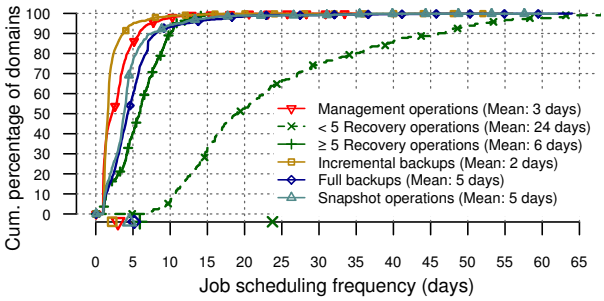


Figure 9: Distribution of the average scheduling frequency of different job types across backup domains. Recovery operations are broken into two groups of domains with more, and less than 5 recovery operations each. Despite being of similar size, the characteristics of each group differ significantly.

rely on the starting times of individual jobs. Specifically, starting times are used to estimate the average occurrence rate of different jobs of each domain policy, on individual clients. In Figure 9, we show the distributions of the scheduling frequency of different job types across backup domains.

Overall, we find that the average frequency of recovery operations differs depending on their number. In Figure 9, we show the distributions of the recovery frequency for two domain groups having recovered data more, and less than 5 times. The former group consists of 337 domains that recovered data 17 times on average, and the latter consists of 262 domains with 3 recovery operations on average. By definition, our analysis excludes an additional 676 domains that initiate recovery only once. For domains with multiple events, the distribution of their frequency spans 1-2 weeks, with an average of 6 days. On the other hand, domains with fewer recovery operations perform them significantly less frequently, up to 2 months apart and every 24 days on average. Since recovery operations are initiated manually by users, we have no accurate way of pinpointing their cause. These results, however, suggest that frequent recovery operations may be attributed to disaster recovery testing, while infrequent ones may be due to actual disasters. Interestingly, both domain groups are equally small, but when domains with a single recovery event are factored in, the group of infrequent recovery operations doubles in size.

In the case of backup jobs, the general belief is that systems in the field rely on weekly full backups, complemented by daily incremental backups [11, 36, 67]. Our results confirm this assumption for incremental backups, which take place every 1-2 days in 81% of domains. Daily incremental backups are also the default option in NetBackup. For full backups, however, our analysis shows that only 17% of domains perform them every 6-8

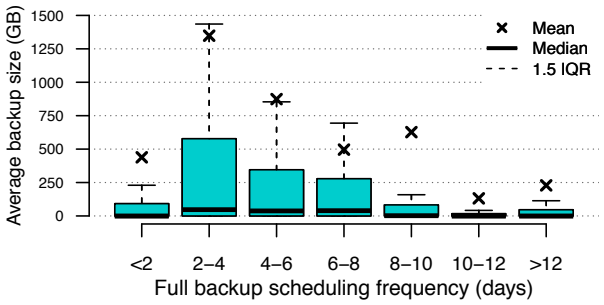


Figure 10: Tukey boxplots (without outliers) that represent the average size of full backup jobs, for different job scheduling frequencies. Means for each boxplot are also shown. Frequent full backups seem to be associated with larger job sizes, suggesting that they may be preferred as a response to high data churn.

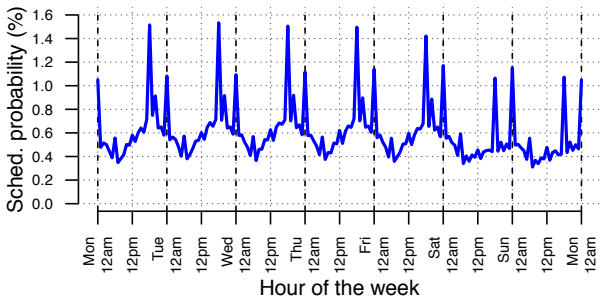


Figure 11: Probability density function for scheduling policy jobs at a given hour of a given day of the week. Policies tend to be configured using the default scheduling windows at 6pm and 12am, resulting in high system load during those hours.

days on average. Instead, the majority of domains perform full backups more often: 15% perform them every 1-2 days, and 57% perform them every 2-6 days. This is despite the fact that weekly full backups is the default option. As expected, management operations take place on a daily or weekly basis, since they usually follow (or precede) an incremental or full backup operation. Snapshot operations display a similar trend to full backups, as they are mostly used by clients in lieu of the latter.

Of the 65% of domain policies that perform full backups every 6 days or fewer, only 33% also perform incremental backups at all. On the other hand, 76% of policies that perform weekly full backups also rely on incremental backups. To determine whether full backups are performed frequently to accommodate high data churn, we group average full backup sizes per client policy according to their scheduling frequency, and present the results as a series of boxplots in Figure 10. Note that regardless of frequency, full backups tend to be small (medians in the order of a few gigabytes), due to the efficiency of deduplication. However, the larger percentiles of each distribution show that larger backup sizes tend to occur when full backups are taken more frequently than once per week. While this confirms our assumption of high data churn for a fraction of the clients, the remaining small backup sizes could also be attributed to overly conservative configurations, a sign that policy auto-configuration is an important feature for future data protection systems.

5.3 Scheduling windows

Observation 6: *Users prefer default scheduling windows during weekdays, resulting in nightly bursts of activity. Default values are overridden, however, to avoid scheduling jobs during the weekend.*

Another important factor for characterizing the workload of a backup system is the exact time jobs are scheduled. A popular belief is that backup operations take place late at night or during weekends, when client systems are expected to be idle [15, 66]. In Figure 11, we show our findings for all the jobs in our dataset. The presented density function was computed by normalizing the number of jobs that take place in a given domain, to prevent domains with more jobs from affecting the overall trend disproportionately. We note that this normalization had minimal effect on the result, which suggests that the presented trend is common across domains.

The hourly scheduling frequency is similar for each day, although there is less activity during the weekend. We also find that the probability of a job being scheduled is highest starting at 6pm and 12am on a weekday. We attribute the timing of job scheduling to customers using the default scheduling windows suggested by NetBackup, which start at 6pm and 12am every day. The choice to exclude weekends, however, seems to be an explicit choice of the user. This result suggests that automated job scheduling, where the only constraints would be to leverage device idleness [4, 26, 48], would be more practical, allowing the system to schedule jobs so that such activity bursts are avoided.

While Figure 11 merges all job types, different jobs exhibit different scheduling patterns, as shown in Figure 9. Our data, however, does not allow a matching of job types to scheduling times at a granularity finer than the day on which the job was scheduled. Thus, we partition jobs based on their type, and in Figure 12 we show the probability that a job of a given type will be scheduled on a given day of the week. We find that incremental backups are scheduled to complement full backups, as they tend to get scheduled from Monday to Thursday, while full backups are mostly scheduled on Fridays. Note that the latter does not contradict our previous result of full backups that take place more often than once a week,

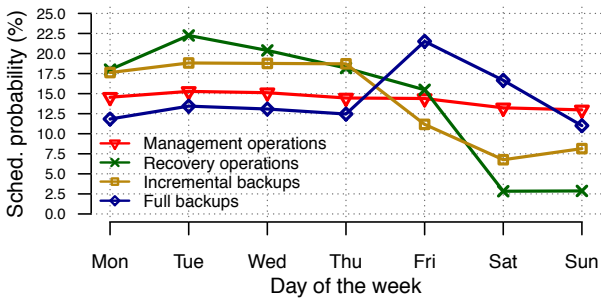


Figure 12: Probability of a policy job occurring on a given day of the week, based on its type. Incremental backups tend to be scheduled to complement full backups, while users initiate recovery operations more frequently at the beginning of the week.

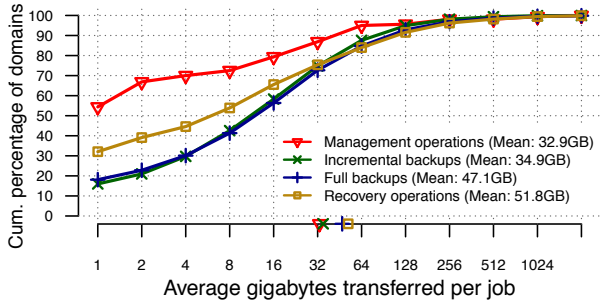


Figure 13: Distribution of the average job size of a given job type across backup domains, after the data has been deduplicated at the client side. Incremental backups resemble full backups in size.

as the probability of scheduling full backups any other day is still comparatively high. Recovery operations also take place within the week, with a slightly higher probability on Tuesdays (which we confirmed as not related to Patch Tuesday [49]). Finally management operations do not follow any particular trend and are equally likely to be scheduled on any day of the week.

6 Backup data growth

Characterizing backup data growth is crucial for estimating the amount of data that needs to be transferred and stored, which allows for efficient provisioning of storage capacity and bandwidth. Towards this goal, we analyze the sizes and number of files of different job types, and their deduplication ratios across backup domains. Finally, we look into the time that backup data is retained.

6.1 Job sizes and number of files

Observation 7: *Incremental and full backups tend to be similar in size and files transferred, due to the effectiveness of deduplication, or misconfigurations. Recovery jobs restore either a few files, or entire volumes.*

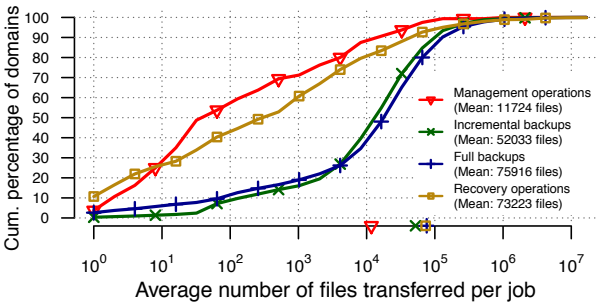


Figure 14: Distributions of average number of files transferred per job, across different job types. The trends are consistent with those for job sizes (Figure 13).

An obvious factor when estimating a domain's data growth is the size of backup jobs. In Figure 13, we show the distributions of the average number of bytes transferred for different job types across all domains, after the data has been deduplicated at the client. Averages for each operation are shown in the legend, and marked on the x axis. Snapshot operations are not included, as they do not incur data transfer.

Surprisingly, incremental backups resemble full backups in size. Although the distribution of full backups is skewed toward larger job sizes, 29% of full backups on domains that also perform incremental backups tend to be equal or smaller in size than the latter, 21% range from $1-1.5$ times the size of incremental backups, and the remainder range from $1.5-10^6$ times. We attribute the small size difference to three reasons. First, systems with low data churn can achieve high deduplication rates, which are common as we show in Section 6.2. Second, misconfigured policies or volumes that do not support incremental backups often fall back to full backups, as suggested by support tickets. Third, maintenance applications, such as anti-virus scanners, can update file metadata making unchanged files appear modified. Overall, the average backup job sizes in Figure 13 are 5-8 times smaller than the file sizes reported by Wallace et al. [66], likely due to their study considering the sizes of all files in the file system storing the backup images.

Since recovery operations can be triggered by users to recover an entire volume or individual files, the distribution of recovery job sizes is not surprising. 32% of recovery jobs restore less than 1GB, while the average job can be as large as 51GB. Finally, management operations, which consist mostly of metadata backups (95.7%), but also backup image (1.5%) and snapshot (2.8%) duplication operations, are much smaller than all other operations, as expected.

Figure 14 shows the distributions of the average number of files transferred for different job types in each domain. Similar to job sizes, the average number of files transferred per incremental backup is 31% smaller than

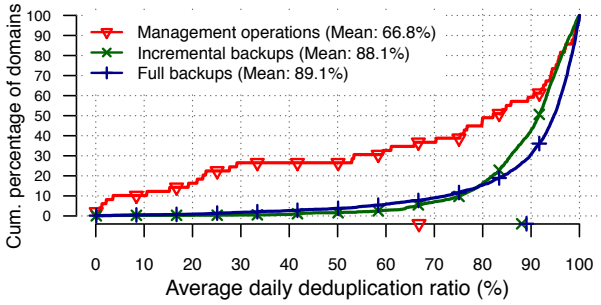


Figure 15: Distributions of the average daily deduplication ratio of different job types, across backup domains. Incremental and full backups observe high deduplication ratios, while the uniqueness of metadata backups (management operations) makes them harder to deduplicate.

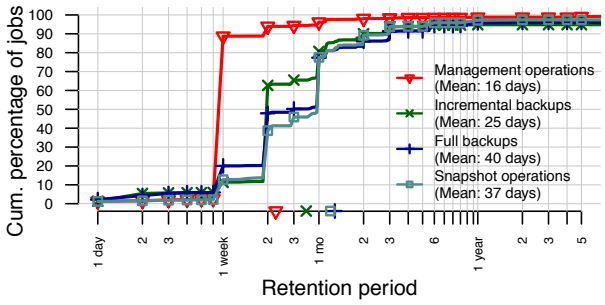


Figure 16: Distributions of retention period lengths for different job types. 3% of jobs have infinite retention periods. Incremental backups are typically retained for almost half the time of full backups, the majority of which are retained for months.

that for full backups, and both job types are characterized by similar CDF curves. Recovery operations transfer as many files as full backups on average, yet the majority transfer fewer than 200 files. This is in line with our results on recovery job sizes. Given that large recovery jobs also occur less frequently, these results suggest that most recovery operations are not triggered as a disaster response, but rather to recover data lost due to errors, or to test the recoverability of backup images. Management operations, being mostly metadata backups, transfer significantly fewer files than other job types on average.

6.2 Deduplication ratios

Observation 8: *Deduplication can result in the reduction of backup image sizes by more than 88%, despite average job sizes ranging in the tens of gigabytes.*

For clients that use NetBackup's deduplication solution, we analyzed the daily deduplication ratios of jobs, i.e. the percentage by which the number of bytes transferred was reduced due to deduplication. Figure 15 shows the distributions of the average daily deduplication ratio for management operations, full, and incremental backups across backup domains. Recovery and snapshot jobs are not included as the notion of deduplication does not apply. Since deduplication happens globally across backup images, deduplication ratios for backups tend to increase after the first few iterations of a policy. In general, sustained deduplication ratios as high as 99% are not unusual. Across all domains in our dataset, however, the average daily deduplication ratio is 88-89%, for both full and incremental backups. It is interesting to note that despite such high deduplication ratios, jobs in the order of tens of gigabytes are common (Figure 13), suggesting that even for daily incremental jobs, the actual job sizes are an order of magnitude larger in size. These results are in agreement with previous work [66], which reports average deduplication ratios of 91%.

Finally, for management operations the average deduplication ratio is 68%. Since only 1.1% of domains that use deduplication enable it for management operations, we do not attach much importance to this result. For the reported domains, however, it can be attributed to the uniqueness of metadata backups, which do not share files with other backup images on the same backup domain and consist of large binary files.

6.3 Data retention

Observation 9: *Incremental backups are retained for weeks, while full backups are retained for months and retention depends on their scheduling frequency.*

Another factor characteristic of backup storage growth is the retention time for backup images, which is a configurable policy parameter. Once a backup image expires, the master server deletes it from backup storage. We have analyzed the retention periods assigned to each job in our telemetry reports, and show the distributions of retention period lengths for different job types in Figure 16. Our initial observation is that job retention periods coincide with the values available by default in NetBackup, although users can specify custom periods. These values range from 1 week to 1 year, and correspond to the steps in the CDF shown. While federal laws, such as HIPAA [63] and FoIA [64], require minimum retention from a few years up to infinity for certain types of data. In our case, 3% of jobs are either assigned custom retention periods longer than 1 year, or are retained indefinitely. On the other extreme, only 3% of jobs are assigned custom retention periods shorter than 1 week. Previous work confirms our findings, by reporting similar ages for backup image files [66].

In particular, management operations (metadata backups and backup image duplicates) are mostly retained for 1 week. Incremental backups are mostly retained for 2 weeks, the default option. Full backups and snapshots,

on the other hand, are more likely retained for months. Overall, 94% of jobs select a preset retention period from NetBackup's list, and 35% of jobs keep the default suggestion of 2 weeks. This suggests that the actual retention period length is not a crucial policy parameter.

Finally, we find a strong correlation (Pearson's $r = 0.53$) between the length of retention periods for full backups, and the frequency with which they take place. Specifically, we find that clients taking full backups less frequently retain them for longer periods of time. On the other hand, no such correlation exists for management operations and incremental backups. This is because almost all data resulting from a management operation is retained for 1 week (Figure 16), and almost all incremental backups are performed with a frequency of 1-2 days apart (Figure 9). The correlation of retention period length and frequency of full backup operations, coupled with the preference for default values, may suggest that retention periods are selected as a function of storage capacity, or that they are at least limited by that factor.

7 Insight: next-generation data protection

This section outlines five major directions for future work on data protection systems. In each case, we identify existing literature and describe how our findings encourage future work.

Automated configuration and self-healing. To alleviate performance and availability problems of data protection systems, existing work uses historical data to perform automated storage capacity planning [9], data prefetching and network scheduling [25]. Our findings support this line of work. We have shown that backup domains grow in bursts, and client policies are either configured using default values, misconfigured, or not configured at all. As a result, clients are left unprotected, jobs are scheduled in bursts, and users are not warned of imminent problems. To enable automated policy configuration and self-healing data protection systems, further research is necessary.

Deduplication. Our findings confirm the efficiency of deduplication at reducing backup image sizes. We further show that in many systems, incremental backups are replaced by frequent full, deduplicated backups. This is likely due to the adoption of deduplication, which improves on incremental backups by looking for duplicates across all backup data in the domain. To completely replace incremental backups, however, it is necessary to improve on the time required to restore the original data from deduplicated storage, which directly affects recovery times. Currently, this is an area of active research [24, 35, 43, 50].

Efficient storage utilization. Our analysis shows that job retention periods are selected as a function of backup frequency, likely to ensure sufficient backup storage space will be available. Additionally, 31% of domains in our dataset use dedicated backup appliances (PBBAs), a market currently experiencing growth [33]. We believe that storage capacity in these dedicated systems should be utilized fully, and retention periods should be dynamically adjusted to fill it, providing the ability to recover older versions of data. In this direction, related work on stream-processing systems [29] could be adapted to the needs of backup data.

Accident insurance. Most recovery operations in our dataset appear to be small in both the number of files and bytes they recover, compared to their respective backups. This result suggests that recovery operations are mostly triggered to restore a few files, or to test the integrity of backup images. This motivates us to re-examine the requirement of instant recovery for backup systems as a problem of determining which data is more likely to be recovered, and storing it closer to clients [40, 45].

Content-aware backups. Data protection strategies can generate data at a rate up to 5 times higher than production data growth [1]. This is due to the practice of creating multiple copies and backing up temporary files used for test-and-development or data analytics processes, such as the Shuffle stage of MapReduce tasks [10]. Depending on the storage interface used, it might be more efficient to recompute these datasets rather than restoring them from backup storage. Another challenge for contemporary backup software is detecting data changes since the last backup among PBs of data and billions of files [31]. By augmenting data protection systems to account for data types and modification events, we can potentially reduce the time needed to complete backup and restore operations.

8 Conclusion

We investigated an extensive dataset representing a diverse population of enterprise data protection systems to demonstrate how these systems are configured and evolved over time. Among other results, our analysis showed that these systems are usually configured to protect one type of data, and while their client population growth is steady and bursty, their backup policies don't change. With regards to job scheduling, we find that the popularity of default values can have an adverse effect on the efficiency of the system by creating bursty workloads. Finally, we showed that full and incremental backups tend to be similar in size and number of files, as a result of efficient deduplication and misconfigurations. We hope that our data and the proposed areas of future research will enable researchers to simulate realistic scenarios for building next generation data protection systems that are easy to configure and manage.

Acknowledgments

The study would not be possible without the telemetry data collected by Symantec's NetBackup team, and we thank Liam McNerney and Aaron Christensen for their invaluable assistance in understanding the data. We also thank the four anonymous reviewers and our shepherd, Fred Douglis, for helping us improve our paper significantly. Finally, we would like to thank Petros Efstathopoulos, Fanglu Guo, Vish Janakiraman, Ashwin Kayyoor, CW Hobbs, Bruce Montague, Sanjay Sahwney, and all other members of Symantec's Research Labs for their feedback during the earlier stages of our study.

References

[1] ACTIFIO. Actifio Copy Data Virtualization: How It Works, August 2014.

[2] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007).

[3] ARCSERVE. arcserve Unified Data Protection. `http://www.arcserve.com`, May 2014.

[4] BACHMAT, E., AND SCHINDLER, J. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and modeling of computer systems* (2002).

[5] BACULA SYSTEMS. Bacula 7.0.5. `http://www.bacula.org`, July 2014.

[6] BAKER, M., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K., AND OUSTERHOUT, J. K. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (1991).

[7] BENNETT, J. M., BAUER, M. A., AND KINCHLEA, D. Characteristics of Files in NFS Environments. In *Proceedings of the 1991 ACM SIGSMALL/PC Symposium on Small Systems* (1991).

[8] BHATTACHARYA, S., MOHAN, C., BRANNON, K. W., NARANG, I., HSIAO, H.-I., AND SUBRAMANIAN, M. Coordinating Backup/Recovery and Data Consistency Between Database and File Systems. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (2002), SIGMOD.

[9] CHAMNESS, M. Capacity Forecasting in a Backup Storage Environment. In *Proceedings of the 25th International Conference on Large Installation System Administration* (2011).

[10] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1802–1813.

[11] CHERVENAK, A. L., VELLANKI, V., AND KURMAS, Z. Protecting File Systems: A Survey Of Backup Techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference* (1998).

[12] COMMVAULT SYSTEMS. Get Smart About Big Data: Integrated Backup, Archive & Reporting to Solve Big Data Management Problems, July 2013.

[13] COMMVAULT SYSTEMS INC. CommVault Simpana 10. `http://www.commvault.com/simpana-software`, April 2014.

[14] COMMVAULT SYSTEMS INC. CommVault Simpana: Solutions for Protecting and Managing Business Applications. `http://www.commvault.com/solutions/enterprise-applications`, April 2015.

[15] DA SILVA, J., GUDMUNDSSON, O., AND MOSSÉ, D. Performance of a Parallel Network Backup Manager, 1992.

[16] DA SILVA, J., AND GUTHMUNDSSON, O. The Amanda Network Backup Manager. In *Proceedings of the 7th USENIX Conference on System Administration* (1993), LISA.

[17] DELL INC. Dell NetVault 10.0. `http://software.dell.com/products/netvault-backup`, May 2014.

[18] DIMENSIONAL RESEARCH. The state of IT recovery for SMBs. `http://axcient.com/state-of-it-recovery-for-smbs`, Oct. 2014.

[19] DOUCEUR, J. R., AND BOLOSKY, W. J. A Large-scale Study of File-system Contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (1999).

[20] DOUGLIS, F., BHARDWAJ, D., QIAN, H., AND SHILANE, P. Content-aware Load Balancing for Distributed Backup. In *Proceedings of the 25th International Conference on Large Installation System Administration* (2011), LISA.

[21] EMC CORPORATION. EMC NetWorker 8.2. `http://www.emc.com/data-protection/networker.htm`, July 2014.

[22] EMC CORPORATION. EMC ProtectPoint: Protection Software Enabling Direct Backup from Primary Storage to Protection Storage, 2014.

[23] EMC CORPORATION. EMC NetWorker Application Modules Data Sheet. `http://www.emc.com/collateral/software/data-sheet/h2479-networker-app-modules-ds.pdf`, January 2015.

[24] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (2014).

[25] GIAT, A., PELLEG, D., RAICHSTEIN, E., AND RONEN, A. Using Machine Learning Techniques to Enhance the Performance of an Automatic Backup and Recovery System. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (2010), SYSTOR.

[26] GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (1995), TCON'95.

[27] HEWLETT-PACKARD. Rethinking backup and recovery in the modern data center, November 2013.

[28] HEWLETT-PACKARD COMPANY. HP Data Protector 9.0.1. `http://www.autonomy.com/products/data-protector`, August 2014.

[29] HILDRUM, K., DOUGLIS, F., WOLF, J. L., YU, P. S., FLEISCHER, L., AND KATTA, A. Storage Optimization for Large-scale Distributed Stream-processing Systems. *Trans. Storage 3*, 4 (Feb. 2008), 5:1–5:28.

[30] HSU, W. W., AND SMITH, A. J. Characteristics of I/O Traffic in Personal Computer and Server Workloads. Tech. rep., EECS Department, University of California, Berkeley, 2002.

[31] HUGHES, D., AND FARROW, R. Backup Strategies for Molecular Dynamics: An Interview with Doug Hughes. *Proc. USENIX ;login: 36*, 2 (Apr. 2011), 25–28.

[32] IBM CORPORATION. IBM Tivoli Storage Manager 7.1. `http://www.ibm.com/software/products/en/tivostormana`, November 2013.

[33] INTERNATIONAL DATA CORPORATION. Worldwide Purpose-Built Backup Appliance (PBBA) Market Revenue Increases 11.2% in the Third Quarter of 2014, According to IDC. `http://www.idc.com/getdoc.jsp?containerId=prUS25351414`, December 2014.

[34] IRON MOUNTAIN. Data Backup and Recovery Benchmark Report. `http://www.ironmountain.com/Knowledge-Center/Reference-Library/View-by-Document-Type/White-Papers-Briefs/I/Iron-Mountain-Data-Backup-and-Recovery-Benchmark-Report.aspx`, 2013.

[35] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing Impact of Data Fragmentation Caused by In-line Deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference* (2012).

[36] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), FAST.

[37] KILLICK, R., AND ECKLEY, I. A. changepoint: An R package for Changepoint Analysis. In *Journal of Statistical Software* (May 2013).

[38] KOLSTAD, R. A Next Step in Backup and Restore Technology. In *Proceedings of the 5th USENIX Conference on System Administration* (1991), LISA.

[39] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and Analysis of Large-scale Network File System Workloads. In *Proceedings of the USENIX 2008 Annual Technical Conference* (2008).

[40] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *Proceedings of the 2014 USENIX Annual Technical Conference* (2014), ATC.

[41] LI, M., QIN, C., LEE, P. P. C., AND LI, J. Convergent Dispersal: Toward Storage-Efficient Security in a Cloud-of-Clouds. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems* (2014), HotStorage.

[42] LI, Z., GREENAN, K. M., LEUNG, A. W., AND ZADOK, E. Power Consumption in Enterprise-scale Backup Storage Systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST.

[43] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *Proceeings of the 11th USENIX Conference on File and Storage Technologies* (2013), FAST.

[44] LIN, X., LU, G., DOUGLIS, F., SHILANE, P., AND WALLACE, G. Migratory Compression: Coarse-grained Data Reordering to Improve Compressibility. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST.

[45] LIU, J., CHAI, Y., QIN, X., AND XIAO, Y. PLC-cache: Endurable SSD cache for deduplication-based primary storage. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on* (2014).

[46] MEISTER, D., BRINKMANN, A., AND SÜSS, T. File Recipe Compression in Data Deduplication Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (2013), FAST.

[47] MEYER, D. T., AND BOLOSKY, W. J. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (2011).

[48] MI, N., RISKA, A., LI, X., SMIRNI, E., AND RIEDEL, E. Restrained utilization of idleness for transparent scheduling of background tasks. In *Proceedings of the 11th International Joint conference on Measurement and modeling of computer systems* (2009), SIGMETRICS.

[49] MICROSOFT CORPORATION. Understanding Windows automatic updating. `http://windows.microsoft.com/en-us/windows/understanding-windows-automatic-updating`.

[50] NG, C.-H., AND LEE, P. P. C. RevDedup: A Reverse Deduplication Storage System Optimized for Reads to Latest Backups. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (2013).

[51] OUSTERHOUT, J. K., DA COSTA, H., HARRISON, D., KUNZE, J. A., KUPFER, M., AND THOMPSON, J. G. A Trace-driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (1985).

[52] PARK, N., AND LILJA, D. J. Characterizing Datasets for Data Deduplication in Backup Applications. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2010), IISWC.

[53] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST.

[54] ROMIG, S. M. Backup at Ohio State, Take 2. In *Proceedings of the 4th USENIX Conference on System Administration* (1990), LISA.

[55] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference* (2000).

[56] SATYANARAYANAN, M. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (1981).

[57] SHIM, H., SHILANE, P., AND HSU, W. Characterization of Incremental Data Changes for Efficient Data Protection. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), ATC.

[58] SMALDONE, S., WALLACE, G., AND HSU, W. Efficiently Storing Virtual Machine Backups. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems* (2013), HotStorage.

[59] SYMANTEC CORPORATION. Symantec NetBackup 7.6 Data Sheet: Data Protection. `http://www.symantec.com/content/en/us/enterprise/fact_sheets/b-netbackup-ds-21324986.pdf`, January 2014.

[60] SYMANTEC CORPORATION. Symantec NetBackup 7.6. `http://www.symantec.com/backup-software`, March 2015.

[61] SYMANTEC CORPORATION. Symantec NetBackup 7.6.1 Getting Started Guide. `https://support.symantec.com/en_US/article.DOC7941.html`, February 2015.

[62] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating Realistic Datasets for Deduplication Analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), ATC.

[63] U.S. DEPARTMENT OF HEALTH AND HUMAN SERVICES. The Health Insurance Portability and Accountability Act. `http://www.hhs.gov/ocr/privacy`.

[64] U.S. DEPARTMENT OF JUSTICE. The Freedom of Information Act. `http://www.foia.gov`.

[65] VANSON BOURNE. Virtualization Data Protection Report 2013 – SMB edition. `http://www.dabcc.com/documentlibrary/file/virtualization-data-protection-report-smb-2013.pdf`, 2013.

[66] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012).

[67] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008).

[68] ZMANDA INC. Amanda 3.3.6. `http://amanda.zmanda.com`, July 2014.

[69] ZWICKY, E. D. Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not. In *Proceedings of the 5th USENIX Conference on System Administration* (1991), LISA.

[70] ZWICKY, E. D. Further Torture: More Testing of Backup and Archive Programs. In *Proceedings of the 17th USENIX Conference on System Administration* (2003), LISA.

Systematically Exploring the Behavior of Control Programs

Jason Croft, *UIUC*
Ratul Mahajan, *Microsoft Research*
Matthew Caesar, *UIUC*
Madan Musuvathi, *Microsoft Research*

Abstract— Many networked systems today, ranging from home automation networks to global wide-area networks, are operated using centralized control programs. Bugs in such programs pose serious risks to system security and stability. We develop a new technique to systematically explore the behavior of control programs. Because control programs depend intimately on absolute and relative timing of inputs, a key challenge that we face is to systematically handle time. We develop an approach that models programs as timed automata and incorporates novel mechanisms to enable scalable and comprehensive exploration. We implement our approach in a tool called DeLorean and apply it to real control programs for home automation and software-defined networks. DeLorean is able to finds bugs in these programs as well as provide significantly better code coverage—up to 94% compared to 76% for existing techniques.

1 Introduction

Control programs that orchestrate the actions of "dumb" devices are becoming increasingly popular. The number of such devices, including locks, thermostats, motion sensors, and packet forwarding switches, is projected to grow beyond 50 billion by 2020 [10]. In a similar vein, software-defined networking (SDN) has become a multi-billion dollar market.

While control programs for these devices may be specified using simple languages (e.g., ISY [14], in the case of home automation), reasoning about their correctness is an incredibly complex task. The programs can have complex interactions across rules due to shared variables and device states. Further, time plays an important role in program behavior, as the behavior can change with the time of day or the time between occurrences of certain events. System behavior is often programmed directly to depend on time, in terms of policies (e.g., different actions during day vs. night) and protocol behavior (e.g., DHCP leases). Therefore, the behavior of these control programs is hard to verify by running the program a few times (e.g., during development) and, as a result, many bugs are discovered in production. These bugs can compromise the safety, security, and efficiency of the system.

One method of uncovering bugs is to systematically explore program behavior using model checking. However, prior work [6, 12, 15, 18, 22] does not address an important aspect of program behavior, specifically time. Instead, these tools abstract away time and, as a result, assume timers of different periods can fire at any time and in any order. Similarly, comparisons involving time can nondeterministically return true or false. Such an imprecise analysis of time is unacceptable for control programs because, as we show later, it generates many states that are not reachable in practice. This can force developers to sort through many false positive bugs reported by these tools. Furthermore, by abstracting time, these tools preclude developers from verifying correctness properties involving time (e.g., that timers fire at the correct time and under the correct conditions). Tools that use coarse heuristics to model time [23] eliminate false positives at the expense of incomplete exploration.

Accurately modeling time when exploring program behavior is a non-trivial problem. The challenge arises because events can occur at any time. To explore all possible behaviors, in theory, we must study all possible events occurring at all possible times. However, this is an ill-defined concept since time is continuous. We describe in §2.1 why circumventing this issue by naively discretizing time is unsatisfactory.

We investigate the use of timed automata (TA) [2] to systematically explore the behavior of control programs. TAs have been previously used to verify models of real-time systems. A TA is a finite state machine extended with real-valued (not discrete) virtual clocks. TA transitions can specify constraints on clock variables. For instance, a timeout transition should happen only when a particular clock variable is greater than a constant. The analyzability of TAs arises from the fact that, under certain conditions on clock constraints, one can define a finite number of *regions* [2]. All program states within a region are equivalent with respect to the untimed behavior of a system. Thus, "all possible times" can be safely translated to "all possible regions."

Prior work [4, 24] on exploring temporal behavior with TAs analyzes only an abstract model of a program or system. However, errors can be introduced in the model if it does not faithfully capture the behavior of the program, and the model can "drift" as the program evolves during development [18]. In this paper, we focus on using TAs to verify temporal properties of actual code. In particular, we ask: *can TAs be used to analyze executable programs? If so, what are the limitations of applying this theory to practice?*

Exploring the temporal behavior of a program without the need to first derive a TA introduces several new challenges. A TA-based exploration requires the set of temporal constraints that appear in the program. We develop a method to extract this information using program analysis [16]. As with many verification techniques, TA-based exploration inherits the state space explosion problem. As prior work is limited to exploring only abstract models, most heuristics to reduce the state space assume full knowledge of the model and cannot be used in our exploration. We develop three new techniques to boost exploration efficiency: (i) reducing the number of clock variables in the program, to cut down the number of regions we must explore; (ii) exploring the program as multiple, independent control loops; and (iii) predicting the response of the program to certain events, reducing the number of times we must execute the program.

We implement our approach for TA-based exploration in a tool called DeLorean and evaluate it in two diverse domains: home automation (HA) and SDNs. We explore 10 real HA programs and 3 SDN applications. Though DeLorean does not completely bridge the gap between the theory and practice of exploring real code with TAs, we see measurable benefit. DeLorean finds bugs uncovered by existing verification tools and new bugs that cannot be uncovered with existing techniques. We find we can achieve higher fidelity in exploring behavior, resulting in improved state and code coverage. We achieve up to 94% code coverage, compared to 76% in existing techniques that explore temporal behavior [23].

2 Background and Motivation

Many networked systems today are logically centralized. An HA system is composed of a controller and devices such as light switches, motion sensors, and locks. The controller receives notifications from the devices (e.g., when motion is sensed), can poll them for their current state (e.g., current temperature), and can send them commands (e.g., turn on the light switch). It uses these capabilities to coordinate the devices. Similarly, in SDNs, a controller manages the operation of switches by configuring them to forward packets as desired. The switches inform the controller when they receive packets for which forwarding actions have not been configured.

At the core of logically centralized control systems is a control program that determines its behavior. While the implementation languages for different systems and domains are different, control programs have a common structure. Their operation can be understood in terms of a set of rules. Each rule has a trigger and associated actions. A trigger is either an event in the environment (e.g., sensed motion, arrival of a packet) or a firing timer. Actions include setting a device state (e.g., turn on the light, installing a new rule in a switch) or a variable and setting timers. Actions can be conditioned on device state, variable and timer values, and time of the day. Programs are single-threaded and each rule runs until completion before another is processed.

Figure 1 shows an example program with three rules. Assume that the user wants to turn on the front porch light when motion is detected and it is dark out, and to automatically turn off this light after 5 minutes if it is daytime. Rule 1 is triggered when motion is detected by the front porch motion sensor. It turns on the light if motion is detected twice within 1 second and the light level sensed by a light meter is less than 20. The first condition is a heuristic to filter out false positives in motion sensing, and the second ensures that light is turned on only when it is dark. Rule 1 also updates the time when motion was last detected. Rule 2 is triggered when the front porch light goes from off to on (either programmatically or through human action) and sets a timer for 5 minutes. Rule 3 is triggered when this timer fires, and turns off the light if the current time is between 6 AM and 6 PM.

2.1 Reasoning about Program Correctness

The correctness of control programs can be hard to reason about. Even if individual rules are simple, reasoning about the program as a whole can be difficult because of complex interactions across rules. These interactions arise from shared state across rules due to the state of variables and devices. Thus, the program's current behavior depends not only on the current trigger but also on the current state, which in turn is a function of the sequence and timings of rules triggered in the past. This dependency and the number of possible sequences makes predicting program behavior difficult.

As an example, even the simple program in Figure 1 has a behavior that may not be expected by the user. Suppose the light is turned from off to on at 9:00 PM either due to sensed motion or by the user, triggering Rule 2. Then, the user walks on to the front porch at 9:04:50 PM, triggering Rule 1. This user might expect the light to stay on for at least 5 minutes, but it goes off unexpectedly 10 seconds later (at 9:05 PM). The fix here is of course to reset the timer in Rule 1, but that may not be apparent to the user until this behavior is encountered in practice.

Control programs are not the only ones whose correctness is difficult to reason about; the same holds true for almost all real-world programs such as network protocols and distributed systems. As a result, in a range of settings, researchers have developed a variety of techniques and corresponding tools, called model checkers, to automatically explore program behavior [6, 18].

Complex dependence on time. The behavior of control programs can depend intimately on time, both absolute time and the relative timing of triggers. For instance, the behavior of the program in Figure 1 depends on the time

```
1   PorchMotion.Detected:          /* Rule 1 */
2       if (Now - timeLastMotion < 1 secs
3           && lightMeter.LightLevel < 20)
4           FrontPorchLight.Set(On);
5       timeLastMotion = Now;
6   FrontPorchLight.StateChange:  /* Rule 2 */
7       if (FrontPorchLightState == On)
8           timerFrontPorchLight.Reset(5 mins);
9  timerFrontPorchLight.Fired:    /* Rule 3 */
10      if (Now.Hour > 6 AM && Now.Hour < 6 PM)
11          FrontPorchLight.Set(Off);
```

Figure 1: An example home automation program.

```
1  Trigger0:
2    timeTrigger0 = Now;
3    timeTrigger1 = Now;
4    trigger1Seen = false;
5  Trigger1:
6    if (Now - timeTrigger0 < 5 secs)
7      trigger1Seen = true;
8  Trigger2:
9    if (trigger1Seen)
10     if (Now - timeTrigger1 < 2 secs)
11       DoOneThing();
12     else
13       DoAnotherThing();
```

Figure 2: An example home automation program.

of day and on how close in time two motion events fire.

Existing model checkers do not systematically model time. Most [6, 15, 18] perform *untimed model checking*. They completely abstract time (in the interest of scalability) and do not maintain temporal consistency. During exploration, calls to gettimeofday() return random values and timers can fire in any order, regardless of their values. This can lead to many false positives (§5.4), i.e., bad states that will not arise in practice. False positives can be highly problematic, and often worse than missing errors [23], because they can send developers on a wild goose chase. Equally important for our context, since time is abstracted, untimed model checkers cannot verify time-related properties of a system, which are of prime interest for control programs.

One model checker that maintains temporal consistency is MoDist [23]. It has a global virtual clock, which is used to return values for gettimeofday(). Timers are fired in order and, when they do, the virtual clock is advanced accordingly. It uses static analysis of program source to infer all timers, including implicit timers. (Line 2 of Figure 1 represents an implicit timer that is set in Line 5 to expire in 1 second. Line 10 checks if the timer has fired, and the program behaves differently for the two cases.) During exploration, MoDist explores two cases, one in which the timer has expired and one in which it has not. In each, the clock value is set appropriately to a value that is consistent with the explored case.

While MoDist's approach does not produce false positives, it does not comprehensively explore all possible behaviors because exploring both cases for timers is not enough. Consider the simple example in Figure 2. This program has three triggers: Trigger0 resets the control loops; Trigger1 is considered as seen if it occurs within 5 seconds of Trigger0; and Trigger2 does different things depending on whether it occurs within 2 seconds of Trigger0. Assume that when Trigger0 fires, the virtual clock time of MoDist is T (seconds). While exploring Trigger1, to cover both cases MoDist will select one virtual clock time in the range $[T, T+5)$ and one greater than $T+5$. But now it has a problem: while exploring Trigger2, it can only explore one of the two branches (Line 10 or 12) and not both. If it had picked $T+1$ in the first case, it cannot explore the path on Line 13; if it had picked $T+3$, it cannot explore the path on Line 11.

Note that at the point of exploring Trigger1, MoDist has no reason to believe the specific selection in the range $[T, T+5)$ matters. All choices lead to the same program state and paths, and only later the choice has an impact. This is just one simple example; in reality, temporal constraints in the program can be highly complex (e.g., the same timer may drive behavior in multiple places).

Without systematic modeling of the temporal behavior of the program, the only way MoDist can explore all possible program behaviors is to explore all possible times of all possible triggers. But "all possible times" is ill-defined because time is continuous. We could discretize time and assume events happen only at discrete moments. But picking a granularity of discretization is tricky—if it is too fine, the exploration will have too much overhead as we would explore too many event occurrences; if it is too coarse, the exploration will miss event sequences that occur at finer granularity in practice and lead to different behaviors. Simply picking the smallest time-related constant in the program is also not enough [2]. Thus, there appears no satisfactory way to pick a granularity that works for all events and programs.

We thus systematically reason about time by exploring the control program as a timed automaton (TA) [2]. This lets us carve time into equivalence regions such that the exact timing of events within a region is immaterial. Thus, instead of exploring all possible times, it suffices to explore all possible regions.

Time-bound correctness properties. Untimed model checkers find violations of properties such as liveness (i.e., the system will *eventually* enter a good state) and safety (i.e., the system *never* enters a bad state). Since these tools abstract time, they cannot verify properties involving concrete time. For example, consider an SDN program caching mappings of ports to MAC addresses. Entries should expire a certain period after their last access. An untimed model checker can prove the entry ex-

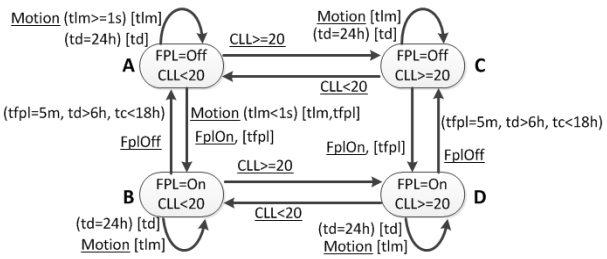


Figure 3: A TA for the program in Figure 1.

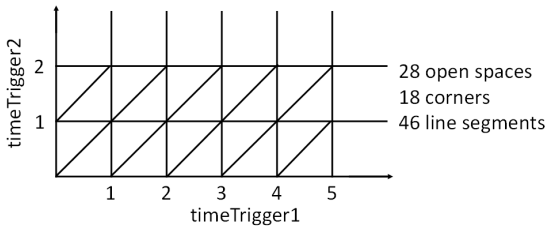


Figure 4: Time regions for the example in Figure 2.

pires after a certain sequence of events, but not that it expires after a certain period of time. To prove such a property, we must prove *time-bound liveness*. Furthermore, untimed model checkers cannot verify correctness properties based on absolute time, such as a light turning on at the right time or never being on at a certain time. In our evaluation of HA programs, we find three bugs (P9-1, P9-2, and P10-1 in §5.5) with this type of invariant.

2.2 Timed Automata

To reason systematically about time, we use timed automata [2] to guide our exploration. TAs are finite state machines extended with real-valued virtual clocks (VC), where a VC represents time elapsed since an event. (Wall clock time is simply one possible VC, which measures elapsed time since Jan. 1, 1970.) The state of a TA is the combination of the state of the underlying finite state machine together with the values of all VCs. A TA transition changes the machine state and may reset one or more VCs. Each transition specifies a set of clock constraints and is enabled from states that satisfy the constraint.

Figure 3 shows a TA that captures the behavior of the program in Figure 1. There are four states, corresponding to the Cartesian product of whether the front porch light (FPL) is on or off and the current light level (CLL). The TA uses three VCs to capture the time since *i*) the last motion (tlm), *ii*) the light was turned on (tfpl), and *iii*) midnight (td). Transitions are labeled with their triggers (underlined), the clock constraints (in parenthesis), and the clocks that are reset (in brackets). Motion denotes motion, and FplOn and FplOff denote the physical acts of manipulating the light. Some transitions have multiple labels, one for each situation where the TA can go from the source to the sink state. Transitions that have no triggers are taken as soon as the clock constraints are met.

In general, systematic exploration of TAs is infeasible, even in theory, as VCs hold non-discrete real values. However, the seminal work on TAs [2] shows that an exhaustive exploration is feasible provided the VC constraints obey certain conditions. The conditions are that arithmetic operations cannot be performed between two VCs and a VC cannot be involved in a multiplication or division operation. But adding or subtracting constants to VCs is allowed, and so is comparing two VCs (potentially after adding or subtracting constants).

Under these conditions the possible behaviors of the TA can be discretized into regions, such that the (uncountably many) states in a region behave the same with respect to the correctness properties of the TA. How such regions emerge can be intuitively understood if one observes that for the TA in Figure 3, after a motion event, the future behaviors are determined by whether the succeeding motion event occurs before or after 1 sec. The exact timing of the second motion event is not critical. Regions exist in multi-dimensional space where each dimension corresponds to one VC, and a point in the space represents concrete values of all the VCs. Regions encompass a set of points such that the exact point is immaterial for the purposes of comprehensive exploration.

The size of the region is proportional to the greatest common denominator (GCD) of constants in clock constraints, and hence the exploration will be faster if the GCD is larger. Regions get exponentially smaller as more VCs are included in the TA, because the plane for each pair of VCs divides open spaces into two parts. Once the regions are known, fully exploring the TA's behavior requires 1) exploring all possible transitions, in response to all possible triggers, from the current state; and 2) exploring exactly one *delay transition* in which there is no state transition but all VCs advance by the same amount. This amount is such that the time progresses to the immediately succeeding region. Figure 4 shows the regions for the example in Figure 2, which has two VCs. The constants in the clock constraints are 5 and 2, and thus the GCD is 1. We get 92 regions in this example.

3 Our Approach

Our goal is to systematically explore control program behavior. From a starting time and state, we want to predict all possible program behaviors. The exploration should be virtual so the actual state of the devices is not impacted. The output should be the set of unique states the system can be in, along with the sequence of events (i.e., triggers, actions) leading to that state.

3.1 Introducing Virtual Clocks

Control programs do not contain explicit references to VCs, but as mentioned previously, all time-related activ-

ities in effect manipulate VCs.

There are three kinds of time-related activities in control programs. The first is measuring the relative time between two events of interest (e.g., consecutive motion events). Here, a variable (e.g., timeLastMotion in Figure 1) is used to store the time of the first event, which is then subtracted from the wall clock time of the second event. To express this as a VC, we set the VC to zero when the first event occurs; the value of the VC when the second event occurs yields the delay, since VCs progress at the same rate as the wall clock unless reset.

The second time-related activity is a timer (e.g., timerFrontPorchLight in Figure 1). To capture this activity using a VC, we reset the VC when the timer is set, and queue a timer trigger to fire after the desired delay, after removing any previously queued event.

The third time-related activity is a sleep call, where actions for a rule are taken after a delay (e.g., turn on fan, sleep 30 seconds, turn it off). We express this by introducing a new timer and new rule. The actions of the new rule corresponds to post-sleep actions of the original rule. The sleep and post-sleep actions in the original rule are replaced by a timer that fires after the desired delay. In our treatment of sleep calls, if the trigger for the original rule occurs again before the timer set by an earlier occurrence fires, the post-sleep actions that correspond to the earlier trigger will not be carried out (because the earlier timer event will be dequeued). This behavior is consistent with the semantics of the systems we study.

3.2 Systematically Exploring Behavior

Given a control program and its starting state as input, our goal is to explore a given duration of wall clock time. A duration must be specified since wall clock time is unbounded and has an infinite number of regions. For programs with no dependency on the wall clock (e.g., many SDN applications), no duration is needed.

We assume that the program can be modeled as a TA. That is, all timers and variables that store time in program are, in effect, VCs. To leverage time regions, these VCs must satisfy the conditions mentioned above. We believe that these conditions are met in many contexts. They are certainly met in the different systems that we study in §5 and §6. The scripting languages of some of these systems cannot even express complex clock operations.

However, our exploration does not assume a TA has been derived from the actual code. Existing methods [4, 24] can explore the behavior of a TA, but deriving the entire TA corresponding to the program may not be feasible. Even the smallest of control programs can have extremely large TAs, as it needs to capture the program logic and its response to possible events.

Thus, we explore the TA dynamically, akin to how FSA-based model checkers dynamically explore the FSA instead of deriving the complete FSA of the program. From a starting program state, we repeatedly derive successor states resulting from triggers or delay transitions. For delay transitions, we must know the timed regions in advance to compute the delay amount. Fortunately, constructing regions does not require the complete TA, but only the constraints on the values of VCs [2]. We extract these constraints using analysis of program source.

```
1:  EndWC=Time.Now + FFduration;  ▷ How long to explore
2:  S0.WC = Time.Now;             ▷ Set the wall clock
3:  ES = {};                      ▷ explored states
4:  US={S0};                      ▷ unexplored states
5:  while US ≠ φ do
6:      Si = US.pop();
7:      ES.push(Si);
8:      for all e in Events, Si.EnTimers do
9:          So = Compute(Si, e);
10:         if !Similar(So, (US ∪ ES)) then
11:             US.push(So);
12:         end if
13:     end for
14:     if Si.EnTimers = φ then
15:         delay = DelayForNextRegion(Si.Region);
16:         if Si.WC + delay > EndWC then
17:             continue;
18:         end if
19:         So = Si.AdvanceAllVCs(delay);
20:         for all timer in So.Timers do
21:             if timer.dueTime >= So.WC then
22:                 So.EnTimers.Push(timer);
23:             end if
24:         end for
25:         if !Similar(So, (US ∪ ES)) then
26:             US.push((So, t));
27:         end if
28:     end if
29: end while
```

Figure 5: Pseudocode for basic TA exploration.

Figure 5 shows how we comprehensively explore program behavior. Assume we want to explore FFDuration of behavior, starting from the program state S_0. Program state includes the values of (non-time) variables, VCs, and enabled timers (i.e., ready to fire). We do a breadth-first exploration using a queue of unexplored states. Obtaining all successors of a state entails firing all possible events and all enabled timers. If a successor state is not similar to any previously seen state, we add it to the queue. Two states are similar if their variable values and set of enabled timers are identical and if their VC values map to the same region; VC values need not be identical since the exact time within a region does not matter.

If the state being explored has no enabled timer, it is eligible for a delay transition. This represents a period of time where nothing happens and time advances to the succeeding region. States with enabled timers need to fire all enabled timers before time can progress. We ig-

nore the successor if this delay takes us past the end time (i.e., starting wall clock time + specified duration). Otherwise, the successor state is computed by advancing all VCs. We treat wall clock time, which is virtualized during exploration, as any other VC except that it never resets; it tracks the progress of absolute time. We then check if any of the timers have been enabled because of this delay and mark them as such. The construction of time regions guarantees that no timers are skipped during the delay transition.

3.3 Achieving Scalable Exploration

The basic TA-based exploration above correctly handles time but is too slow to be practical. We use three general techniques to make it practical:

Predicting successor states Our first technique reduces the time to obtain successor states of a state being explored. We must first define the notion of *clock personality*. Two program states have the same clock personality if their values of all the VCs are equivalent with respect to all the clock constrains of the TA. Two program states can have the same clock personalities even if they are not in the same region.

If two states $S1$ and $S2$ with the same clock personalities have identical variable values and enabled timers, then any stimulus (i.e., combination of trigger and environmental conditions) will have exactly the same effect on both states. Thus, it is necessary to compute the successor of only one such state, say $S1$. The successor of $S2$ can be obtained from the successor of $S1$ while retaining the clock values of $S2$ for all VCs except those that are reset by the current stimulus.

Computing a successor requires deserializing the parent's state, running the program, subjecting it to the stimulus, and serializing the successor's state. These are costly operations. In contrast, prediction requires only copying the state and modifying VC values.

Independent control loops Our second optimization is based on the observation that large control programs may often be composed of multiple, independent control loops manipulating different parts of the program state. For instance, thermostats and furnaces may be controlled by a climate control loop, and locks and alarms by a security loop. These two may manipulate different variables and clocks, but otherwise share no state. In such cases, we can explore the loops independently, instead of exploring them jointly. Separate exploration is faster since joint exploration considers the Cartesian product of the values of independent variables and clocks. We use taint tracking to identify independent loops.

Reducing the number of clocks The number of VCs in the program has a significant impact on exploration efficiency because the size of regions shrinks exponentially with it. When transforming a control program, we should introduce the minimum number of VCs. We exploit two opportunities. First, consider cases where the actions in a rule have multiple sleeps, e.g., `action1; sleep(5); action2; sleep(10); action3`. Instead of using two timers (one per sleep), we can use only one because the two sleeps can never be active at the same time [7]. To retain the original dynamic behavior, we introduce a new program variable to track which actions should be taken when the timer fires. In the example above, when the rule is triggered, after `action1` is taken, this variable is reset to 0 and the timer is set to fire after 5 seconds. When the timer fires: *i*) if the variable value is 0, `action 2` is taken, the variable is set to 1, and the timer is set to fire after 10 seconds; *ii*) if the variable value is 1, `action 3` is taken.

Second, control programs often have daily action for different times of day (e.g., sunrise, one hour after sunrise). The straightforward translation is to introduce a new timer per unique activity. A more efficient method is to use one timer to conduct all such activities, using a method similar to the above — introduce an additional program variable to cycle through the different actions and reset it after the last action is conducted.

3.4 Theory-Practice Gap

Existing TA-based model checkers work with abstract models and assume the model is provided as input. In building the model incrementally and dynamically, we uncover several gaps in using TAs on real code.

A transition in a TA must occur instantaneously since time only progresses explicitly through a delay transition. In practice, however, the processing of an event (e.g., in response to motion occurring) may involve a non-trivial amount of time. In our implementation, we assume processing time is instantaneous, but propose a technique to handle events with non-trivial processing times. For each of these event handlers in the program, we can introduce a timer to expire after the expected processing time. When the timer is active and has not expired, the system is processing the event. If the timer is inactive, either because the timer has expired or has not been activated, the system is not processing the event.

In some systems, clocks may be created in response to events. In SDN programs, for example, flows installed in switches in response to a packet arrival at the controller introduce two new VCs—one each for the soft and hard timeouts. We can use symbolic execution to extract the clock constraints for all possible values of timeouts, but we cannot determine the number of occurrences of the event triggering this behavior. Rather, this is dependent on the number of times the event occurs along a path generating a specific state. We cannot add a new clock with a new constraint to the region construction, as we cannot change regions during exploration. Regions are constructed using GCD of the clock constraints and adding

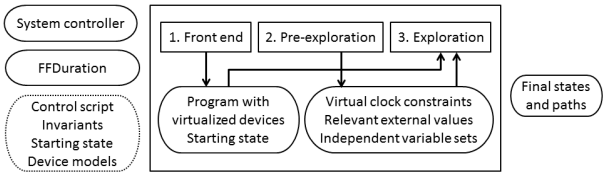


Figure 6: Overview of DeLorean.

a new clock mid-exploration could change the delay if the GCD changes. Instead, we pre-allocate a number of VCs, each with a specific constraint. During exploration, when a new VC needs to be created, it is allocated an available VC from a queue of pre-allocated VCs.

4 Design

We now describe the design of DeLorean, our TA-based model checker. As shown in Figure 6, the primary inputs to DeLorean is the control program—including a model of the controller (on which the program runs) and devices—and the duration of wall clock time to explore (FFDuration). If the program has no dependence on wall clock time, i.e., timers are relative, this parameter is not needed. The user can also specify three optional inputs. The first is a list of invariants on device states and system behavior, which should be satisfied at all times. These are specified in a manner similar to the *if* conditions in the rules and can include time. The second is the wall clock time. The third is the starting state of (a subset of) devices and variables from which exploration should begin.

The output of DeLorean is all the unique states of the devices. If invariants are specified and violated by any state during the exploration, we also output the state. In addition, DeLorean outputs the path that leads to each state—a timestamped sequence of triggers, along with the values of environmental factors during those firings.

DeLorean has three stages. First, the front end converts the program to one where clocks have been virtualized, using the method in §3.1. Second, pre-exploration analyzes this program to recover information required for the optimizations in §3.3. The final stage is the exploration itself.

4.1 Pre-Exploration

This stage analyzes the program produced by the front end to recover the information needed for constructing timed regions and implementing the optimizations in §3.3. Here, we use symbolic execution [16] of program source. Symbolic execution simulates the execution of code using a symbolic value σ_x to represent the value of each variable x. As the symbolic executor runs, it updates the symbolic store that maintains information about program variables. For example, after the assignment y=2x the symbolic executor does not know the exact value of y but has learned that σ_y=2σ_x. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression given the information in the store. The executor only explores the branch corresponding to the guard's value as returned by the constraint solver, ensuring infeasible paths are ignored. If there is insufficient information to determine the guard's value, both branches are explored. This produces a tree of all possible program execution paths. Each path is summarized by a path condition that is the conjunction of branch choices made to go down that path.

We symbolically execute the program's main control loop, which is the starting point for all processing activity. We configure the symbolic executor to treat the following entities as symbolic: program state (variables and clocks) and the parameters of the control loop. The output of the symbolic executor is the set of possible paths for each possible trigger. For each path, we obtain the i) constraints that must hold for the program to traverse that path, and ii) the program state that results after its traversal. The constraints and the resulting program state are in terms of input symbols, the entities we made symbolic in the configuration.

We can now recover the following information.

Virtual clock constraints These are required for constructing time regions and for predicting successor states. We obtain them from the output of symbolic execution by taking the union of constraints on VCs along each path. Additionally, program statements that reset a timer x to k secs are essentially clock constraints of the form $x \geq k$. We extract such statements from the program source and add corresponding constraints to the set.

Independent control loops We also use the output of symbolic execution for taint tracking. We analyze the program state that results from each path. If the final value of a variable along any path is different from its (symbolic) input value, that variable is impacted along the path. This impact depends on the input symbols that appear in the output value (data dependency) and path constraints (control dependency). The variables corresponding to those input symbols are tainting the variable.

We use this information to identify independent sets of variables and VCs. Two variables or VCs are deemed dependent if they either taint each other in the program, or they occur together in a user-supplied invariant (as we must do a joint exploration in this case as well). After determining pairwise dependence, we compute the independent sets that cover all variables and VCs.

4.2 Exploration

This stage implements the method outlined in §3. To start, it runs the program and initializes the starting state. We then checkpoint the program by serializing its internal state. The checkpoint captures the values of all variables, including time related variables, and the times

	type	#rules	#devs	SLoC	#VCs	GCD (s)
P1	OmniPro	6	3	59	2	7200
P2	Elk	3	3	75	2	1800
P3	MiCasaVerde	6	29	143	2	300
P4	Elk	13	20	193	5	5
P5	ActiveHome	35	6	216	14	5
P6	mControl	10	19	221	4	5
P7	OmniIIe	15	27	277	6	60
P8	HomeSeer	21	28	393	10	2
P9	ISY	25	51	462	6	60
P10	ISY	90	39	867	6	10

Table 1: The HA programs we study.

when various timers will fire.

We maintain a table that contains the values of the VCs of a state. Many states differ only in VC values—the successor state after a delay transition differs from the parent only in VC values, so does the successor that is predicted from another state. Maintaining this table separately lets us quickly obtain these successor states. It also helps reduce the memory footprint, since two states that differ only in VC values can share the same checkpoint. However, this implies that the VC values in a table can be out of sync with those embedded in the checkpoint. Thus, when restoring a state, we update its VC values from the table before any other processing.

4.3 Implementation

DeLorean is implemented with 10k+ lines of C# code. The bulk of this code implements the pre-exploration and exploration stage, which we developed from scratch. We could not use existing tools for exploring TAs [4, 24] because we do not have the complete TA for the program. Our implementation includes models of controllers and devices in the two domains we study—home automation (§ 5) and software-defined networks (§ 6).

For HA applications, we implemented front end modules for two systems—ISY [14] and ELK [9]. We chose these two because of their popularity. The front end parses ISY or ELK programs using ANTLR [3] and produces a C# program that captures the behavior of the program and contains additional variables, rules, and actions needed for modeling devices. As the state of these devices is typically simple and can be represented using boolean or integer variables, we can model the devices automatically from the ISY or ELK program.

The pre-exploration stage uses Pex [19] to symbolically execute the main event loop of this C# program. Pex is a modern symbolic execution engine that mixes concrete and symbolic execution ("concolic" execution) to boost path coverage and efficiency.

5 Case Study: HA Networks

To evaluate a TA-based exploration against existing verification techniques, we examine DeLorean in two environments: home automation networks and SDNs.

5.1 Domain-Specific Optimizations

A common behavior in HA is a dependence on environmental factors (e.g., temperature, light level) sensed by devices in the system. For a comprehensive evaluation, we must explore all combinations of values of external factors. To address this challenge, we build on prior work and combine symbolic execution with model checking [6]. We use symbolic execution of program source to infer equivalence classes of combinations of values of environmental factors. In Figure 1, for example, there are two equivalence classes, corresponding to light level values below or higher than 20. Then during exploration, we use one set of values per class, instead of having to explore all possible combinations of values. So, if a program depends on temperature and light level, for every trigger, its response must be explored with all combinations of temperature and light levels.

5.2 Dataset

We evaluate DeLorean using real HA programs. We solicited these programs on a mailing list for HA enthusiasts. We picked the 10 programs shown in Table 1. We selected them for the diversity of HA systems and the number of rules and devices. We see that most installations have tens of rules and devices, with the maximums being 90 and 51. This points to the challenge users face today in predictably controlling their homes. Collectively, these installations had 19 different types of devices, including motion sensors, temperature sensors, sprinklers, and thermostats.

The table shows the source lines of code (SLoC) and the number of VCs in the program after transformation in the first stage. Systems which we have not implemented a front end yet were transformed manually. We see that most installations have 5 or more VCs, indicating a heavy reliance on time. The table also shows the GCD (greatest common denominator) across all constants in VC constraints in the program. The GCD can be coarsely thought of as the detail with which the program observes the passage of time. Since the size of the regions depends on it, it also heavily influences the exploration time.

5.3 Exploration Performance

We run DeLorean over all 10 programs and conduct 20 trials, each with randomly selected starting state and time (since program behavior depends on both). All experiments use an 8 Core 2.5Ghz Intel Xeon PC with 16GB RAM. Table 2 shows the number of transitions and average CPU time needed to explore one hour of wall clock time for each program. We estimate DeLorean makes 200k transitions per second. Since HA programs depend on wall clock time, we can also measure the CPU time with respect to wall clock time. We also see that DeLorean can explore real programs 3.6 times to 36K times

	# Transitions	CPU Time (sec)	Reduction w/ Prediction
P1	72	0.10	-11.11%
P2	123	0.12	-20%
P3	178	0.15	-7.14%
P4	19.7M	176.10	75.16%
P5	78.7K	1.03	61.28%
P6	51K	1.04	48%
P7	36.M	17.87	89.53%
P8	8.1M	89.50	84.36%
P9	121M	793.90	95.24%
P10	256M	998.0	83.5%

Table 2: Performance for exploring one hour of wall clock time and the reduction in CPU time from predicting states.

faster than wall clock time.

An important element in obtaining quick explorations for these programs is predicting successor states. While this is a general optimization for dynamically exploring TAs, its effectiveness depends on how often we encounter non-similar states with identical clock personalities, variables, and enabled timers. To evaluate it, Table 2 show the percentage reduction in CPU time when prediction is used compared to when it is not used. For the smallest programs, prediction leads to slower exploration. This is because in such cases the overhead associated with checking for past states that can be used for prediction is greater than any benefit it brings. However, for larger programs, prediction brings substantial benefit. For P9, prediction cuts the exploration time by 95%, i.e., exploring without prediction is slower by a factor of 20.

5.4 Comparison with Alternatives

Untimed exploration As mentioned earlier, current model checkers ignore time and can thus generate invalid program states that will not be generated in real executions. If there were just a few, it is conceivable that users would be willing to put up with occasional incorrectness. However, we find that untimed exploration results in many incorrect states. Figure 7 shows the percentage of additional, invalid states produced by untimed model checking,[1] when beginning from the same starting state as DeLorean and running until it cannot find any new states. Untimed exploration differs from DeLorean in three aspects: *i*) in addition to successors based on device notifications, each state has successors based on each queued timer, independent of the target time of the timer; *ii*) if a comparison to time is encountered during exploration both true and false possibilities are considered; *iii*) there are no delay transitions. The graph averages results over 10 paired trials with different starting

[1]This comparison based on invalid states alone hides one additional limitation of untimed model checking. Untimed exploration is incapable of verifying program behaviors that depend on time (e.g., light turned off a second after turning on).

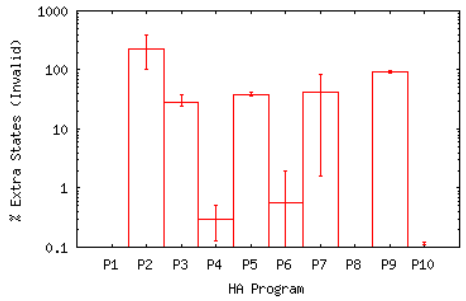


Figure 7: Invalid states generated by untimed exploration.

inputs, and the error bars shows maximum and minimum percentage of invalid states.

We see that untimed exploration produces a significant number of invalid states. For most programs, the number of invalid states is of the same order as the number of valid states produced by DeLorean. Closer inspection of results from untimed exploration provides insight into how some invalid states are produced. One common case is where devices such as lights are programmed to turn on in the evening, using a timer. Because timers can fire anytime, untimed model checking incorrectly predicts that the light can be off in the evening, which will not happen in practice. Another case is where certain actions are meant to occur in a sequence, e.g., open the garage door after key press and then close it 5 minutes later. With DeLorean, these actions are carried out in the right sequence, correctly predicting that the door is left in the closed state. But both possible sequences are explored by untimed exploration, one which incorrectly predicts that the garage door is left open.

MoDist Unlike untimed exploration, MoDist maintains temporal consistency during exploration, but at the expense of incomplete exploration (§2.1). To illustrate this, we implement MoDist's algorithm for exploring timers in DeLorean and compare it with our exploration. We compare two metrics–state coverage and code coverage. State coverage measures the number of unique program states explored and code coverage measures the number of lines of code exercised during exploration. Figures 8 and 9 show code and state coverage, respectively, for MoDist and DeLorean averaged over 24 trials, each exploring one hour. Programs omitted in Figure 8 have equivalent coverage in MoDist and DeLorean.

5.5 Unintended Behaviors

To informally gauge DeLorean's ability to find such behaviors, we inspect comments in two of the programs (P9, P10) and turn them into invariants for which DeLorean should report violations. We find four violations.

P9-1 A comment indicated the lights in the back of the house should turn on if motion is detected in the evening (i.e., sunset to 11:35PM). But DeLorean found that the

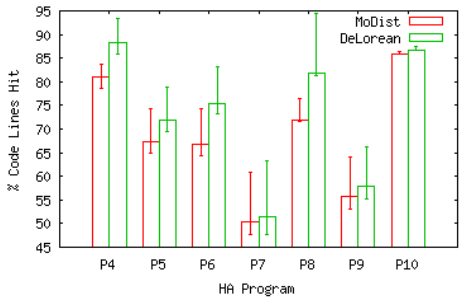


Figure 8: Code coverage.

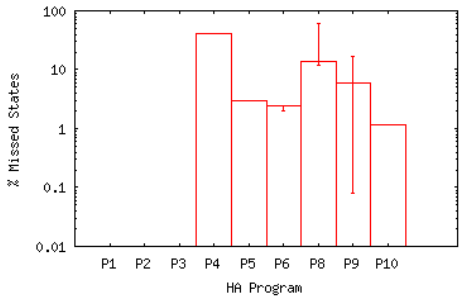


Figure 9: States missed by MoDist.

lights could be on even if there was no motion. A rule appears misprogrammed—instead of using conjunction as the condition to turn on the light (sunset $<$ Now $<$ 11:35PM && MotionDetected), it was using disjunction (sunset $<$ Now $<$ 11:35PM || MotionDetected)

P9-2 A comment indicated the front porch light should stay on from a half hour after sunset until 2AM. There were two rules to implement this invariant: one turning the light on at a half hour after sunset and one turning it off at 2AM. But DeLorean found cases where the light was off in that time window. Inspection revealed another rule to turn off the light at 7:45PM. Thus, the invariant is violated if sunset occurs after 7:15PM, which can happen where the user of P9 resides. Exploring sunset values higher than 7:15PM uncovered the violation.

P10-1 A comment indicated the user wanted to turn on a dimmer switch in the master bath room when motion is detected. But we found instances where the motion occurred but the dimmer was not on. Inspection revealed that the user's detailed intent, implemented using two rules, was to turn on the dimmer half-way when motion occurs during the day, and to turn it on fully when its detected during night. But the way day and night time periods were defined left a 2 minute gap where nothing would happen in response to motion.

P10-2 A comment indicated the user wanted to treat three devices identically (i.e., all on or all off). Inspection of a violation of this invariant showed that while three of the four rules that involved these devices correctly manipulated them as a group, one rule had left out a device.

	SLoC	#VCs	GCD (s)	#trans 1 VC	#trans 2 VCs	#trans 4 VCs
PySwitch	234	13	1	6210	49k	8.8M
LoadBalancer	2063	14	2	351k	512k	3.8M
EnergyTE	434	10	5	442k	1.7M	21M

Table 3: The OpenFlow programs we study.

6 Case Study: SDN

To further demonstrate the value of TA-based exploration, we model and test SDN programs in DeLorean. Similar to NICE [6], we create a model of the NOX platform in C#, including the controller, switches, and hosts. We discover relevant packet headers during pre-exploration, using Pex to symbolically execute the event handlers that make up the OpenFlow program. Since OpenFlow switches have complex internal behavior (e.g., flow tables) that we cannot observe externally, we manually define models of OpenFlow switches and hosts. OpenFlow programs have no dependency on absolute time, therefore we use no wall clock time.

6.1 Dataset

We evaluate DeLorean using three real programs—a MAC-learning switch (PySwitch), a web server load balancer[21], and energy efficient traffic engineering (REsPoNse) [20]. We manually translate the programs from Python into C# for testing in DeLorean. Table 3 shows the source lines of code (SLoC), the total number of VCs that can be dynamically created during an exploration, and the GCD of the clock constraints. As in NICE, we bound the state space by limiting certain events, such as the number of times a host sends a packet.

Each program has dependencies on relative time. PySwitch, for example, uses a timer to periodically check entries in a cache of MAC address-port mappings and expire entries older than a specific time. In this case, a VC is needed to express the timer scheduling the periodic check, and another VC for each entry in the cache.

6.2 Comparison with Alternatives

We compare DeLorean to NICE, a model checker for OpenFlow programs. Similar to untimed model checking, NICE does not systematically model time. Instead, application-specific heuristics are used to trigger timers in each of the SDN applications tested. We construct a model of the NOX platform for DeLorean and simulate NICE's exploration by running DeLorean with no VCs. We also implement NICE's heuristics for exploring timer behavior. To informally gauge DeLorean's ability to find unintended behaviors, we create invariants from the 11 bugs discovered by NICE. We find DeLorean can reproduce violations for all 11 bugs.

We now compare DeLorean's coverage of a program's

	% Missing			% Incorrect		
	1 VC	2 VCs	4 VCs	1 VC	2 VCs	4 VCs
Pyswitch	0%	34%	84%	0%	0%	0%
Loadbalancer	95%	95%	95%	117%	123%	123%
EnergyTE	69%	87%	97%	26%	12%	46%

Table 4: Missing and invalid states generated by NICE, compared to explorations in DeLorean using 1, 2, and 4 VCs.

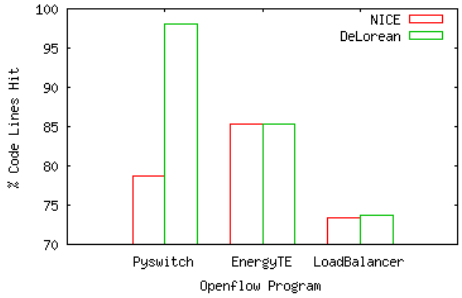


Figure 10: Code coverage in DeLorean and NICE.

state space to that of NICE. In Figure 10, we show the code coverage of DeLorean and NICE for the three programs. In PySwitch, NICE does not explore the timer for periodically checking cache entries, therefore missing an entire function.

In Table 4, we show the number of missed and incorrect states generated by NICE compared to explorations in DeLorean using 1, 2, and 4 VCs. Because NICE does not trigger any timers in PySwitch, it misses potential behavior, but does not introduce any invalid states that would be generated from timers firing at incorrect times from incorrect states The heuristic for trigger timers in the EnergyTE application, however, fires timers from every possible state, resulting in invalid states. Similarly, in LoadBalancer, timers can also fire from invalid states.

Further, NICE's heuristics do not test the expiration of flows. Correctly exploring this behavior requires, and verifying correctness properties related to flow expirations, requires more systematic treatment of time. This results in missed states in both the EnergyTE and LoadBalancer applications.

We see that with non-systematic treatment of time, program exploration can introduce false behaviors or miss potential behaviors. In programs dependent on absolute and relative time, such as HA programs, we find untimed exploration can produce too many invalid states to be useful. Even in programs with dependencies only on relative time, such as SDN programs, we see non-systematic treatment of time can also produce as many invalid states as valid states.

7 Related Work

Our work builds on progress the research community has made towards verifying the behavior of real systems.

Model checking programs One class of techniques is model checking, where programs are modeled as FSAs and their behavior is comprehensively explored [12, 15, 18]. Recent work, like us, also combines model checking with symbolic execution [5, 8, 22]. However, most model checking work ignores time. This approach works well for programs that have a weak dependence of time, but the behavior of control programs that we study is intricately linked with time. Ignoring time in such programs can lead to exploring infeasible executions, and it cannot discover unexpected behaviors in which the mismatch is the time gap between events. One exception is NICE, which studies OpenFlow applications whose behavior can vary considerably based on packet timings [6]. However, its treatment of time is not systematic and instead relies on heuristics to explore timer behavior.

Model checking using TA There has been much work on TA-based model checking in the real-time systems community. It includes developing efficient tools to explore the TA [4, 24] as well as transformations that speed explorations [7, 13]. This body of work assumes that the entire TA is known in advance, and it does not target program analysis. While we draw heavily on the insights from it, to our knowledge, our work is the first to use TA to model check programs. We describe general methods to dynamically and comprehensively explore program executions and techniques to optimize exploration.

Other debugging techniques Explicit state model checking, which we use, is complementary to other program debugging approaches. Record and replay [17] can help diagnose faults after-the-fact and is especially useful for non-deterministic systems; in contrast, we want to determine if faults can arise *in the future*. There has also been work on "what-if" analysis in IP networks, e.g., with the use of shadow configurations [1] and route prediction [11]. These focus on computing the outcomes of configuration changes; in contract, we study the dynamic behavior of more general programs.

8 Conclusions

Mistakes in control programs can impact the safety and efficiency of their system. We develop a technique using timed automata to systematically explore program behavior and verify temporal properties. We implement our approach in a tool named DeLorean and apply it to two domains where timing in control programs is important—home automation and software-defined networks. We show it results in higher fidelity analysis, including better state and code coverage, than existing techniques that do not systematically model time.

References

[1] ALIMI, R., WANG, Y., AND YANG, Y. Shadow configuration as a network management primitive. *ACM SIGCOMM* (August

2008).

[2] ALUR, R., AND DILL, D. L. A theory of timed automata. *Theoretical Computer Science* (1994).

[3] ANTLR parser generator. `http://antlr.org/`.

[4] BENGTSSON, J., LARSEN, K., LARSSON, F., PETTERSSON, P., AND YI, W. UPPAAL: A tool suite for automatic verification of real-time systems. *Hybrid Systems III* (1996).

[5] CANINI, M., JOVANOVIC, V., VENZANO, D., SPASOJEVIC, B., CRAMERI, O., AND KOSTIC, D. Toward online testing of federated and heterogeneous distributed systems. In *USENIX ATC* (2011).

[6] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A NICE way to test OpenFlow applications. In *NSDI* (2012).

[7] DAWS, C., AND YOVINE, S. Reducing the number of clock variables of timed automata. In *Real-Time Systems Symposium* (1996).

[8] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *NSDI* (2014).

[9] ELK products, inc. `http://www.elkproducts.com/`.

[10] EVANS, D. The Internet of things. `http://blogs.cisco.com/news/the-internet-of-things-infographic/`, 2011.

[11] FEAMSTER, N., AND REXFORD, J. Network-wide prediction of BGP routes. *IEEE/ACM Trans. Networking* (April 2007).

[12] GODEFROID, P. Model checking for programming languages using verisoft. In *POPL* (1997).

[13] HENDRIKS, M., AND LARSEN, K. G. Exact acceleration of real-time model checking. In *Electronic Notes in Theoretical Computer Science* (2002).

[14] Universal devices products/insteon/isy-99i series. `http://www.universal-devices.com/99i.htm`.

[15] KILLIAN, C. E., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

[16] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976).

[17] LEBLANC, T., AND MELLOR-CRUMMEY, J. Debugging parallel programs with instant replay. *IEEE Transactions on Computers 36* (1987).

[18] MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., AND DILL, D. L. CMC: A programatic approach to model checking real code. In *OSDI* (2002).

[19] TILLMANN, N., AND DE HALLEUX, J. Pex: White box test generation for .NET. In *Tests and Proofs (TAP)* (April 2008).

[20] VASIĆ, N., NOVAKOVIĆ, D., SHEKNAR., S., BHURAT, P., CANINI, M., AND KOSTIĆ, D. Identifying and using energy-critical paths. In *CoNEXT* (2011).

[21] WANG, R., BUTNARIU, D., AND REXFORD, J. OpenFlow-based server load balancing gone wild. In *Hot-ICE* (2011).

[22] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI* (2009).

[23] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MoDist: Transparent model checking of unmodified distributed systems. In *NSDI* (2009).

[24] YOVINE, S. Kronos: A verification tool for real-time systems. *Int'l Journal of Software Tools for Technology Transfer* (1997).

Fence: Protecting Device Availability With Uniform Resource Control

Tao Li[†] Albert Rafetseder[†] Rodrigo Fonseca[*] Justin Cappos[†]
[†]*Polytechnic Institute of New York University* [*]*Brown University*

Abstract

Applications such as software updaters or a run-away web app, even if low priority, can cause performance degradation, loss of battery life, or other issues that reduce a computing device's *availability*. The core problem is that OS resource control mechanisms unevenly apply uncoordinated policies across different resources. This paper shows how handling resources – e.g., CPU, memory, sockets, and bandwidth – in coordination, through a unifying abstraction, can be both simpler and more effective. We abstract resources along two dimensions of fungibility and renewability, to enable resource-agnostic algorithms to provide resource limits for a diverse set of applications.

We demonstrate the power of our resource abstraction with a prototype resource control subsystem, Fence, which we implement for two sandbox environments running on a wide variety of operating systems (Windows, Linux, the BSDs, Mac OS X, iOS, Android, OLPC, and Nokia) and device types (servers, desktops, tablets, laptops, and smartphones). We use Fence to provide system-wide protection against resource hogging processes that include limiting battery drain, preventing overheating, and isolating performance. Even when there is interference, Fence can double the battery life and improve the responsiveness of other applications by an order of magnitude. Fence is publicly available and has been deployed in practice for five years, protecting tens of thousands of users.

1 Introduction

Unfortunately, it is still common for end-user devices to suffer from unexpected loss of *availability*: applications run less responsively, media playback skips instead of running smoothly, or a full battery charge lasts hours less than expected. The cause might be a website running JavaScript [5, 6], a software updater overstepping its bounds [10], an inopportune virus scan [20, 27], or a file sync tool such as Dropbox indexing content [3, 7]. A buggy shell that leaks file descriptors can prevent other applications from running correctly [14]. Moreover, the cause may be unwanted apps or functionalities bundled with a legitimate application, such as advertisements [71] or hidden BitCoin miners [18]. Malicious applications may even attempt to overheat the device to cause permanent damage [54].

Although dealing with resource contention is a problem as old as multiprogrammed operating systems, today's platforms present renewed challenges. Key properties of a system such as battery life, application performance, or device temperature depend on multiple resources. This, along with the resource-constrained nature of mobile devices and the number and ease of installing applications from "app stores," exacerbates the problem.

In many cases, one cannot simply identify and kill processes that consume too many resources, because many programs execute useful-but-gluttonous tasks. For example, a web browser may execute inefficient JavaScript code from a site that slows down the device. However, the browser overall consists of many interrelated processes that render and execute code from different sites. Selectively containing resource usage is much better than killing the browser.

Our work addresses the problem of improving device availability in the presence of useful-but-gluttonous applications that can consume one or more types of resources. We introduce a non-intrusive mechanism that mediates and limits access to diverse resources using *uniform resource control*. Our approach has two parts: a unifying resource abstraction, and resource control. We abstract resources along two dimensions, allowing uniform reasoning about all system resources, such as CPU, memory, sockets, or I/O bandwidth. The first dimension classifies each resource as *fungible* (i.e., interchangeable, such as disk space) or *non-fungible* (i.e., unique, such as a specific TCP port). The second dimension classifies resources as either *renewable* (i.e., automatically replenished over time, such as CPU quanta) or *non-renewable* (i.e., time independent, such as RAM space). Using only these two dimensions, we are able to fully define a policy to control a specific resource. Adding a mechanism that quantifies and regulates access to each resource, either through interposition or polling, provides the desired level of control.

We demonstrate the feasibility of our approach by designing and building a resource control subsystem for sandbox environments, which we call Fence. Fence allows arbitrary limits to be placed on the resource consumption of an application process by applying a resource control policy consistently across resource types. This allows it to provide much better availability, when faced with a diverse set of resource-intensive applications, than would resource-specific tools. As we show in §5, when the system is under contention from a resource-hungry process,

our current user-space implementation of Fence provides double the battery life and a performance improvement of an order of magnitude relative to widely used OS-level and hardware solutions.

Fence is the first approach for controlling multiple resources across multiple platforms. By arbitrarily limiting resource consumption, Fence bounds the impact that applications can have on device availability as well as on properties such as heat or battery drain. Fence runs in user space, which makes it usable even within Virtual Machines (VMs) and on already deployed systems, where privileged access might be problematic and low-level changes could be disruptive.

2 Scope

Fence's controls work insofar as it can interpose on applications' requests for resources. As a proof of concept, Fence is targeted at application hosts, such as sandbox environments, VMs, browsers, or plugin hosts. The same unified policies could be implemented in an OS.

Most modern operating systems have mechanisms for identifying and limiting the resource usage of applications and processes [11, 15, 25]. However, as the prevalence of availability problems indicates, and we further demonstrate in §5, existing mechanisms do not adequately isolate resources among applications. A major cause of device availability problems is that there is no single resource where contention happens, which implies that any mechanism that controls a single resource will not be effective. Furthermore, because existing mechanisms use different abstractions and control points, it is very hard, if not ineffective, to write coordinated policies across different resources. Priority-based allocation and resource reservation systems (along with hybrids of the two) offer partial solutions in some cases, but the combined properties and guarantees are far from sufficient in practice. They apply ad-hoc and uncoordinated resource control across diverse resource types, and do not offer uniform functionality, abstractions, or behavior.

Recent developments in Linux's `cgroups` (cf. §7) provide a more unified approach to resource management, but are still restricted to one operating system. We implement Fence's resource control mechanisms – polling and interposition – at the user level, making it easily portable across many different platforms.

In this paper we assume that hog applications are "useful-but-gluttonous." That is, a hog application may attempt to consume a large amount of resources, but is otherwise desirable. Specifically, we do not attempt to protect against applications that perform clearly malicious actions, such as deleting core OS files, killing random processes, or installing key loggers (existing work addresses such issues). The desired outcome is that the user has control over properties such as the responsiveness, battery life, and heat for any set of applications, especially hogs.

Our main focus in this paper is how to provide the needed *mechanisms* for resource-control, not on providing a specific resource control *policy*. As a first step, we demonstrate the effectiveness of Fence with two simple policies, both assuming that we can identify and have the privilege to control the hog process: one in which we manually provide static limits (most scenarios in §5), and one in which the policy sets dynamic resource limits to achieve a desired battery lifetime (§5.5). We anticipate, however, that more sophisticated policies are possible and applicable in other scenarios.

Lastly, a note about who interacts with Fence. Fence's mechanisms are integrated into a platform, e.g., a new sandbox environment, by the authors of the environment. Policies, on the other hand, can be written by the same authors, by third parties such as machine administrators, or by end-users. This paper does not specify higher-level interfaces for specifying policies. Fence should be transparent to applications running within the environment.

3 Managing Resources

In this section, we present Fence's resource abstraction, and discuss how using a simple classification of resources, along the dimensions of *fungibility* (§3.1) and *renewability* (§3.2), makes it possible to control their usage in a unified way. We assume a simple policy of limiting resource consumption by imposing a quota, given in terms of either an absolute quantity or a utilization.[1] Following that, §3.3 explains how control is enforced on resources, and §3.4 discusses choosing resource limit settings.

3.1 Fungible and non-fungible

One way to characterize low-level resources is by whether or not they are *fungible*, i.e., whether one instance of the resource can stand in for or indiscernibly replace any other instance. A fungible resource is something like a slot in the file descriptor table. It does not matter to the application that accesses a file where exactly the slot maps into the kernel's table. However, overconsumption of the resource can cause stability issues [14]. Fungible resources only need to be counted. The maximum allowed utilization by an application is capped by a quota. Fungible resources can be managed by maintaining, according to the resource's usage, an available quantity relative to a quota.

Lines 1, 2 and 3 of Table 1 summarize how gaining access to fungible resources works. As long as the requested resource quantity is within the quota, access is granted and the consumed quantity is logged. If a request exceeds the quota, granting access depends on the renewability of the resource (defined in §3.2 below), and will either fail or block until the required quantity becomes available. This

[1] As we show in §5.5, this quota needs not be static.

#	Condition	Fungible	Renewable	Result
1	quantity $\leq$ quota	Yes	either	reduce quota, grant access
2	quantity $>$ quota	Yes	Yes	block until replenished
3	quantity $>$ quota	Yes	No	error
4	unallocated resource	No	either	allocate, grant access
5	busy resource	No	Yes	block until replenished
6	busy resource	No	No	error

Table 1: Accessing resources with different characteristics

will happen either over time (for a renewable resource), or through explicit release by the caller (for a non-renewable resource).

For a *non-fungible* resource, like a listening TCP or UDP port, each resource is unique. Each application may request one specific instance and will effectively block all other uses of this resource instance. Non-fungible resources need to be controlled on a system-wide basis. In most cases, non-fungible resources are assigned by Fence and reserved by applications, even when the application is not executing. Thus, even if a webserver application is not running, it may reserve TCP port 8080 to prevent other applications from using the port (closing a common security hole [83]).

Non-fungible resource access is summarized in lines 4, 5 and 6 of Table 1. A non-fungible resource that is not currently allocated can be accessed (after allocating it to the caller). Otherwise, access to a busy resource will either block or raise an error. Similar to fungible resources, non-fungible resources will either replenish over time, or are released explicitly by the caller.

3.2 Renewable and non-renewable

Some low-level resources, such as CPU cycles and network transmit rate, have an innate time dimension wherein the operating system continually schedules the use of the device or resource. These low-level resources are *renewable resources* because they automatically replenish as time elapses. Put another way, one cannot conserve the resource by not using it. If the CPU remains idle for a quantum, this does not mean there is an extra quantum of CPU available later. The control mechanism for a renewable resource is to limit the rate of usage, or utilization. Utilization is controlled over one or more periods, where the application's use of the resource is first measured and then paused for as long as required to bound it below a threshold, on average.

For example, to bound data plan costs, one may set a per-month limit for an application. Preventing the application's data usage from impacting the responsiveness of other network applications may also require a per-second limit. If an application attempts to consume a renewable resource at a faster rate than is allowed, the request is not issued until sufficient time has passed. (An extreme version of this involves batching requests together [17, 58, 76].)

Lines 1 and 4 in Table 1 show that access is granted when there is sufficient quota or unallocated resources. When a renewable resource is oversubscribed or busy (as shown in lines 2 and 5), it will block access until the resource is replenished, which will happen when the caller refrains from using the resource for an interval.

Non-renewable resources like memory, file descriptors, or persistent storage space are acquired by an application and (ignoring memory paging) are not time-sliced out or shared by other applications. As a result, granting access to a non-renewable resource is a conceptually permanent allocation from the application's perspective. For resources other than persistent storage space, the allocation usually coincides with the lifespan of the application instance, although the application may often choose to voluntarily relinquish the resource at any time. Short of forcibly stopping the application instance, there is little remedy for reclaiming most non-renewable resources once they are allocated.

To monitor and control the usage of non-renewable resources, it suffices to keep a table of resource assignments that track and update requests and releases. Once resource caps are set, an application can never request more non-renewable resources than the cap. A request can be met if the requested quantity is no greater than the cap on a fungible resource (see line 1 of Table 1), or, for a non-fungible resource, if the resource is not already allocated (line 4). Trying to exceed the cap will result in an error signal (line 3), as will trying to access an already allocated resource (line 6).

3.3 Enforcing Resource Controls

In the preceding discussion it was assumed that resource consumption can be measured and controlled in some way. While the restrictions Fence intends to place upon resources are fully specified by their fungibility and renewability, the method of enforcing resource controls is not, and will potentially vary across implementations of Fence, on different platforms. We describe two specific types of enforcement that allow Fence to control resources: (1) call interposition and (2) polling and signaling (indicated in Figure 1).

Call interposition. Interposition allows Fence to be called every time a process consumes a resource. However, there are different strategies that one may choose to control resources. For example, consider the case of restricting network I/O rate (a renewable resource). Suppose that a program requests to send 1MB of data over a connected TCP socket. The fundamental question is when Fence's control of this resource should be enforced.

Pre-call strategy. If an application is only charged for sent data before the call executes, then often times it will be overcharged. Given our example of a 1MB send, the send buffer may not be large enough to accept delivery of the entire amount and so much less may be sent. In such

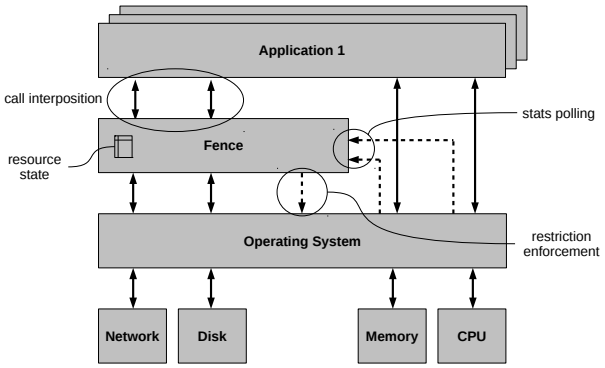


Figure 1: Fence's resource control via call interposition and polling/signaling.

a scenario, Fence would block the program for a longer time than is actually needed.

Post-call strategy. If the application is only charged after the call executes, however, it is possible to start threads that make huge requests to transmit information. This could allow the user to monopolize resources in the short term and to consume a large amount of resources.

Pre-and-post-call strategy. To better meter renewable resources, one can decouple the potential delay of access (done to adjust utilization based on past usage), from accounting, which is best performed after access. With this method, the pre-call portion of control will block until the use from prior calls has replenished; after that, the function is executed and the post-call portion charges the consumed amount.

Micro-call strategy. Finally, another option is to break a large call up into smaller calls and then allow Fence to handle them separately. However, this has the drawback of changing the original call semantics in some cases. For example, suppose that an application is only permitted to send 20kB of data per second. However, the caller wants to send a UDP datagram of 60kB. Instead of sending one 60kB datagram, the application could send three separate 20kB datagrams. Since each datagram is a "message" and the loss / boundaries between messages are meaningful (unlike the stream abstraction of TCP), sending three 20kB datagrams has a different meaning than does sending one 60kB datagram.

In practice, Fence is largely used with the *pre-and-post-call* strategy for renewable resources. For non-renewable resources, the *pre-call* strategy is predominant, because it will block access to an unauthorized resource before it occurs. Other strategies can be used, depending on how Fence is integrated into a platform.

Polling and signaling. For resources on which it cannot interpose, Fence uses polling, coupled with some form of limit enforcement, such as signaling, as a complementary mechanism to control resource usage. For example, for CPU scheduling, Fence does not modify the OS; rather, the OS scheduler manages process scheduling directly. Fence needs to poll to understand how often a process has been scheduled and must signal the scheduler to stop or start executing the application. This has a number of implications [80, §5.5.3], [78, §18.3] that call interposition does not present.

Atomicity. While trivially implementable for call interposition (e.g., by guarding calls with semaphores), resource access is not guaranteed to be atomic. This raises potential Time-Of-Check To Time-Of-Use (TOCTTOU) issues between threads of the same application as well as other applications wanting to access the resource.

Interference due to load. Fence's polling and signaling impose a certain amount of overhead that depends on the rate of checking and control. When the machine is under load, Fence might not be able to keep up with its planned check and control schedule. As a result, a process under its control can consume a resource for longer than the time scheduled by Fence. Thus, in the worst case, this causes overconsumption and reinforces the overload condition.

Rate of checking and control. The fidelity and overhead of resource control depend on the rate at which control is enforced. The minimum rate (i.e. the longest interval between interruptions) is given by the minimal fidelity desired. If the rate is too low, a process might overspend a resource between checks. The maximum rate is bounded by the maximum acceptable overhead – each check and interruption causes additional cost – and the granularity of checking and control (which defines the minimum possible length of interruption).

Polling and signaling granularity. Granularity is the smallest unit of measurement and control of a resource, e.g., the resolution of the system clock, or the unit job that control functions can handle (process, thread). If the granularity is too coarse, Fence might misapprehend the resource consumption of a process under its control during polling or enforce undue restrictions on the process.

3.4 Choosing Resource Control Settings

A question that follows from the previously discussed framework is how to choose which resources to allocate to a process. This can be viewed along two axes. First, policies can set direct per-resource usage limits or quotas, or indirectly establish limits based on the effect of resource usage on device availability. Second, such limits may be static, set manually and *a priori*, or dynamic, where policies continually adjust limits to achieve availability goals[2].

For most non-fungible resources (like TCP / UDP port

[2]We also envision that different stakeholders desire different policies: For example, a programmer could include Fence in their application in order to tame it; a sysadmin may specify policies based on user groups, time of day etc.; the end user sets values with respect to their current workload and usage requirements.

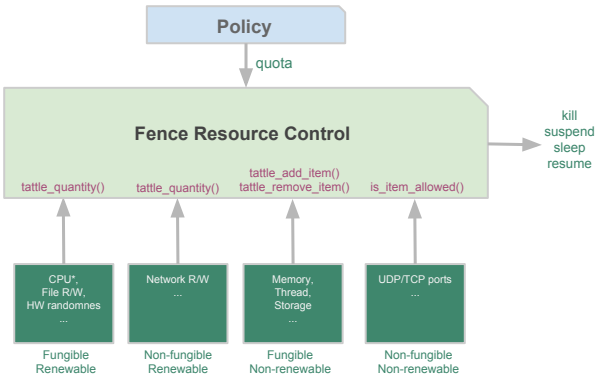


Figure 2: Fence's uniform control APIs for resources based on fungibility and renewability

```
def sendmessage(destip,destport,msg,localip,localport):
  ...
  # check that we are permitted to use this port...
  if not fence.is_item_allowed('messport',localport):
    raise ResourceAccessDenied("...")

  # get the OS's UDP socket
  sock = _get_udp_socket(localip, localport)

  # Register this socket descriptor with fence
  fence.tattle_add_item("outsockets", id(sock))

  # Send this UDP datagram
  bytessent = sock.sendto(msg, (destip, destport))

  # Account for the network bandwidth utilized
  if _is_loopback_ipaddr(destip):
    fence.tattle_quantity('loopsend', bytessent + 64)
  else:
    fence.tattle_quantity('netsend', bytessent + 64)
  ...
```

Figure 3: Fence additions to the Seattle sandbox's sendmessage call. Added lines of code are in bold text.

numbers) or for items that are hard to quantify (like privacy, cf. §6), choosing appropriate resource settings is typically done manually during packaging or installation.

However, the relationship between a limit (say, 30% of CPU or disk bandwidth), and user-observable properties (such as response time, battery life, or heat) is non-obvious and dependent on a number of factors. Accurate resource modeling is very difficult and in general requires advanced techniques [35, 45, 68, 75, 89], but determining a workload that approximates the maximum negative impact is an easier task.

Fence provides the capability to automatically choose resource limits, in particular, for fungible resources. When Fence is installed, it can benchmark the platform and check the usage impact on different resources, such as performance degradation, battery usage, or temperature. By understanding the impact of individual resources, one can choose resource settings that provide device availability in the face of contention. In §5.5 we show a policy that sets limits both indirectly and dynamically, based on a target battery lifetime.

4 Implementation

Fence's fully functional open source implementation that runs across various hardware platforms and OSses includes the features and functionality described in the previous sections. The core implementation is 790 lines of Python code (as counted by `sloccount`). There are two major portions of the Fence code: the uniform resource control code (140 LOC) and the operating system specific code (650 LOC).

4.1 Uniform Resource Control

Because the uniform resource control code only differs in its characteristics of fungibility and renewability, it fits within 140 LOC. Fence is informed about resource consumption by performing a set of four calls as is shown in Figure 2.

The specific call made depends on the type of resource being consumed. For example, the Seattle sandbox (described in more detail in § 6) required an additional 79 lines of code to support Fence, 68 of which were direct calls to Fence Figure 3 shows some of the code changes made to `sendmessage`, Seattle's API call for sending UDP datagrams. The first call, `is_item_allowed()`, checks whether the UDP port (a non-fungible, non-renewable resource) can be consumed. The `tattle_add_item()` call is used to charge for entries in the socket descriptor table to prevent the kernel from being overloaded with active sockets. The `tattle_quantity()` call charges for the consumed bandwidth (a renewable resource), depending on the destination interface.

In addition, there is an API that can be used to set high level policy. This is done by setting low-level resource quotas for the different resources that Fence manages. For example, if energy is the primary concern, resource quotas can be set to restrict the maximum expected energy consumption over a polling period. Actual energy consumed can be measured and the resource quota values updated appropriately.

4.2 Operating System Specific Code

The bulk of the Fence code (650 LOC) is operating system specific and involves supporting polling or enforcement across various platforms, including Windows XP and later, Mac OS X, Linux, BSD variants, One Laptop Per Child, Nokia devices, iPhones / iPods / iPads, and Android phones and tablets.

While all these platforms have the necessary low-level functionality, a convergence layer is still required because polling and enforcement semantics differ from platform to platform. For example, resource statistics are gathered in a fundamentally different way across many types of devices. However, the operating systems for many platforms

are derived from similar sources and thus share some subset of resource control functionality. For example, the Android port of Fence is based on the convergence layer of the Linux implementation of Fence and reuses almost all of its code.

4.3 Operating System Hooks Utilized

At the lowest level, Fence polls OS specific hooks for performance profiling to obtain statistics about the resource consumption of an application. To gain control over scheduling, Fence uses the job control interface on most OSes (SIGSTOP and SIGCONT), or the `SuspendThread` / `ResumeThread` interface on Windows. This informs the scheduler when to suspend or continue a process, in order to impose a different scheduling policy than the underlying OS scheduler. This can give a program less CPU time than it would ordinarily have to limit its performance impact, rate of battery drain, or heat.

To control resources other than the CPU, Fence constructs a uniform interface to low-level functionality that operates in an OS-specific manner. Consider memory as an example: On Linux and Android, Fence can use `proc` to read the memory consumption (Resident Set Size) for the process. On Mac OS X, similar actions are performed by interfacing with `libproc`. On Windows, calls to `kernel32` and `psapi`, the process status API, reveal the required information. Inside of its resource measurement and control routines, Fence can then use a single high-level call to gather the memory usage of a process, no matter what underlying OS or platform is used.

5 Evaluation

We evaluated Fence's software artifact and its deployment to investigate the following questions.

In situations with resource contention, how effectively do uniform resource control and legacy ad-hoc techniques provide device availability? (§5.2)

How well does uniform resource control function across diverse platforms? (§5.3)

How much overhead is incurred when employing uniform resource control during normal operation? (§5.4)

Can realistic, high level policies be expressed with Fence? (§5.5)

How diverse are the resources that can be metered by uniform resource control? (§6)

How time consuming and challenging is it to add a new resource type to Fence? (§6)

5.1 Experiment Methodology

To understand the tradeoffs between customized solutions and uniform resource control, we compared Fence to well-known, deployed tools found on common OSes. These included `nice` (which sets the scheduling priority of a process), `ionice` (similarly for I/O priority), `ulimit` (which imposes hard limits on the overall consumption of resources like file sizes, overall CPU time, and stack size), as well as a combination of these tools. We also included `cpufreq-set`, which changes the CPU frequency within the device and slows down all processing.

To create a model hog application that stresses resources, we created a series of processes that were intended to consume the entirety of a specific type of resource. For example, a CPU hog will simply go into an infinite loop, while a memory hog will acquire as much memory as possible and constantly touch different parts of it. We also created an 'everything hog' process that would use all of the memory, CPU, network bandwidth, and storage I/O it was allowed to consume.

For those experiments that looked at power consumption and temperature, we used the devices' built-in ACPI interfaces. Measurements were taken after a machine had been in a steady load state for ten minutes, to account for heating and cooling effects between load changes.

5.2 Availability of Fence vs Legacy Tools

5.2.1 Performance Degradation

Setup. To evaluate all of the tested tools' abilities to contain a hog process, we ran an experiment where the 'everything hog' interfered with VLC playback of a 1080p H.264 video [24] stored on disk. The results presented here were generated on a Dell Inspiron 630m laptop running Ubuntu 10.04; however, our results were similar across different device types and operating systems. We set all of the tools, including Fence, to their most restrictive settings[3] in order to contain the hog processes. For `nice` and `ionice`, VLC was additionally set to have the highest possible priority.

Result. Figure 4 shows the results[4] in terms of the proportion of frames decoded by the player when competing with hog processes. Existing tools performed very poorly when competing with the 'everything hog' – using `nice` (19.2%), `ionice` (17.7%), or `ulimit` (16.4%) showed not much more impact than not using them (16.7%). Even using a combination of all these tools showed little effect (21.8%). Setting the CPU frequency lower (7.9%) slowed down the entire system, including the video player, causing even more dropped frames. Because Fence limits all types of resources, even the everything hog has very limited impact (Fence delivers 99.8% of the frames).

One surprising finding was that `nice` was highly ineffective in protecting the video player against a CPU hog.

[3] I.e. `nice` level +20, `ionice` class *idle*, lowest CPU frequency setting, and `ulimit` memory to 10 MB; for Fence, 1% of the CPU, 10kBps disk rate, 10 MB of memory

[4] An anonymized video that shows the actual playback quality in this experiment is available [8].

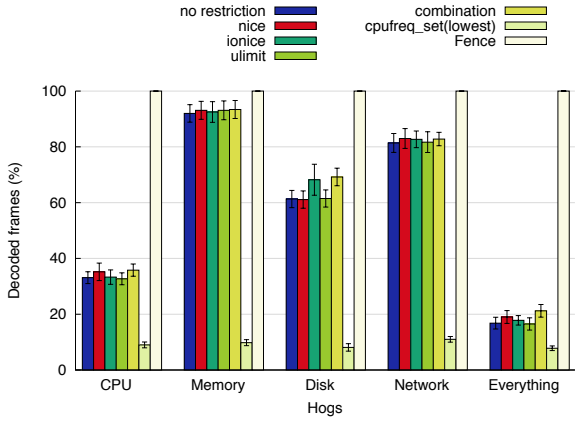


Figure 4: Proportion of decoded frames during video playback when competing with different resource hogs (averages of ten runs $\pm\sigma$, larger is better).

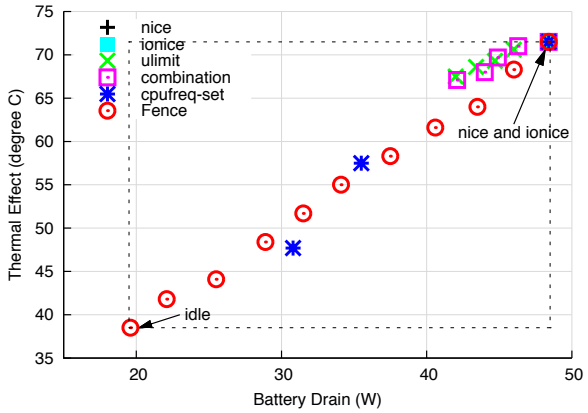


Figure 5: Battery drain and heat control ranges for various tools. The dashed rectangle bounds the minimum observed (when idle) and maximum observed battery drain / heat of the device. Fence effectively sweeps the full range.

It provides little benefit for a high-priority program that exhibits bursty behavior with implicit deadlines for work activities. However, tasks that constantly consume CPU, such as benchmarks, benefited substantially from `nice`.

Evaluations of the 'everything hog' showed that heavy use of one resource tended to cause a performance problem for other resources. Thus, resource conflicts seemed to compound, yet defenses were about the same as the strongest individual defense. This implies that uniformly strong defenses are essential to preventing an application from degrading performance.

5.2.2 Control Of Power And Heat

Setup. We next compared the effectiveness of popular tools in controlling the rate of power consumption and thermal effect (heat) caused by an application. To make this comparison, we ran an 'everything hog' process on the laptop for ten minutes and examined the ACPI battery and temperature sensors. We then applied Fence, `ulimit`, `cpufreq-set`, `nice`, and `ionice` on the hog process, for ten minutes each, and read the battery and temperature sensors again. To understand the impact of the tool, we varied the hog's priority or resource settings by choosing a variety of settings, including the lowest to highest settings permitted by the tool. Intermediate settings were used to further understand the tool's effective range of control. An ideal tool would allow the temperature and battery drain of the 'everything hog' to be precisely controlled in a range from idle (no impact from the hog) to full system utilization (full performance of the hog).

Results. Figure 5 shows the ability of tools to control the rate of battery consumption and heat. Existing tools, such as `ulimit` and `nice`, showed no measurable impact on a hog's ability to consume power (48.4W) or to raise the temperature of a device (71.5°C). This is because when there are available resources, any program (no matter how low its priority) can exhaust the battery and thermal capabilities of a device. In comparison, `ulimit` can at least slightly improve battery life (41.9W) and thermal effect (67.4°C) because it will reduce memory use. Using all these tools simultaneously on a hog produced a similar effect to `ulimit` (42.2W, 67.1°C). Thus `ulimit`, `nice`, and `ionice` are not effective in controlling battery drain or heat.

In our experiment, `cpufreq-set` was much more effective than were other off-the-shelf tools (30.8W, 47.7°C). However, as described in the previous section, `cpufreq-set` negatively impacted all applications. It also failed to control access to resources other than the CPU, which resulted in high residual battery drain and temperature (e.g., from wireless network adapters).

Fence was much more effective in controlling battery consumption; it brought these values down to within .2% of idle. The uniform resource control impacted all power consuming resources, which led to effective control of battery drain and heat.

5.3 Effectiveness on Diverse Platforms

Setup. We ran a series of benchmarks to investigate whether uniform resource control is effective on diverse platforms. We tested Fence on smartphones (Samsung Galaxy Y, Nokia N800), tablets (Samsung Galaxy Tab, Apple iPad), laptops (MacBook, Inspiron, Thinkpad), desktop PCs (Alienware, Ideacenter), and also the commercial Amazon EC2 cloud computing platform. Operating systems tested included Windows 7, Ubuntu 10.04 to 12.04, Mac OS X 10.8.4, Nokia's OS2008, Android 2.3.5 and 4.0.3, and a jailbroken iOS 5.0.1

We ran five benchmarks for the Seattle testbed, including an HTTP server serving a large file, an Amazon S3-like block storage service, an HTTP server benchmark with small files / directory entries, a UDP P2P messaging

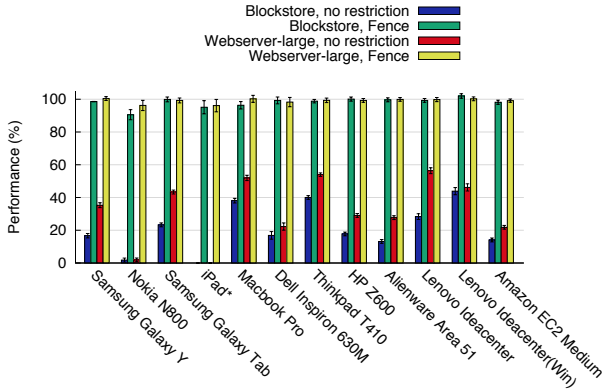


Figure 6: Benchmark performance across devices and OSes in the face of a hog (average of eight runs $\pm\sigma$, larger is better). *The iPad crashed when run with an unrestricted 'everything hog'.

program, and the Richards benchmark [19], set up against an 'everything hog'.

Note that the last four benchmarks were qualitatively and quantitatively similar, so we only present the results from the S3 blockstore. Figure 6 shows the results of these benchmarks with both an 'everything hog' with no restrictions and an 'everything hog' under Fence's control. The values are all normalized by dividing by the benchmark time on an idle device.

Results. The benchmark performance when the hog was unrestricted ranged from about 60% (Lenovo IdeaCentre running Ubuntu 11.04) down to 2% (Nokia N800). Note that performance results for the unrestricted hog on a jailbroken iPad are not represented because parts of the system crashed when we instantiated an unrestricted hog process.

If the hog process is bound by Fence to consume at most 1% of a device's resources, a typical VM size in the Seattle testbed [67], the hog's impact on a benchmark is minimal. If the hog is restricted to 1%, then the benchmark should run with about 99% of the original performance. The lowest value (90%) was recorded on a Nokia N800 that was purchased in 2007. On very low power devices such as this, Fence's overhead for polling and signaling is significant. However on other platforms, the benchmark's execution time was within 98% of the original performance. Our tests show that uniform resource control provides strong device availability with low overhead across a diverse array of modern platforms.

5.4 Overhead of Fence

Both interposition and polling can incur overhead on applications. Although the exact overhead will depend on where and how uniform resource control is implemented, it is important to have a rough understanding of the cost. We measured the overhead from both pre-post call interposition (§3.3) and polling / signaling using

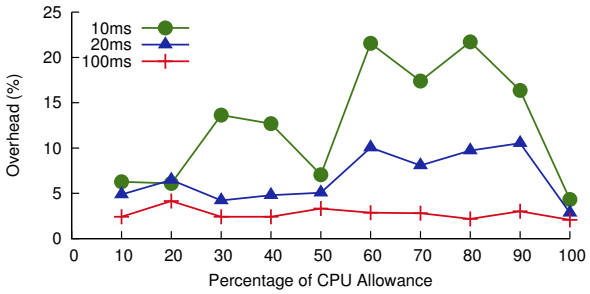


Figure 7: Overhead of CPU limiting which Fence inflicts on a CPU-intensive benchmark, at different polling rates.

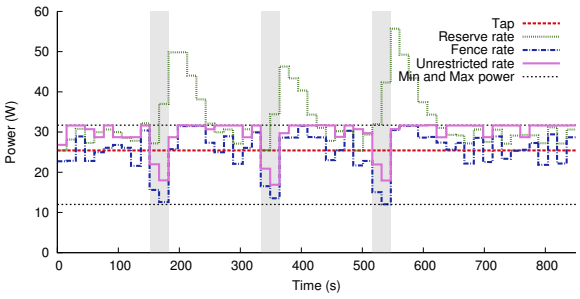


Figure 8: Fence restricts a benchmark's power draw using a simple token-bucket high-level policy [75]. *Tap* is the long-term power target; *reserve rate* the maximum currently allowed power. Benchmark idles in shaded regions.

Pulse-Frequency Modulation [37]. Figure 7 shows the relative overhead incurred by the CPU polling mechanism[5] used by Fence when compared with an unrestricted benchmark on a Lenovo IdeaCentre K330 desktop PC running Ubuntu 11.04. Results for other operating systems and platforms were qualitatively similar. Fence's current implementation uses a 100ms polling interval for Linux on a desktop machine, which causes an overhead of less than 4% across the range of CPU allowances. If the polling interval was reduced to 20ms, control would be finer-grained, but the overhead would exceed 10% for allowance values above 50%. (Only at 100% allowance, *i.e.*, no CPU restrictions, does the overhead drop again. In this case, Fence never detects that the CPU allowance was exceeded; thus, only polling causes overhead and no signaling overhead is incurred.) Results for many shorter polling rates (e.g., 1ms) are omitted because they do not fit on the graph. Using a 100ms polling interval for Fence results in low overhead across a wide variety of resource restriction settings.

5.5 Expressing a High-Level Policy

We now explore how a high-level policy can be expressed with Fence. To demonstrate this we implemented a simple policy from Cinder [75, Fig.1] to limit the long-term power draw of an application. Cinder uses *reserves* that

[5]Resource types other than CPU have an impact on the order of 100 *ns* to 8 μs per interposition.

store energy and *taps* that transfer energy between reserves per unit time to control energy use. In this scenario, a fixed-rate tap replenishes a reserve at fixed intervals, and the application can only draw energy from the reserve. (This is analogous to a token bucket where the tap is the filling rate of the bucket and the reserve is the number of tokens in the bucket.) The tap limits the long-term power draw, which is useful to provide battery lifetime guarantees, while the reserve allows for bursts. We implemented the policy as a control loop that reads the battery information from ACPI to determine the power draw for the previous interval, computes the necessary limits for processes, and sets the quota for Fence to enforce.

We examined the power draw on a Dell M1210 laptop. The long-term power draw goal from the battery was set to 25.4W, corresponding to about 69% between idle (12.0W) and maximum (31.5W) power draw on this machine. The "battery tap" replenished a reserve from which the application drew energy. To read the battery information from the system, we used the power supply API that updated approximately every 15 seconds. The theoretical maximum power draw in the reserve for one period was tracked at the same temporal resolution.

Fence operates at the user level and as such does not have low-level information about the energy consumption of individual hardware components. To enforce energy consumption restrictions, we requested that Fence set the quota on every renewable resource based on the amount of energy in the reserve. For example, if the reserve allowed for 28W for the next interval, then all renewable resources were set to 82% of their maximum value, as $28\text{W} = 12.0\text{W} + 0.82\% \cdot (31.5\text{W} - 12.0\text{W})$. While this policy has many simplifying assumptions, we find that in practice it works.

We ran a Richards benchmark [19] in the following fashion. It ran for 10 periods (where a period is the amount of time between power supply readings), and then slept for 2 periods. After doing this three times, it ran continually for 20 periods. The benchmark was run according to this schedule, both with and without Fence.

Results of this benchmark are shown in Figure 8. The shaded areas indicate when the benchmark was sleeping. The tap rate, representing the long-term power draw goal, was 25.4W for each period. The unrestricted rate shows the benchmark's power draw in absence of any power restrictions: For the majority of time that the benchmark test was active, it caused the battery to drain at approximately the maximum rate.

The Fence rate shows the benchmark's power draw when run under Fence. Note that unlike the unrestricted benchmark, the behavior of this line is limited by *two* power rates: The maximum power draw that the system allows, and also the amount in reserve in each period. When the reserve is large (for example, right after a sleep period), Fence behaves similarly to the unrestricted rate, and uses energy at approximately the maximum rate. When the reserve is small, Fence restricts power use to approximately the rate of the tap. Cinder's policy uses Fence to enforce that the overall power use stays within budget, while allowing the application to have flexibility in when it consumes its energy budget.

Fence's power restrictions are inaccurate, due in part to our implementation working at user space. As such we do not directly account for complex issues of power use in the underlying platform (e.g. tail power consumption [34]). However, Fence will read the new battery level during the next period and will drain the reserve based upon what was actually consumed. Fence's adjustment in subsequent periods mitigates the effect of this inaccuracy.

Fence made implementing Cinder's policy very straightforward. The implementation is 150 lines of code and did not require any detailed knowledge of the underlying resource types or control mechanisms. The implementation simply reads the battery level and adjusts the values in the resource table based upon the reserve and tap settings. This demonstrates that uniform resource control may make policy implementation easier, which we hope will lead to more application developers and system designers using such mechanisms.

6 Practical Fence Deployments

Seattle's Use of Fence. Fence is deployed as a part of the Seattle testbed [21]. Seattle runs on laptops, tablets, and smartphones, with more than twenty thousand installs distributed around the world [88]. The Seattle testbed is used to measure end user connectivity, to build peer-to-peer services, and as a platform for apps that measure and avoid Internet censorship [16, 44, 46, 73, 81]. Seattle is also widely used in computer science education where it has been used in more than fifty classes at more than a dozen universities [39, 41, 67].

Each device running Seattle uses Fence to allocate a fixed percentage (usually 10%) of the device's CPU, memory, disk, and other resources to one or more VMs. When a Seattle sandbox [40] is started, it reads a text file that lists the resources allocated to the program. Each line contains a resource type and quantity (for fungible resources) or the name of the resource (for non-fungible resources). For example, `resource memory 10000000` sets the memory cap to 10 million bytes, and `resource udpport 12345` allows the program to send and receive UDP traffic on port 12345.

Seattle's categorization of resources and enforcement mechanisms are shown in Table 2. In the Seattle platform's sandbox, the calls to network and disk devices are routed through Fence, whereas usage statistics on memory and CPU are polled from the operating system. Unfortunately, there is not a clean, cross-platform way

Resource	Fungible	Renewable	Seattle Control	Lind Control
CPU	Yes	Yes	Polling	Polling
Threads	Yes	No	Interposition	Interposition
Memory	Yes	No	Polling	Interposition
Storage space	Yes	No	Polling	Interposition
UDP / TCP ports	No	No	Interposition	Interposition
Open sockets	Yes	No	Interposition	Interposition
Open files	Yes	No	Interposition	Interposition
File R/W	Yes	Yes	Interposition	Interposition
Network R/W	No	Yes	Interposition	Interposition
HW randomness	Yes	Yes	Interposition	Interposition

Table 2: The user-space Fence implementation's resource categorization used in Seattle [40] and Lind [65].

for Fence to reclaim memory used by an application. As such, Seattle enforces a hard maximum allowed memory limit for an application, so a process trying to exceed the limit will be forcefully killed.

Lind's Use of Fence. In addition to being used in Seattle's sandbox, Fence is also used in a Google Native Client sandbox called Lind [65]. Since this sandbox provides a different abstraction (a POSIX system call), some of the low-level resource characteristics (and thus, means of controlling consumption) vary from Seattle's deployment. This sandbox has a hook that allows Fence to interpose on memory requests and avoid polling. (This also allows Fence to control native programs that cannot be executed in the Seattle sandbox.) The rightmost column of Table 2 overviews Lind's resource categorization.

The Sensibility Testbed's Use of Fence. Fence is not limited to controlling traditional computational resources – it is currently being integrated by the Sensibility Testbed [22] developers. Sensibility Testbed consists of smartphones and tablets where researchers get access to dozens of diverse sensors including WiFi network information, accelerometer readings, battery level, device ID, GPS, and audio. Sensibility Testbed uses the same sandbox as the Seattle testbed. However, in addition to limiting a program's access to resources for performance reasons, Sensibility Testbed allows users who pass an IRB review to get access to devices' sensors at a rate meant to preserve user privacy (e.g. by limiting the rate or accelerometer queries to prevent sniffing keystrokes [70]). The demonstrated ability of Fence to provide privacy guarantees across "sensor resources" validates the generality of uniform resource control as a technique.

Experiences / Limitations. Our experience is that Fence's effectiveness depends on the amount of resource information available to the developer. E.g., it is relatively straightforward for a sandbox operating in user space to limit disk I/O when interposing on a `read` or `write` call on a file descriptor for a regular file – the number of disk blocks accessed is fairly easy to predict. However, it is very hard to provide performance isolation for a call like `mount`: It may perform a substantial number of disk I/O operations in the kernel, which are not predictable by a sandbox in user space. (For similar reasons, a hardware resource like L2 cache may be difficult to meter using software in an OS kernel.)

The time it takes a developer to understand the resources consumed by a call depends a lot on the implementer. In our experience, adding calls into Fence in the appropriate parts of the code only takes a few minutes per API call. In fact, outside groups have used Fence to provide resource controls on many platforms (Android, iOS, Nokia, Raspberry PI, and OpenWrt) each necessitating only a few days' worth of effort. The bulk of the effort lies in understanding what resources a call will consume.

7 Related Work

Our work follows a substantial amount of prior work that has recognized the need to prevent performance degradation, enhance battery control, and manage heat. Fence is unique in that it works across diverse platforms and presents a portable, user-space solution that unifies resource control across resource types.

Deployed Solutions For Improved Availability. The need for improved device availability has produced strategies for preventing performance degradation, with different levels of required privileges, including per-application, per-user, OS-wide, and hypervisor-based approaches. For example, modern web browsers monitor the run time of their JavaScript engine to detect "runaway" scripts, i.e., programs that take excessive time to execute and allow the user to stop the script. A malicious script can fool the timer however, by partitioning its workload, or by using Web Workers [69]. The runaway timer also ignores other resources that the JavaScript program may take, such as network and memory. Another application-level example, the Java Virtual Machine [12], supports setting a limit on the amount of memory that can be allocated by a process. However, much like Lua [56], Flash [1], and other programming language VMs, the Java Virtual Machine (JVM) does not support limiting the rate of some fungible/renewable resources, such as CPU — the primary cause of energy drain and heat on many devices.

Operating system virtual machine monitors control many different resource types, depending on the implementation [13, 26, 28, 29, 36, 72]. However, the resource controls are ad hoc and specific to the type of resource. Bare-metal hypervisors [13, 36] require kernel changes, whereas hosted hypervisors [26, 28] have substantial per-VM resource costs. As a result, none of these are practical to deploy on a per application basis, especially on devices like smartphones and tablets.

More recently, the `cgroups` [4] infrastructure in Linux addresses many of the issues discussed in this paper (CPU, memory use, and disk I/O). `cgroups`, however, is specific to newer versions of Linux. Due to its location in the kernel, it has better resource granularity than Fence. However, `cgroups` focuses on point solutions for spe-

cific resources, rather than a more uniform and general solution. In contrast, Fence provides a user-space uniform resource control solution that works across a wide array of devices without kernel modifications.

Operating Systems Research. There are many clean slate OS approaches that would achieve the same improvements as Fence. Since many deployed OSes are ineffective at preventing performance degradation, Fence focuses on providing this property in user space. Several OS-level efforts aim at managing processes' low-level resources consumption in a system-wide manner to control battery life. ECOSystem [89], Odyssey [48], Cinder [75], and ErdOS [85] focus on extending battery life. Their reasoning about resources is fixated on the energy consumed by renewable resources. Scheduling decisions are made exclusively on this foundation. Fence also supports energy-aware resource limiting, but we measure resource consumption as "unit of resource," which enables interesting use cases, such as control strategies based on device responsiveness or service availability.

Research projects [42, 43, 54, 60, 61, 66, 87] try to manage thermal effects through temperature-aware priority-based scheduling and thread placement on the CPU. Our work demonstrates that priority-based scheduling is ineffective at upper-bounding a process.

The efforts presented above require deep changes to applications to support resource aware operation, are based on new kernels, or use forms of priority-based scheduling that do not succeed in limiting resource consumption. In contrast, Fence requires no changes to OSes or applications, and operates entirely in user space. Furthermore, Fence puts boundaries on any type of resource consumption and can affect more than battery drain. This is achieved by reasoning about, tallying, and controlling multiple different resources in a uniform way.

Controlling resource consumption is well researched in the real-time OS community [31, 64, 74]. Our research focuses on techniques to enhance general-purpose OSes with minimal disruption to existing systems.

There have been a substantial number of complimentary user-space techniques for improving security that involve system-call interposition [52, 53, 59], host intrusion detection [47, 50], and access control [55, 77, 86]. These mechanisms aim to permit or deny access to resources requested by applications based upon how they will impact the security of the system. Fence's goal is fundamentally different: It limits the rate of resources consumption so that the use of allowed resources does not impact the availability or correct operation of a device.

Idle Resource Consumption A variety of frameworks, such as Condor [63], SETI@Home [23], and Folding@Home [9] allow trusted developers to consume idle resources on a users device. These wait for the user's system to be idle and then run, so as to not interfere with performance. However, once they run, these programs may fully utilize the CPU, GPU, and similar resources on the device, often leading to significant power drain [2]. Fence may also operate in such an on-off manner, but it is flexible enough to allow more advanced policies.

One related effort to Fence in this domain was the construction of an idle resource consumption framework by Abe et al. [30]. This system runs in user space and leverages special functionality from OS-specific hooks in Solaris revolving around `dtrace` [38]. While this provides easy control of native code (which Fence lacks), equivalent techniques do not exist across platforms. As such, this will only work for a few environments, such as BSD, that support similar hooks. As a result, Abe's work cannot be deployed on many systems. (For example, Windows lacks a non-bypassable method for interposing on an untrusted application's operating system calls.) Additionally, as this work seeks to enable background execution only when foreground execution is idle, many of the detection and scheduling results from this work do not apply to our domain.

Distributed Systems Research Controlling resource utilization is also an important problem in distributed contexts [32, 33, 49, 51, 57, 62, 79, 82, 84]. Significant prior work has focused on efficiently allocating available resources between multiple parties. While managing distributed resources is orthogonal to our goals, Fence embraces richer semantics to reason about resource consumption, rather than utilization; we believe that our approach towards uniform resource control would apply well as a heterogeneity-masking technique in distributed contexts.

8 Conclusion

This paper introduces uniform resource control by classifying resources along the dimensions of renewability and fungibility. Our system, Fence, demonstrates that uniform resource control provides flexibility by controlling multiple heterogeneous resources across almost a dozen diverse operating systems. Furthermore, we demonstrate that this technique is particularly adept at addressing issues of performance degradation, heat, and battery drain that many users face today.

In addition to the experimental validation presented in this paper, Fence has been deployed and adopted to provide resource containment of untrusted user code in several testbeds. As a result, tens of thousands of smartphones, tablets, and desktop OSes around the world rely on Fence to prevent device degradation. Beyond our deployment, we believe Fence's abstractions and mechanisms could be used to provide better resource control for sandboxes, web browsers, virtual machine monitors, and operating systems.

References

[1] Adobe flash player. http://www.adobe.com/software/flash/about.

[2] BOINC wiki: Heat and energy considerations. http://boinc.berkeley.edu/wiki/Heat_and_energy_considerations.

[3] Box Sync ruins my rMBP. https://support.box.com/entries/21766312-Box-Sync-ruins-my-rMBP.

[4] cgroups. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[5] Crash Internet Explorer 9 in one line of Javascript! http://www.roslindesign.com/2011/04/08/crash-internet-explorer-9-in-one-line-of-javascript.

[6] Death to Javascript: CNN Edition. http://www.goodbyemicrosoft.net/news.php?item.708.4.

[7] Dropbox using 100% of each core when I turn my machine on. https://forums.dropbox.com/topic.php?id=62054.

[8] Fence vs nice, ionice, ulimit, cpufreq-set demo video. https://www.youtube.com/user/fencedemo.

[9] Folding@Home. http://folding.stanford.edu/.

[10] installd took 130% of CPU and sent temperatures through the roof. Why? https://discussions.apple.com/thread/3738340.

[11] ionice. http://linux.die.net/man/1/ionice.

[12] Java Virtual Machine. http://docs.oracle.com/javase/specs/jvms/se7/html/index.html.

[13] Kernel-based virtual machine. http://www.linux-kvm.org/page/Main_Page.

[14] Linux - File descriptors exhausted, how to recover. http://www.linuxquestions.org/questions/linux-newbie-8/linux-file-descriptors-exhausted-how-to-recover-4175417070/.

[15] nice. http://www.kernel.org/doc/man-pages/online/pages/man2/nice.2.html.

[16] Open3GMap. https://o3gm.cs.univie.ac.at/.

[17] Optimizing downloads for efficient network access. https://developer.android.com/training/efficient-downloads/efficient-network-access.html.

[18] Potentially Unwanted Miners – Toolbar Peddlers Use Your System To Make BTC. http://blog.malwarebytes.org/fraud-scam/2013/11/potentially-unwanted-miners-toolbar-peddlers-use-your-system-to-make-btc/.

[19] Richards benchmark. http://www.cl.cam.ac.uk/~mr10/Bench.html.

[20] Scan causes CPU to overheat. https://community.norton.com/t5/Norton-Internet-Security-Norton/Scan-causes-CPU-to-overheat/td-p/642743.

[21] Seattle web page. https://seattle.poly.edu/.

[22] Sensibility Testbed. https://sensibilitytestbed.com/.

[23] SETI@Home. http://setiathome.berkeley.edu/.

[24] Sintel trailer. http://www.sintel.org/download.

[25] ulimit. http://linux.die.net/man/1/ulimit.

[26] VirtualBox. https://www.virtualbox.org/wiki.

[27] VirusScan freezes up completely during full scan. https://community.mcafee.com/message/211811.

[28] VMware workstation. http://www.vmware.com.

[29] Windows Virtual PC. http://support.microsoft.com/kb/958559.

[30] Y. Abe, H. Yamada, and K. Kono. Enforcing appropriate process execution for exploiting idle resources from outside operating systems. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 27–40, 2008.

[31] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating multimedia applications in hard real-time systems. In *In Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[32] Y. Agarwal, S. Savage, and R. Gupta. Sleepserver: a software-only approach for reducing the energy consumption of pcs within enterprise environments. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.

[33] A. AuYoung, B. Chun, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Bellagio: An economic-based resource allocation system for planetlab. `http://www.sysnet.ucsd.edu/~aauyoung/bellagio/about.php`.

[34] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293. ACM, 2009.

[35] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[36] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *the nineteenth ACM symposium on Operating systems principles*, 2003.

[37] M. Broughton. Pulse-frequency modulation applied to the digital control of a thyristor. *IEEE Trans. Industrial Electronics and Control Instrumentation*, 24(2):173–177, May 1977.

[38] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, 2004.

[39] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: a platform for educational cloud computing. In *40th ACM technical symposium on Computer science education*, 2009.

[40] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *the 17th ACM conference on Computer and communications security*, 2010.

[41] J. Cappos and R. Weiss. Teaching the security mindset with reference monitors. In *45th ACM technical symposium on Computer science education*, 2014.

[42] J. Chen, C. Hung, and T. Kuo. On the minimization fo the instantaneous temperature for periodic real-time tasks. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 236–248. IEEE, 2007.

[43] J. Choi, C. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 international symposium on Low power electronics and design*, pages 213–218. ACM, 2007.

[44] L. Collares, C. Matthews, J. Cappos, Y. Coady, and R. McGeer. Et (smart) phone home! In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, pages 283–288, New York, NY, USA, 2011. ACM.

[45] S. S. Craciunas, C. M. Kirsch, and H. Röck. I/O resource management through system call scheduling. *ACM SIGOPS Operating Systems Review*, 42(5):44–54, 2008.

[46] J. Eisl, A. Rafetseder, and K. Tutschku. Service architectures for the future converged internet: Specific challenges and possible solutions for mobile broad-band traffic management. *Future Internet Services and Service Architectures*, 15:49, 2011.

[47] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 194–208, 2004.

[48] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. In *ACM Transactions on Computer Systems (TOCS)*, 2004.

[49] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking energy in networked embedded systems. In *the 8th USENIX conference on Operating systems design and implementation*, 2008.

[50] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128, 1996.

[51] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, 2003.

[52] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *IN NDSS*, 2003.

[53] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, 1996.

[54] J. Hasan, A. Jalote, T. N. Vijaykumar, and C. E. Brodley. Heat stroke: Power-density-based denial of service in SMT. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 166–177, Washington, DC, USA, 2005. IEEE Computer Society.

[55] T. L. Hinrichs, D. Martinoia, W. C. Garrison III, A. J. Lee, A. Panebianco, and L. Zuck. Application-sensitive access control evaluation using parameterized expressiveness. In *Computer Security Foundations Symposium, 2013. 26th IEEE*, 2013.

[56] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua - an extensible extension language. In *Software: Practice & Experience*, 1995.

[57] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing networked resources with brokered leases. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, 2006.

[58] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical report, Technical report HPL–CSP–91–7rev1, 1991.

[59] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[60] R. Jayaseelan and T. Mitra. Temperature aware scheduling for embedded processors. In *VLSI Design, 2009 22nd International Conference on*, pages 541–546. IEEE, 2009.

[61] A. Kumar, L. Shang, L. Peh, and N. Jha. Hybdtm: a coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd annual Design Automation Conference*, pages 548–553. ACM, 2006.

[62] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3), Aug. 2005.

[63] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, 1988.

[64] R. J. Masti, C. Marforio, A. Ranganathan, A. Francillon, and S. Capkun. Enabling trusted scheduling in embedded systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 61–70, New York, NY, USA, 2012. ACM.

[65] C. Matthews, J. Cappos, R. McGeer, S. Neville, and Y. Coady. Lind: Challenges turning virtual composition into reality. In *Freeco 2011 Onward! Workshop: Towards Free Composition of Software Modules*, 2011.

[66] A. Merkel and F. Bellosa. Task activity vectors: a new metric for temperature-aware scheduling. *ACM SIGOPS Operating Systems Review*, 42(4):1–12, 2008.

[67] Monzur Muhammad and Justin Cappos. Towards a Representive Testbed: Harnessing Volunteers for Networks Research. In *The First GENI Research and Educational Workshop*, GREE'12, 2012.

[68] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 67–72. IEEE, 2001.

[69] Nicholas C. Zakas. Responsive Interfaces. `http://de.slideshare.net/nzakas/responsive-interfaces`.

[70] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang. Accessory: Password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM.

[71] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[72] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 351–366, 2006.

[73] A. Rafetseder, F. Metzger, D. Stezenbach, and K. Tutschku. Exploring youtube's content distribution network through distributed application-layer measurements: a first view. In *Proceedings of the 2011 International Workshop on Modeling, Analysis, and Control of Complex Networks*, Cnet '11, pages 31–36. ITCP, 2011.

[74] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 150–164, 1998.

[75] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *EuroSys 2011*, 2011.

[76] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer Volume 27 Issue 3*, 1994.

[77] M. Sherr and M. Blaze. Application containers without virtual machines. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 39–42, New York, NY, USA, 2009. ACM.

[78] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[79] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. Technical report, Old Dominion University, Norfolk, VA, USA, 1996.

[80] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[81] K. Tutschku, A. Rafetseder, J. Eisl, and W. Wiedermann. Towards sustained multi media experience in the future mobile internet. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–6. IEEE, 2010.

[82] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol.*, 9(1), Feb. 2009.

[83] Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE. Accessed April 14, 2012. `http://msdn.microsoft.com/en-us/library/ms740621%28VS.85%29.aspx`.

[84] M. Valero, A. Bourgeois, and R. Beyah. Deep: A deployable energy efficient 802.15.4 mac protocol for sensor networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–6, 2010.

[85] N. Vallina-Rodriguez and J. Crowcroft. Erdos: achieving energy savings in mobile os. In *Proceedings of the sixth international workshop on MobiArch*, MobiArch '11, pages 37–42, New York, NY, USA, 2011. ACM.

[86] H. Vijayakumar, J. Schiffman, and T. Jaeger. Process firewalls: Protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 57–70, New York, NY, USA, 2013. ACM.

[87] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin. Dynamic thermal management through task scheduling. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 191–201. IEEE, 2008.

[88] Yanyan Zhuang and Albert Rafetseder and Justin Cappos. Experience with Seattle: A Community Platform for Research and Education. In *The Second GENI Research and Educational Workshop*, GREE'13, 2013.

[89] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *10th international conference on Architectural support for programming languages and operating systems*, 2002.

Request-Oriented Durable Write Caching for Application Performance

Sangwook Kim[†], Hwanju Kim[§,*], Sang-Hoon Kim[‡], Joonwon Lee[†], Jinkyu Jeong[†]
[†]*Sungkyunkwan University,* [§]*University of Cambridge,* [‡]*KAIST*
{*sw.kim, joonwon, jinkyu*}*@skku.edu, hwandori@gmail.com, sanghoon@calab.kaist.ac.kr*

Abstract

Non-volatile write cache (NVWC) can help to improve the performance of I/O-intensive tasks, especially write-dominated tasks. The benefit of NVWC, however, cannot be fully exploited if an admission policy blindly caches all writes without differentiating the criticality of each write in terms of application performance. We propose a request-oriented admission policy, which caches only writes awaited in the context of request execution. To accurately detect such writes, a critical process, which is involved in handling requests, is identified by application-level hints. Then, we devise criticality inheritance protocols in order to handle process and I/O dependencies to a critical process. The proposed scheme is implemented on the Linux kernel and is evaluated with PostgreSQL relational database and Redis NoSQL store. The evaluation results show that our scheme outperforms the policy that blindly caches all writes by up to 2.2× while reducing write traffic to NVWC by up to 87%.

1 Introduction

For decades, processor and memory technologies have been significantly improved in terms of performance whereas the performance of storage still lags far behind that of other components. To remedy this performance gap, modern operating systems (OSes) use main memory as a cache for underlying storage. With large main memory, this technique is effective for read-intensive tasks by hiding long latency of storage reads [63]. For write operations, however, caching is less effective because the volatility of main memory may lead to data loss in the event of power failure. As a consequence, write operations dominate the traffic to storage in production workloads operating with large main memory [13, 50, 69, 77].

Non-volatile write cache (NVWC) can help to improve the performance of I/O-intensive tasks, especially write-dominated tasks. For this reason, battery-backed DRAM (NV-DRAM) has been widely exploited as an NVWC device for file systems [11, 15, 41], transaction processing systems [27, 57, 74], and disk arrays [36, 37]. In addition, various caching solutions based on flash memory have been extensively studied to efficiently utilize fast random access of flash memory [10, 19, 46, 49, 65]. Storage-class memory (SCM), such as spin-transfer torque magneto-resistive memory (STT-MRAM) [17] and phase change memory (PCM) [67], is expected to be deployed as NVWC since it provides low latency comparable to DRAM and persistency without backup battery.

Blindly caching all writes, however, cannot fully utilize the benefit of NVWC for application performance due to the following reasons. Firstly, it can frequently stall writes in the performance-critical paths of an application due to the lack of free blocks in NVWC, especially for capacity-constrained devices, such as NV-DRAM and STT-MRAM. Secondly, it can cause severe congestion in OS- and device-level queues of NVWC, thereby delaying the processing of performance-critical writes. Finally, it would hurt the reliability and performance depending on the characteristics of the NVWC device used. For instance, caching non-performance-critical writes exacerbates wear-out of storage medium, such as flash memory [39, 40, 80] and PCM [17, 66], without any gain in application performance.

We propose a *request-oriented admission policy* that only allows *critical writes* (i.e., performance-critical writes) to be cached in NVWC. In particular, we define critical writes as the writes awaited in the context of *request execution* since the performance of processing an external *request*, like a key-value PUT/GET, determines the level of application performance. By using the proposed policy, a large amount of non-critical writes can be directly routed to backing storage bypassing NVWC because typical data-intensive applications, such as relational database management system (RDBMS) [38, 59] and NoSQL store [25, 31], delegate costly write I/Os

[*]Currently at EMC

to background processes while concurrently handling requests using other processes; we refer to any kind of execution context as process in this paper.

The key challenge of realizing the proposed policy is how to accurately identify all critical writes. Basically, synchronous writes requested by a *critical process*, which is involved to handle requests, are critical writes by definition. This simple identification, however, cannot detect process and I/O dependency-induced critical writes generated by complex synchronizations during runtime. Since synchronization is the technique frequently used to ensure correct execution among concurrent processes and I/Os, unresolved dependencies can significantly delay the progress of a critical process, thereby degrading application performance.

We devise *hint-based critical process identification* and *criticality inheritance protocols* for accurate detection of critical writes. Basically, the proposed scheme is guided by a hint on a critical process from an application. Based on the given hint, synchronous writes requested by a critical process are cached in NVWC. To handle process dependency, we inherit criticality to a non-critical process on which a critical process depends to make progress. To handle I/O dependency, we dynamically reissue an outstanding non-critical write with which a critical process synchronizes to NVWC without compromising the correctness. We also resolve cascading dependencies by tracking blocking objects recorded in the descriptors of processes who have dependencies to a critical process.

Our proposed scheme was implemented on the Linux kernel and FlashCache [2]. Based on the prototype implementation, we evaluated our scheme using PostgreSQL [5] and Redis [23] with a TPC-C [7] and YCSB [26] benchmark, respectively. The evaluation results have shown that the proposed scheme outperforms the policy that blindly caches all writes by 3–120% and 17–55% while reducing write traffic to NVWC by up to 72% and 87%, for PostgreSQL and Redis, respectively.

Our key contributions are the followings:

- We introduce a novel NVWC admission policy based on request-oriented write classification.
- We devise criticality inheritance protocols to handle complex dependencies generated during runtime.
- We prove the effectiveness of our scheme by conducting case studies on real-world applications.

The remainder of this paper is organized as follows: Section 2 describes the background and motivation behind this work. Section 3 and Section 4 detail the design of the proposed policy. Section 5 explains the prototype implementation, and Section 6 presents our application studies. Section 7 presents the evaluation results. Finally, Section 8 presents related work and Section 9 concludes our work and presents future direction.

2 Background and Motivation

2.1 Non-volatile Write Caches

Unlike conventional volatile caches, non-volatile write cache (NVWC) is mainly used to durably buffer write I/Os for improving write performance. Traditionally, NV-DRAM has been widely used as an NVWC device to enhance write performance by exploiting its low latency and persistency. Typical usages of NV-DRAM are writeback cache in RAID controllers [36, 37] and drop-in replacement for DDR3 DIMMs [8, 78]. An inherent limitation of NV-DRAM is small capacity due to high cost per capacity and battery scaling problem.

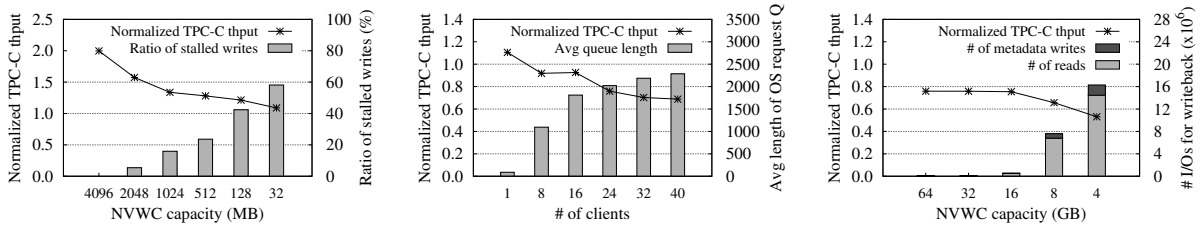
Recently, flash memory-based caching is gaining significant attention because it delivers much higher performance than traditional disks and much higher density than NV-DRAM. Thus, flash memory is widely adopted in many storage solutions, such as hybrid storage [19, 70] and client-side writeback caches in networked storage systems [10, 49, 65]. Despite of the benefits, flash memory also has caveats to be used as an NVWC device because it has limited write endurance [39, 40, 80] and garbage collection overheads [44, 45, 47, 62].

Emerging SCM, such as STT-MRAM [17] and PCM [67], is also a good candidate for an NVWC device since it provides low latency comparable to DRAM and persistency without backup power. Though STT-MRAM promises similar access latency to that of DRAM, its capacity is currently very limited due to technical limitation [1]. On the other hand, PCM has been regarded as more promising technology to be deployed at commercial scale than STT-MRAM [3]. PCM, however, has limited write endurance [52, 66], which necessitates careful management when it is used as an NVWC device.

2.2 Why Admission Policy Matters

A straightforward use of NVWC is to cache all writes and to writeback cached data to backing storage in a lazy manner. This simple admission policy is intended to provide low latency for all incoming writes as much as possible for improving *system performance* (e.g., IOPS). However, blindly caching all writes cannot fully utilize the benefit of NVWC in terms of *application performance* (e.g., transactions/sec) for the following reasons.

Firstly, caching all writes can frequently stall writes that are in the critical paths of an application due to the lack of free blocks in NVWC. This is because the speed of making free blocks is eventually bounded by the writeback throughput to backing storage, such as



(a) Stalled writes (ramdisk) (b) Queueing delay (64GB SSD) (c) Concurrent I/Os for writeback (SSD)

Figure 1: **Limitations of caching all writes.** *TPC-C throughput is normalized to the case of No NVWC.*

disks. The stalled writes problem becomes more serious in capacity-constrained devices, such as NV-DRAM and STT-MRAM. In order to quantify the impact of stalled writes on application performance, we ran a TPC-C benchmark [7] against PostgreSQL RDBMS [5] on a Linux-based system having NV-DRAM-based NVWC (emulated via ramdisk); see Section 7 for the detailed configuration. As shown in Figure 1(a), the TPC-C throughput normalized to the case without NVWC (i.e., disk only) drops from 1.99 to 1.09 as the capacity of NVWC decreases. This is because the frequency of write stalls in critical paths of PostgreSQL (e.g., stalls during log commit) is highly likely to increase as the ratio of stalled writes increases.

Secondly, caching all writes can incur significant congestion in OS- and device-level request queues of NVWC, thereby delaying the processing of critical writes. When the request queues of NVWC are congested, write requests need to wait at the queues even though the NVWC has sufficient free blocks. Moreover, queue congestion of a storage-based NVWC such as SSD can be exacerbated by concurrent I/Os for writing back cached data. The concurrent I/Os include NVWC reads for retrieving a dirty cache block into main memory and NVWC writes for updating the corresponding metadata. In order to measure the impact of aggravated queueing delay, we ran the TPC-C benchmark with a flash SSD-based NVWC. As shown in Figure 1(b), the average length of OS request queue increases as the number of clients increases, thereby gradually degrading the normalized TPC-C throughput. In addition, the performance further decreases as the concurrent I/Os increase as shown in Figure 1(c); the frequency of writebacks depends on the ratio of dirty blocks in NVWC for this measurement. In most cases, NVWC provides even lower performance than that without NVWC (up to 47% performance loss) though write stalls did not occur at all in all the configurations.

Finally, caching all writes would hurt reliability and performance depending on the characteristics of an NVWC device. For example, caching non-critical writes exacerbates the wear-out of an NVWC device, like flash and PCM, without any gain in application performance. In addition, caching non-critical writes can increase the probability of garbage collection while processing critical writes in flash-based NVWCs.

For these reasons, caching only critical writes to NVWC is vital to fully utilize a given NVWC device for application performance. From the analysis based on the realistic workload (Section 7.2), we found that all writes do not equally contribute to the application performance. This finding implies that there is a need to classify write I/Os for typical data-intensive applications such as databases and key-value stores.

3 Which Type of Write is Critical?

3.1 Request-Oriented Write Classification

The primary role of a data-intensive application is to provide a specific data service in response to an external *request*, like a PUT/GET request to a key-value store. In such an application, the performance of request processing determines the level of application performance a user perceives. Therefore, we need to identify which type of writes delays the progress of request processing to classify critical writes.

Synchronous writes can be a good candidate for the type of critical writes. Traditionally, write I/O is broadly classified into two categories in the system's viewpoint: asynchronous and synchronous. When a process issues an asynchronous write, it can immediately continue processing other jobs without waiting for the completion of the write. A synchronous write, on the other hand, is awaited by a requesting process until the write completes. Due to this difference, prioritizing synchronous writes over asynchronous ones is known as a reasonable method to reduce system-wide I/O wait time [35], and hence it is adopted in commodity OS [28].

However, not all synchronous writes are truly synchronous from the perspective of *request execution*. Typical data-intensive applications delegate a large amount of synchronous writes to a set of background processes as a way of carrying out internal activities. For instance, RDBMS [38, 59] and NoSQL store [25, 31] adopt a variant of logging technique that accompanies only a small amount of (mostly sequential) synchronous writes during request processing while conducting a burst of (mostly

Write Type	Process	Ratio (%)
Sync.	backends	44.312
	checkpointer	34.664
	log writer	0.368
	jbd2 (kernel)	0.094
	etc	0.007
Async.	kworker (kernel)	20.554
	etc	0.002

Table 1: **Breakdown of writes by type and process.**

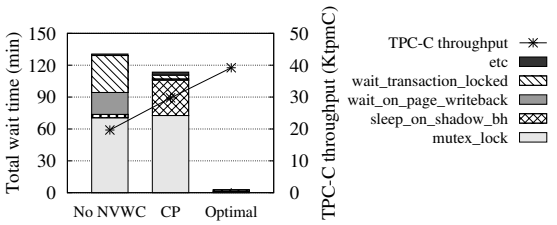


Figure 2: **Impact of dependencies on application performance.** *CP caches synchronous writes requested by critical processes while Optimal caches all writes without stalls. Network latencies are omitted for brevity.*

random) synchronous writes in background. This is an intrinsic design to achieve high degree of application performance without loss of durability by segregating costly synchronous writes from the critical path of request execution as much as possible.

To verify such behaviors, we ran the TPC-C benchmark using 24 clients without NVWC and recorded the type of write issued per process. As shown in Table 1, about 80% of the writes are synchronous, and most of them are performed by *backends* and *checkpointer*. In PostgreSQL, the backend is a dedicated process for handling requests while the checkpointer periodically issues a burst of synchronous writes to reflect buffer modifications to backing storage. Likewise, in kernel-level, journaling daemon (i.e., *jbd2*) also issues synchronous writes (though small amount in this case) for committing and checkpointing file system transactions. Basically, the synchronous writes requested by the processes other than the backends are irrelevant to request processing. Furthermore, according to our analysis result (Table 3), asynchronous writes occasionally block the backends because of complex synchronizations during runtime. The conventional synchrony-based classification, therefore, is inadequate for classifying critical writes.

We introduce *request-oriented write classification* that classifies a write awaited in the context of request execution as a critical write regardless of whether it is issued synchronously or not. Based on this classification, only critical writes are cached into NVWC while non-critical writes are routed to backing storage directly. As a result, a request can be handled quickly by avoiding excessive write stalls and queue congestion. In addition, device-specific reliability and performance issues, which are discussed in Section 2.2, can be eased without hurting application performance.

3.2 Dependency-Induced Critical Write

In data-intensive applications, one or more processes are involved in handling requests. Synchronous writes issued by these processes are definitely critical; hence, we refer to this type of processes as a *critical process*. Caching these synchronous writes alone, however, is insufficient for identifying all critical writes. This is because runtime dependencies generated by complex synchronizations among concurrent processes and I/Os make critical processes wait for the writes that are not synchronously issued by them.

There are two types of dependencies associated with write I/Os: a *process dependency* and an *I/O dependency*. The process dependency occurs when two processes interact with each other via synchronization primitives, such as a lock and condition variable. The process dependency complicates the accurate detection of critical writes because a non-critical process may issue synchronous writes within a critical section making a critical process indirectly wait for the completion of the writes. On the other hand, the I/O dependency occurs between a critical process and an ongoing write I/O. Basically, the I/O dependency is generated when a critical process needs to directly wait for the completion of an outstanding write in order to ensure consistency and/or durability.

In order to quantify the significance of the dependency problems, we measured the wait time of critical processes (i.e., PostgreSQL backends) using *LatencyTOP* [34] during the execution of the TPC-C benchmark using 24 clients with 4GB ramdisk-based NVWC. Figure 2 shows the impact of complex dependencies on the TPC-C throughput; CP caches only synchronous writes requested by critical processes while Optimal caches all writes without stalls and queue congestion. As we expect, CP mostly eliminates the latency incurred by synchronous writes (i.e., `wait_on_page_writeback()`). However, CP still suffers from excessive latencies mainly caused by process dependency (i.e., `mutex_lock()`) and I/O dependency (i.e., `sleep_on_shadow_bh()`). Note that the I/O dependency occurs because a critical process attempts to update a buffer page that is under writing back as the part of a committing file system transaction. Consequently, CP achieves only a half of the performance improvement compared to Optimal.

In addition, there are many other sources of excessive latencies in terms of the average and worst case as shown in Table 2. The read/write semaphore for serializing on-disk inode modifications represented as `down_read()` induces about one second and several seconds latencies in the average and worst case, respec-

tively. The journaling-related synchronizations for ensuring file system consistency also incur high latencies. In particular, `wait_transaction_locked()` is called to synchronize with all the processes updating the current file system transaction to complete their execution while `jbd2_log_wait_commit()` is called to wait for the journaling daemon to complete the commit procedure of a file system transaction. The synchronization methods that induce I/O dependency, such as `lock_buffer()` and `lock_page()`, delay the progress of critical processes up to several seconds. Though some of the synchronization methods account for a small portion of the total latency, they would increase tail latency, thereby degrading user experience in large-scale services [29]. Therefore, all synchronization methods causing latency to the critical processes should be handled properly in order to eliminate unexpected request latency spikes.

4 Critical Write Detection

4.1 Critical Process Identification

In order to detect all critical writes, we should identify critical processes in the first place. To do so, we adopt an application-guided approach that exploits application-level hints.

The main benefit of the application-guided approach is that it does not increase the complexity of the OS kernel. Accurately identifying critical processes without application guidance requires huge engineering effort to the kernel. For instance, similar to the previous approaches [83, 84], the kernel should track all inter-process communications and network-related I/Os to infer the processes handling requests. In addition, the kernel should adopt complex heuristics (e.g., feedback-based confidence evaluation [84]) to reduce the possibility of misidentification. On the other hand, an application can accurately decide the criticality of each process since the application knows the best which processes are currently involved in handling requests.

Though the application-guided approach requires application modifications, the engineering cost for the modifications is low in practice. This is because an application developer does not need to know the specifics of underlying systems since the hint (i.e., disclosure [64]) revealing a critical process remains correct even when the execution environment changes. In addition, typical data-intensive applications, such as MySQL [4], PostgreSQL, and Redis, already distinguish foreground processes (i.e., critical processes) from background processes. This distinction is also common for event-driven applications since they need to clearly separate internal activities from request flows as exemplified in Cassandra [43]. As a consequence, the required modification is only a few lines of code in practice; see Section 6 for our application studies.

Dep. Type	Synchronization Method	Avg (ms)	Max (ms)
Process	`down_read`	1088.09	6065.2
	`wait_transaction_locked`	493.05	4806.8
	`mutex_lock`	134.55	6313.55
	`jbd2_log_wait_commit`	40.96	391.36
I/O	`lock_buffer`	912.38	3811.35
	`sleep_on_shadow_bh`	225.25	3560.47
	`lock_page`	8.08	3009.84
	`wait_on_page_writeback`	0.04	19.12

Table 2: **Sources of dependencies.** *Average and maximum wait times of backends are shown in the CP case.*

Since a hint is solely used for deciding admission to NVWC, a wrong hint does not affect the correct execution of an application. However, hint abuse by a malicious or a thoughtless application may compromise performance isolation among multiple applications sharing NVWC. This problem can be solved by overriding criticality of each write at the kernel based on a predefined isolation policy. Addressing the issue resulting from sharing NVWC is out of scope of this paper.

4.2 Criticality Inheritance Protocols

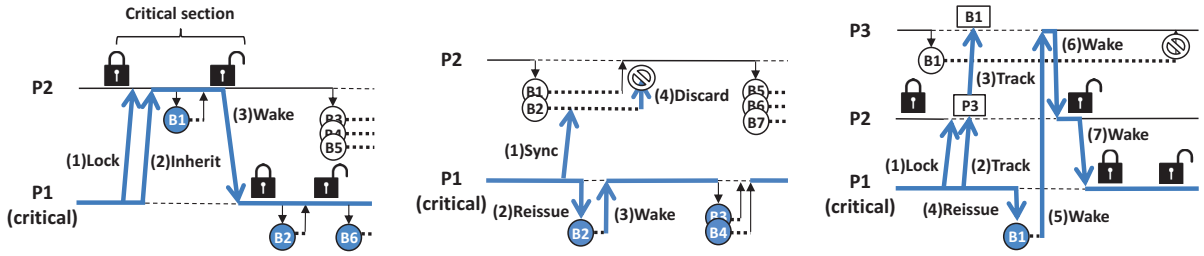
As we discussed in Section 3.2, the process and I/O dependencies can significantly delay the progress of a critical process. In the rest of this section, we explain our *criticality inheritance protocols* that effectively resolve the process and I/O dependencies.

4.2.1 Process Criticality Inheritance

Handling the process dependency has been well-studied in the context of process scheduling because the process dependency may cause priority inversion problem [51]. Priority inheritance [72] is the well-known solution for resolving the priority inversion problem.

Inspired by the previous work, we introduce *process criticality inheritance* to resolve the process dependency. Process criticality inheritance is similar to the priority inheritance in that a non-critical process inherits criticality when it blocks a critical process until it finishes its execution within the synchronized region. The main difference between process criticality inheritance and priority inheritance is that the former is used to prioritize I/Os whereas the latter is used to prioritize processes.

Figure 3(a) illustrates an example of process criticality inheritance: (1) critical process P1 attempts to acquire a lock to enter a critical section. (2) Non-critical process P2 inherits criticality from P1 since the lock is held by P2. Then, the synchronous write to block B1 issued by P2 is directed to NVWC to accelerate the write within



(a) Process criticality inheritance (b) I/O criticality inheritance (c) Handling cascading dependencies

Figure 3: **Criticality inheritance protocols.** *Thick lines represent a critical path of request execution while dotted lines indicate blocked execution. Circles and boxes represent write I/Os to specific blocks and blocking objects of specific processes, respectively. Thick arrows indicate specific actions described in the corresponding texts while thin arrows to/from write I/Os show I/O submission/completion.*

the critical path. (3) P2 wakes up P1 when its execution within the critical section has been completed, and P1 continues the rest of its job. In this example, the write latency of block B1 is minimized by using process criticality inheritance since it indirectly delays the request execution.

4.2.2 I/O Criticality Inheritance

Handling the I/O dependency is more complicated than that of the process dependency since inheriting criticality to the ongoing write to backing storage requires reissuing the write to NVWC without side effects (e.g., duplicated I/O completion). A possible solution for eliminating the side effects is canceling an outstanding write to a disk. In practice, however, canceling a specific ongoing write needs significant engineering efforts due to multiple abstraction stages in I/O stack. In Linux, for example, an I/O can be staged in either an OS queue managed by an I/O scheduler or a storage queue managed by a device firmware. Hence, the procedure of canceling requires lots of modifications to various in-kernel components.

In order to rapidly resolve the I/O dependency while maintaining low engineering cost, we devise immediate reissuing and I/O completion discarding as a technique for I/O criticality inheritance. Figure 3(b) describes the proposed mechanism: (1) critical process P1 needs to wait for the completion of the write request to block B2. (2) P1 reissues B2 to NVWC to resolve the I/O dependency between P1 and B2. (3) The event of I/O completion of the reissued B2 wakes up P1. (4) Later, the I/O completion of the original write to B2 is discarded to suppress the duplicated completion notification.

The main drawback of the proposed technique for I/O criticality inheritance is that it cannot eliminate unnecessary write traffic to a disk since it does not cancel ongoing writes to the disk. However, the performance penalty would be small since the duplicated write to a specific block is highly likely to be processed as a single sequential write merged with other writes to adjacent blocks in modern OSes. Moreover, discarding several blocks included in a write request may result in splitting the request into multiple requests, thereby decreasing the efficiency of I/O processing. In practice, the amount of reissued writes is insignificant despite of its large contribution to application performance (Table 3).

4.2.3 Handling Cascading Dependencies

Cascading dependencies, a chain of process and I/O dependencies, make precise detection of critical writes more difficult if the chain contains a process that is already blocked. For example, as illustrated in Figure 3(c), non-critical process P3 issues a synchronous write and is blocked to wait for the completion of the write. Later, non-critical process P2 sleeps while holding a lock because it needs to wait for an event from P3. In this situation, if critical process P1 attempts to acquire the lock that is held by P2, P1 blocks until the write issued by P3 is completed even though P2 inherits criticality from P1. We found that this scenario occurs in practice because of complex synchronization behaviors for ensuring file system consistency.

In order to handle the cascading dependencies, we record a blocking object to the descriptor of a process when the process is about to be blocked. There are two types of the blocking object in general: a process and an I/O for process dependency and I/O dependency, respectively. As a special case, a lock is recorded as the blocking object when a process should sleep to acquire the lock, in order to properly handle the cascading dependencies to both the lock owner and the waiters having higher lock-acquisition priority. Based on the recorded blocking object, a critical process can effectively track the cascading dependencies and can handle them using the process and I/O criticality inheritances.

Figure 3(c) demonstrates an example to describe how the cascading dependencies are handled: (1) critical pro-

cess P1 attempts to acquire the lock that is held by non-critical process P2. (2) Then, P2 inherits criticality from P1, and P1 checks P2's blocking object. (3) Since P2 is currently blocked waiting for an event from non-critical process P3, P3 also inherits criticality and P1 again checks P3's blocking object. (4) Due to P3 currently blocks on B1, P1 initiates reissuing of block B1 to NVWC and sleeps until the lock has been released by P2. (5) The I/O completion of the reissued B1 wakes up P3, and (6) P3 wakes up P2 after doing some residual work. (7) P2, in turn, wakes up P1 after completing its execution in the critical section. Finally, P1 enters the critical section and continues its job.

5 Implementation

We implemented our scheme on x86-64 Linux version 3.12. For the critical process identification, we added a pair of special priority values to dynamically set or clear criticality of a process (or a thread) to the existing `setpriority()` system call interface. We also added a field to the process descriptor for distinguishing criticality of each process.

For handling process and I/O dependencies, we implemented criticality inheritance protocols to blocking-based synchronization methods. Specifically, process and I/O criticality inheritances are implemented to the methods that synchronize with a process (e.g., `mutex_lock()`) and an ongoing I/O (e.g., `lock_buffer()`), respectively. In all the synchronization points, a blocking object is recorded into the descriptor of a process who is about to be blocked for synchronization.

We implemented our admission policy to FlashCache [2] version 3.1.1, which is a non-volatile block cache implemented as a kernel module. We modified the admission policy of FlashCache to cache only the writes synchronously requested by both critical and criticality-inherited processes. We also added the support for I/O criticality inheritance to FlashCache. In particular, the modified FlashCache maintains the list of outstanding non-critical writes to disk and searches the list when a critical process requests for reissuing a specific write. If the requested write is found in the list, FlashCache immediately reissues that write to NVWC and discards the result of the original write upon completion.

6 Application Studies

To validate the effectiveness of our scheme, we chose two widely deployed applications: PostgreSQL RDBMS [5] version 9.2 and Redis NoSQL store [23] version 2.8. For the critical process identification, we inserted eleven and two lines of code excluding comments to PostgreSQL and Redis, respectively. This result indicates that adopting the interface for critical process identification is trivial for typical data-intensive applications.

PostgreSQL RDBMS. In PostgreSQL, *backend* is dedicated to client for serving requests while other processes, such as *checkpointer*, *writer*, and *log writer*, carry out I/O jobs in background. The checkpointer flushes all dirty data buffers to disk and writes a special checkpoint record to the log file when the configured number of log files is consumed or the configured timeout happens, whichever comes first. The writer periodically writes some dirty buffers to disk to keep regular backend processes from having to write out dirty buffers. Similarly, the log writer periodically writes out the log buffer to disk in order to reduce the amount of synchronous writes needed for backend processes at commit time.

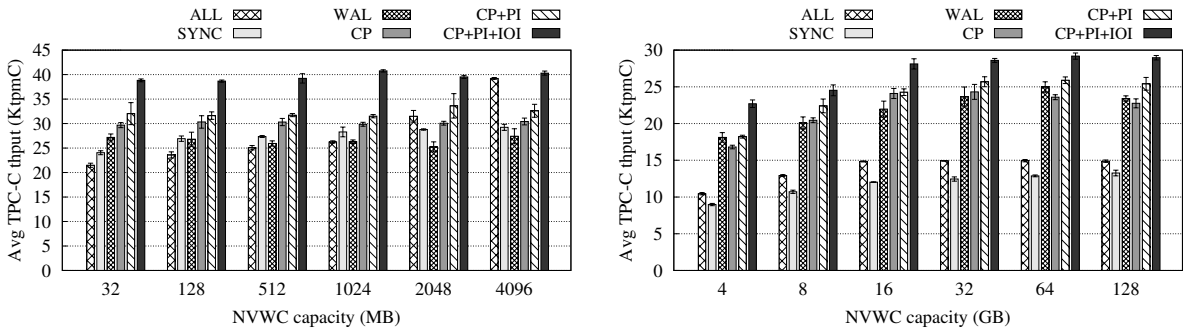
We classified backends as critical processes by calling the provided interface before starting the main loop of each backend. We also classified a process who is holding *WALWriteLock* as a temporary critical process because WALWriteLock is heavily shared between backends and other processes, and flushing the log buffer to a disk is conducted while holding the lock. This approach is similar to the priority ceiling [72] in that a process inherits criticality of a lock when the process acquires the lock.

Redis NoSQL store. Redis has two options to provide durability: snapshotting and command logging. The snapshotting periodically produces point-in-time snapshots of the dataset. The snapshotting, however, does not provide complete durability since up to a few minutes of data can be lost. The fully-durable command logging, on the other hand, guarantees the complete durability by synchronously writing an update log to a log file before responding back to the command. In the command logging, log rewriting is periodically conducted to constrain the size of the log file. Though the command logging can provide stronger durability than the snapshotting, it is still advisable to also turn the snapshotting on [25].

Similar to the PostgreSQL case, the snapshotting and log rewriting are conducted by child processes in background while a main server process serves all requests sequentially. Hence, we classified only the main server process as a critical process by calling the provided interface before starting the main event loop.

7 Evaluation

This section presents evaluation results based on the prototype implementation. We first detail the experimental environment. Then, we show the experimental results for both PostgreSQL and Redis to validate the effectiveness of the proposed scheme.



(a) Ramdisk NVWC (b) SSD NVWC

Figure 4: **PostgreSQL performance.** *TPC-C throughput is averaged over three runs for each admission policy.*

7.1 Experimental Setup

Our prototype was installed on Dell PowerEdge R420, equipped with two quad-core Intel Xeon E5-2407 2.4GHz processors and 16GB RAM; CPU clock frequency is set to the highest level for stable performance measurement. The storage subsystem is comprised of three 500GB 10K RPM WD VelociRaptor HDDs, one of which is dedicated to OS and the others are used as backing storage of NVWC. We used Ubuntu 14.04 with the modified Linux kernel version 3.12 as an OS and ext4 file system mounted with the default options.

For NVWC devices, we used a 4GB ramdisk (allocated from the main memory) and a 256GB Samsung 840 Pro SSD. To correctly emulate the persistency of the ramdisk-based NVWC in the existence of volatile CPU caches, we used the non-temporal memory copy described in [20] when the data is written to ramdisk. For the stable performance measurement of the SSD-based NVWC, we discard all blocks in SSD and give enough idle time before starting each experiment. In addition, in-storage volatile write cache was turned off to eliminate performance variations caused by internal buffering.

We used two criticality-oblivious admission policies: ALL and SYNC. ALL, which is the default of FlashCache, caches all incoming writes while SYNC caches only synchronous writes. In addition, we used three criticality-aware admission policies: CP, CP+PI, and CP+PI+IOI. CP caches synchronous writes requested by critical processes. CP+PI caches direct and cascading process dependencies-induced critical writes in addition to CP. CP+PI+IOI additionally caches direct and cascading I/O dependencies-induced critical writes.

7.2 PostgreSQL with TPC-C

We used TPC-C [7] as the realistic workload for PostgreSQL. We set TPC-C scale factor to ten, which corresponds to about 1GB of initial database, and simulated 24 clients running on a separate machine for 30 minutes. We report the number of New-Order transactions executed per minute (i.e., tpmC) as the performance metric. PostgreSQL was configured to have 512MB buffer pool, and the size of log files triggering checkpointing was set to 256MB. The database and log files are located on different HDDs according to the recommendation in the official document [6]. As the practical alternative of selective caching [21, 27, 42, 53], we used an additional policy denoted as WAL that caches all write traffics toward the log disk in NVWC. Since our work focuses on caching write I/Os, we eliminate read I/Os by warming up the OS buffer cache before starting the benchmark.

Performance with ramdisk-based NVWC. Figure 4(a) shows the TPC-C throughput averaged over three runs as the capacity of ramdisk-based NVWC increases from 32MB (scarce) to 4GB (sufficient). ALL achieves the lowest performance in the 32MB case because it stalls 58% of all writes. ALL gradually improves the performance as the NVWC capacity increases due to the reduction of write stalls. Note that the performance of ALL in the 4GB case is the optimal performance in our configuration because the capacity and bandwidth of NVWC are sufficient for absorbing all writes. SYNC slightly improves the performance compared to ALL in the low capacities since it reduces the number of write stalls by filtering out asynchronous writes. SYNC, however, cannot catch up the performance of ALL in the high capacities since it suffers from the dependencies induced by the asynchronous writes. Though WAL and CP do not suffer from write stalls at all in all the capacities, they achieve still lower performance than CP+PI and CP+PI+IOI due to runtime dependencies. CP+PI further improves performance by 4–12% over CP by handling process dependencies. CP+PI+IOI outperforms CP+PI by 18–29% by additionally handling I/O dependencies. Compared to ALL, CP+PI+IOI gains 80% performance improvement in the 32MB case and 72% reduction of cached writes without performance loss in the 4GB case.

To further analyze the reason behind the performance differences, we measured the wait time

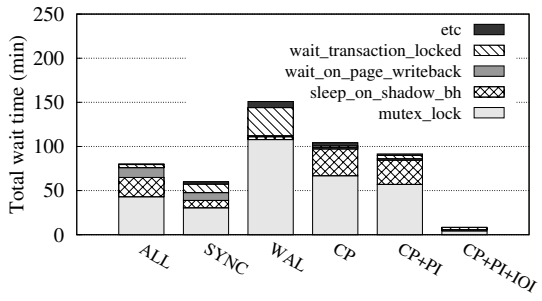


Figure 5: **Breakdown of PostgreSQL backends latency.** *512MB ramdisk is used as the NVWC device and network latencies are omitted for brevity.*

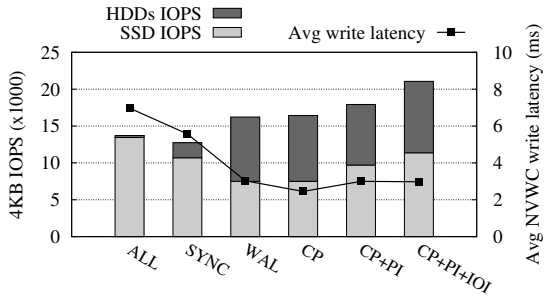


Figure 6: **Impact of queueing delay.** *16GB SSD is used as the NVWC device and synchronous write latency of critical processes is represented as NVWC write latency.*

of critical processes (i.e., PostgreSQL backends) in the 512MB NVWC case. As shown in Figure 5, ALL and SYNC incur the synchronous write latency (i.e., `wait_on_page_writeback`) and the mutex- and file system journaling-induced latencies (i.e., `sleep_on_shadow_bh` and `wait_transaction_locked`) described in Section 3, mainly due to frequent write stalls. Though WAL and CP mostly eliminate the synchronous write latencies by eliminating write stalls, they still incur excessive latencies mainly caused by the mutex and file system journaling. Though CP+PI further reduces latencies by resolving the mutex-induced dependency, it delays the progress of the critical processes because of unresolved I/O dependences. CP+PI+IOI eliminates most of the latencies since it additionally resolves I/O dependencies including the dependency to the journaling writes. As a result, CP+PI+IOI achieves the highest level of application performance in all the capacities.

Performance with SSD-based NVWC. Figure 4(b) shows the TPC-C throughput averaged over three runs as the capacity of SSD-based NVWC increases from 4GB to 128GB; the amount of concurrent I/Os for writeback decreases as the NVWC capacity increases. Unlike the case of the ramdisk-based NVWC, ALL achieves lower performance than the criticality-aware policies in all the

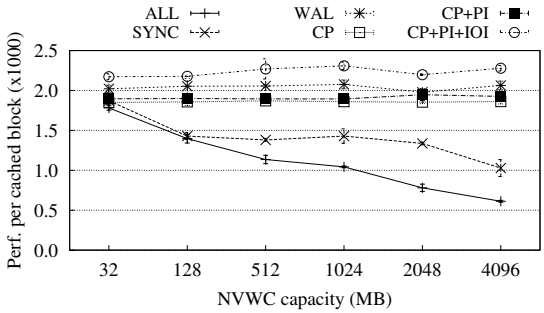


Figure 7: **Caching efficiency.** *Ramdisk is used as the NVWC device.*

capacities due to severe congestion in the request queues of the SSD. Though SYNC can ease the contention in the SSD more than ALL, it still has dependency problems incurred by the asynchronous writes directly routed to HDDs. On the other hand, WAL and the criticality-aware policies improve the application performance by reducing queue congestion compared to both ALL and SYNC. In particular, CP+PI+IOI outperforms ALL and SYNC by about 1.8–2.2× and 2.2–2.5×, respectively, because it minimizes the queueing delays of critical writes by filtering out more than a half of writes while effectively handling process and I/O dependencies.

To show the impact of queueing delay on critical writes, we measured 4KB IOPS for both the SSD and HDDs, and the average latency of synchronous writes requested by backends. As shown in Figure 6, ALL and SYNC utilize the SSD better than the other policies. This high utilization, however, causes severe congestion in the request queues of SSD, thereby delaying the processing of critical writes. WAL and the criticality-aware policies utilize both the SSD and the HDDs in a more balanced manner, thereby decreasing the queueing delay of critical writes.

Caching efficiency. In order to quantify the caching efficiency in terms of application performance, Figure 7 plots the performance per cached block as the capacity of the ramdisk-based NVWC increases. WAL and the criticality-aware policies show higher caching efficiencies compared to ALL and SYNC. Note that ALL and SYNC unexpectedly show high caching efficiency in the low NVWC capacities because FlashCache directs a write I/O to backing storage instead of waiting for a free block when there is no free block in NVWC. Overall, CP+PI+IOI utilizes NVWC more efficiently by 1.2–3.7× and 1.2–2.2× compared to ALL and SYNC, respectively.

Breakdown of critical writes. To help understand which types of data and I/O constitute critical writes, Table 3 shows the breakdown of critical writes in terms of data and I/O types. As we expect, the dominant type of data comprising critical writes is the logs that are synchronously written by backends during transaction com-

	Ratio (%)	CP	PI	IOI	Total
Data	Data (DB)	0	0.5209	0.0515	0.5723
	Data (LOG)	98.7409	0.5707	0.0007	99.3123
	Metadata	0	0	0.0014	0.0014
	Journal	0	0.0782	0.0357	0.1140
	Total	98.7409	1.1698	0.0893	100
I/O	Sync.	98.7409	1.0312	0.0357	99.8078
	Async.	0	0.1387	0.0535	0.1922
	Total	98.7409	1.1698	0.0893	100

Table 3: **Breakdown of critical writes.** *4GB ramdisk is used as the NVWC device in the case of CP+PI+IOI.*

mits. However, the rest of the critical writes is still crucial since it contributes to additional 32% performance improvement over CP alone (Figure 4(a)). On the other side, the dominant type of I/O comprising critical writes is synchronous writes. Though the portion of asynchronous writes is insignificant, it contributes to additional 38% performance improvement over SYNC (Figure 4(a)). Overall, dependency-induced critical writes have significant impact on application performance.

Performance disparity. Interestingly, we found the disparity between the system performance (i.e., IOPS) and the application performance (i.e., tpmC). For instance, as shown in Figure 8, ALL better utilizes NVWC by 40% than CP+PI+IOI leading to achieve 10% higher system performance in the 512MB ramdisk case. However, CP+PI+IOI accomplishes 57% higher application performance than that of ALL because CP+PI+IOI avoids write stalls in the critical paths. This result validates our argument on the necessity of the request-oriented approach in order to effectively utilize a given NVWC device.

7.3 Redis with YCSB

For Redis, we used the update-heavy (Workload A) and read-mostly (Workload B) workloads provided by the YCSB benchmark suite [26]. The data set was composed of 0.5 million objects each of which is 1KB in size. We simulated 40 clients running on a separate machine to generate ten millions of operations in total. We report operations per second (i.e., ops/sec) as the performance metric. We enabled both snapshotting and command logging according to the suggestion in the official document [25]. Due to the single threaded design of Redis [24], we concurrently ran four YCSB benchmarks against four Redis instances to utilize our multi-core testbed.

Performance. Figure 9 demonstrates the average YCSB throughput over three runs normalized to ALL. SYNC improves the performance over ALL since it filters out the asynchronous writes issued by the kernel thread that cleans the OS buffer cache. Unlike the case of PostgreSQL, CP shows significantly low performance

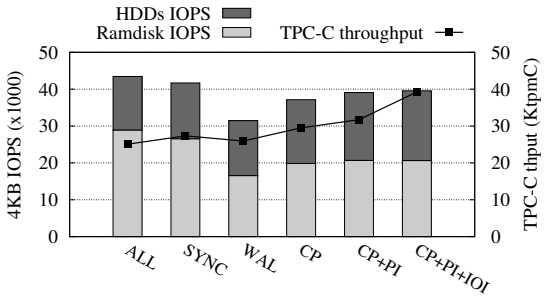


Figure 8: **Performance disparity.** *512MB ramdisk is used as the NVWC device.*

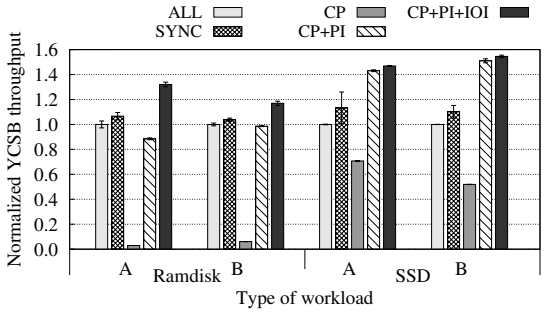


Figure 9: **Redis performance.** *512MB ramdisk and 16GB SSD are used as the NVWC devices.*

compared to the other policies because Redis frequently incurs synchronous writes conducted by a journaling daemon, which cannot be detected as the critical process by CP, within the critical path of update request. CP+PI dramatically improves the performance by 2–30× over CP because it additionally caches the journaling writes when there is a dependency between a critical process and the journaling daemon. CP+PI+IOI further improves the performance by 3–49% over CP+PI by additionally resolving the I/O dependencies mainly incurred by the synchronizations to guarantee the file system consistency. Though Workload B mostly consists of read requests, the performance is affected by the admission policy used. This is because Redis serves the requests from all clients sequentially using a single thread, thereby delaying the processing of read requests that are queued behind update requests. By providing the first class support for critical writes, CP+PI+IOI outperforms ALL by 17–32% and 47–55% while reducing cached write by 20–29% and 84–87% in the ramdisk- and SSD-based NVWC, respectively.

Tail latency. To show the impact of the admission policies on tail latency, we present the latency distribution of YCSB requests in the SSD-based NVWC case. As shown in Table 4, only CP+PI+IOI keeps 99.9^{th} and 99.99^{th}-percentile latencies below 100ms, which makes users feel more responsive than higher latencies [18]. ALL and SYNC, on the other hand, increase the 99.9^{th}-

	Latency (ms)	99th-%ile	99.9th-%ile	99.99th-%ile
A	ALL	80	649	>1000
	SYNC	72	678	>1000
	CP+PI+IOI	32	50	79
B	ALL	70	572	>1000
	SYNC	59	438	>1000
	CP+PI+IOI	23	32	83

Table 4: **Redis tail latency.** *16GB SSD is used as the NVWC device.*

percentile latency by an order of magnitude compared to that of CP+PI+IOI. Moreover, the 99.99^{th}-percentile latencies of ALL and SYNC exceeds one second, which is the maximum latency reported by YCSB. Considering the significance of providing consistent response latencies to users [16, 71] especially for large-scale services [29, 75], this result indicates that the proposed scheme is essential for providing high quality services to users.

8 Related Work

Non-volatile cache. A large volume of work has been done to efficiently utilize non-volatile caches based on NV-DRAM [15, 36, 37, 41], flash memory [19, 62, 70], and SCM [14, 33, 48, 56, 61]. In addition, the case of client-side non-volatile caches has been widely explored for networked storage systems [10, 11, 49, 65]. On the other side, researchers have extensively investigated the case of non-volatile cache optimized for database systems such as cost- and pattern-aware flash caches for relational database [32, 46, 55, 58] and persistent key-value store [30]. In addition, several studies have been investigated the case of dedicating NV-DRAM [27, 42] and flash memory [21, 53] to buffer or store transaction logs of relational databases. None of the previous work has taken the context of request execution into account for managing a non-volatile cache despite of its importance.

I/O classification. Prioritizing synchronous I/Os over asynchronous ones has been known as a reasonable method for improving system performance [28, 35]. Classifying I/Os based on explicit hints from data-intensive applications has been well-studied. Li *et al.* [54] proposed a cache replacement policy that exploits different write semantics in a relational database to maintain exclusivity between storage server and client caches. Later, Xin *et al.* [79] proposed a more general framework for the client hint-based multi-level cache management. Similarly, Mesnier *et al.* [60] proposed an I/O classification interface between computer and storage systems. For user-interactive desktop environments, Redline [76] statically gives higher I/O priority to interactive applications over non-interactive ones. Unlike the previous work, our classification scheme considers the I/O priority as dynamic property since it can be changed during runtime due to complex dependencies.

Request tracing. Request-oriented performance debugging has been widely explored for the end-user experience. Instrumentation-based profilers such as Project 5 [9] and MagPie [12] have been used for tracking request flows triggered by user requests. The Mystery Machine [22] and the lprof tool [82] extract the per-request performance behaviors from the log files to diagnose performance problems in large-scale distributed systems. In addition, Shen has studied architectural implications of request behavior variations in modern computer systems [73]. For user-interactive mobile platforms, AppInsight [68] and Panappticon [81] provide the information on the critical path of user request processing to application developers for improving user-perceived responsiveness. In this work, we focus on tracking request execution in the write I/O path and apply the acquired information to the admission policy of NVWC for improving application performance.

9 Conclusion and Future Direction

We present the request-oriented admission policy, which selectively caches the writes that eventually affect the application performance while preventing unproductive writes from occupying and wearing-out capacity-constrained NVWCs. The proposed scheme can contribute to reducing capital cost of expensive NVWCs satisfying desired service-level objectives. The results from the in-depth analysis on realistic workloads justify our claim that storage systems should consider the context of request execution to guarantee a high degree of application performance.

We plan to develop automatic critical process identification at kernel-level without an application hint in order to support legacy and proprietary applications. We also plan to apply the proposed classification to interactive systems, such as mobile systems, considering a direct user input as an external request.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Phillipa Gill, for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A1A10049626).

References

[1] Everspin Spin-Torque MRAM. `http://www.everspin.com/spinTorqueMRAM.php`.

[2] FlashCache. `https://github.com/facebook/flashcache`.

[3] Micron annouces availability of phase change memory for mobile devices. http://investors.micron.com/releasedetail.cfm?ReleaseID=692563.

[4] MySQL 5.7 reference manual. http://dev.mysql.com/doc/refman/5.7/en/threads-table.html.

[5] PostgreSQL. http://www.postgresql.org.

[6] PostgreSQL WAL internals. http://www.postgresql.org/docs/9.2/static/wal-internals.html.

[7] The TPC-C benchmark. http://www.tpc.org/tpcc.

[8] AGIGATECH. AGIGRAM (TM) Non-Volatile System. http://www.agigatech.com/agigaram.php.

[9] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP '03*.

[10] ARTEAGA, D., AND ZHAO, M. Client-side flash caching for cloud systems. In *Proceedings of the International Conference on Systems and Storage - SYSTOR '14*.

[11] BAKER, M., ASAMI, S., DEPRIT, E., OUSETERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '92*.

[12] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation - OSDI '04*.

[13] BHADKAMKAR, M., GUERRA, J., USECHE, L., BURNETT, S., LIPTAK, J., RANGASWAMI, R., AND HRISTIDIS, V. BORG: Block-reORGanization for self-optimizing storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies - FAST '09*.

[14] BHASKARAN, M. S., XU, J., AND SWANSON, S. Bankshot: Caching slow storage in fast non-volatile memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads - INFLOW '13*.

[15] BISWAS, P., RAMAKRISHNAN, K. K., AND TOWSLEY, D. Trace driven analysis of write caching policies for disks. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '93*.

[16] BRUTLAG, JAKE. Speed matters for Google Web search. http://googleresearch.blogspot.kr/2009/06/speed-matters.html.

[17] BURR, G. W., KURDI, B. N., SCOTT, J. C., LAM, C. H., GOPALAKRISHNAN, K., AND SHENOY, R. S. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development 52*, 4.5 (July 2008), 449–464.

[18] CARD, S. K., ROBERTSON, G. G., AND MACKINLAY, J. D. The information visualizer: An information workspace. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems - CHI '91*.

[19] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing - ICS '11*.

[20] CHEN, F., MESNIER, M. P., AND HAHN, S. A protected block device for persistent memory. In *Proceedings of the 30th Symposium on Mass Storage Systems and Technologies - MSST '14*.

[21] CHEN, S. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the 35th SIGMOD International Conference on Management of Data - SIGMOD '09*.

[22] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The Mystery Machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation - OSDI '14*.

[23] CITRUSBYTE. Redis. http://redis.io/.

[24] CITRUSBYTE. Redis latency problems troubleshooting. http://redis.io/topics/latency.

[25] CITRUSBYTE. Redis persistence. http://redis.io/topics/persistence.

[26] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing - SoCC '10*.

[27] COPELAND, G., KELLER, T., KRISHNAMURTHY, R., AND SMITH, M. The case for safe RAM. In *Proceedings of the 15th International Conference on Very Large Data Bases - VLDB '89*.

[28] CORBET, J. Solving the ext3 latency problem. http://lwn.net/Articles/328363/.

[29] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74.

[30] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment 3*, 1-2 (Sept. 2010), 1414–1425.

[31] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP '07*.

[32] DO, J., ZHANG, D., PATEL, J. M., DEWITT, D. J., NAUGHTON, J. F., AND HALVERSON, A. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data - SIGMOD '11*.

[33] DOH, I. H., LEE, H. J., MOON, Y. J., KIM, E., CHOI, J., LEE, D., AND NOH, S. H. Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems. In *Proceedings of the 2009 ACM Symposium on Applied Computing - SAC '09*.

[34] EDGE, JAKE. Finding system latency with LatencyTOP. http://lwn.net/Articles/266153/.

[35] GANGER, G. R., AND PATT, Y. N. The process-flow model: Examining I/O performance from the system's point of view. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '93*.

[36] GILL, B. S., KO, M., DEBNATH, B., AND BELLUOMINI, W. STOW: A spatially and temporally optimized write caching algorithm. In *Proceedings of the 2009 USENIX Annual Techinal Conference - USENIX '09*.

[37] GILL, B. S., AND MODHA, D. S. WOW: Wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies - FAST '05*.

[38] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1992.

[39] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro '09*.

[40] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies - FAST '12*.

[41] HAINING, T. R., AND LONG, D. D. E. Management policies for non-volatile write caches. In *Proceedings of the 1999 IEEE International Performance, Computing and Communications Conference - IPCCC '99*.

[42] HEISER, G., LE SUEUR, E., DANIS, A., BUDZYNOWSKI, A., SALOMIE, T.-L., AND ALONSO, G. RapiLog: Reducing system complexity through verification. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*.

[43] HEWITT, E. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[44] HUANG, P., WU, G., HE, X., AND XIAO, W. An aggressive worn-out flash block management scheme to alleviate SSD performance degradation. In *Proceedings of the 9th European Conference on Computer Systems - EuroSys '14*.

[45] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems - HotStorage '14*.

[46] KANG, W.-H., LEE, S.-W., AND MOON, B. Flash-based extended cache for higher throughput and faster recovery. *Proceedings of the VLDB Endowment 5*, 11 (July 2012), 1615–1626.

[47] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture - ISCA '08*.

[48] KIM, H., SESHADRI, S., DICKEY, C. L., AND CHIU, L. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies - FAST '14*.

[49] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies - FAST '13*.

[50] KOLLER, RICARDO AND RANGASWAMI, R. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies - FAST '10*.

[51] LAMPSON, B. W., AND REDELL, D. D. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (Feb. 1980), 105–117.

[52] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture - ISCA '09*.

[53] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data - SIGMOD '08*.

[54] LI, X., ABOULNAGA, A., SALEM, K., SACHEDINA, A., AND GAO, S. Second-tier cache management using write hints. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies - FAST '05*.

[55] LIU, X., AND SALEM, K. Hybrid storage management for database systems. *Proceedings of the VLDB Endowment 6*, 8 (June 2013), 541–552.

[56] LIU, Z., WANG, B., CARPENTER, P., LI, D., VETTER, J. S., AND YU, W. PCM-based durable write cache for fast disk I/O. In *Proceedings of the IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems - MASCOTS '12*.

[57] LOWELL, D. E., AND CHEN, P. M. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles - SOSP '97*.

[58] LUO, T., LEE, R., MESNIER, M., CHEN, F., AND ZHANG, X. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proceedings of the VLDB Endowment 5*, 10 (June 2012), 1075–1087.

[59] MALVIYA, N., WEISBERG, A., MADDEN, S., AND STONEBRAKER, M. Rethinking main memory OLTP recovery. In *Proceedings of the 30th IEEE International Conference on Data Engineering - ICDE '14*.

[60] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles - SOSP '11*.

[61] MILLER, E. L., BRANDT, S. A., AND LONG, D. D. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems - HOTOS '01*.

[62] OH, Y., CHOI, J., LEE, D., AND NOH, S. H. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies - FAST '12*.

[63] OUSTERHOUT, J., AND DOUGLIS, F. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review 23*, 1 (Jan. 1989), 11–28.

[64] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles - SOSP '95*.

[65] QIN, D., BROWN, A. D., AND GOEL, A. Reliable writeback for client-side flash caches. In *Proceedings of the 2014 USENIX Annual Technical Conference - USENIX '14*.

[66] QURESHI, M. K., KARIDIS, J., FRANCESCHINI, M., SRINIVASAN, V., LASTRAS, L., AND ABALI, B. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro '09*.

[67] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4.5 (July 2008), 465–479.

[68] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. AppInsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation - OSDI '12*.

[69] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference - USENIX '00*.

[70] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*.

[71] SCHURMAN, E., AND BRUTLAG, J. *The user and business impact of server delays, additional bytes, and HTTP chunking in Web search.* O'Reilly Velocity, 2009.

[72] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.

[73] SHEN, K. Request behavior variations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '10*.

[74] SOLWORTH, J. A., AND ORJI, C. U. Write-only disk caches. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data - SIGMOD '90*.

[75] TIMOTHY, Z., TUMANOV, A., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. PriorityMeister: Tail latency QoS for shared networked storage. In *Proceedings of the 2014 ACM Symposium on Cloud Computing - SoCC '14*.

[76] TING YANG, TONGPING LIU, EMERY D. BERGER, SCOTT F. KAPLAN, J. ELIOT, B. M. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation - OSDI '08*.

[77] VERMA, A., KOLLER, R., USECHE, L., AND RANGASWAMI, R. SRCMap: Energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies - FAST '10*.

[78] VIKING TECHNOLOGY. ArxCis-NV (TM) Non-Volatile Memory Technology. `http://www.vikingtechnology.com/arxcis-nv`.

[79] XIN LIU, ASHRAF ABOULNAGA, XUHUI LI, K. S. CLIC: Client-informed caching for storage servers. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies - FAST '09*.

[80] YANG, J., PLASSON, N., GILLIS, G., AND TALAGALA, N. HEC: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Conference on Systems and Storage - SYSTOR '13*.

[81] ZHANG, L., BILD, D. R., DICK, R. P., MAO, Z. M., AND DINDA, P. Panappticon: Event-based tracing to measure mobile application and platform performance. In *Proceedings of the 2013 International Conference on Hardware/Software Codesign and System Synthesis - CODES+ISSS '13*.

[82] ZHAO, X., ZHANG, Y., LION, D., FAIZAN, M., LUO, Y., YUAN, D., AND STUMM, M. lprof: A nonintrusive request flow profiler for distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation - OSDI '14*.

[83] ZHENG, H., AND NIEH, J. RSIO: Automatic user interaction detection and scheduling. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems - SIGMETRICS '10*.

[84] ZHENG, H., AND NIEH, J. SWAP: A scheduler with automatic process dependency detection. In *In Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation - NSDI '04*.

NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store

Leonardo Marmol[‡], Swaminathan Sundararaman[†], Nisha Talagala[†], Raju Rangaswami[‡]
[†]*SanDisk* [‡]*Florida International University*

Abstract

Key-value stores are ubiquitous in high performance data-intensive, scale out, and NoSQL environments. Many KV stores use flash devices for meeting their performance needs. However, by using flash as a simple block device, these KV stores are unable to fully leverage the powerful capabilities that exist within Flash Translation Layers (FTLs). NVMKV is a lightweight KV store that leverages native FTL capabilities such as sparse addressing, dynamic mapping, transactional persistence, and support for high-levels of lock free parallelism. Our evaluation of NVMKV demonstrates that it provides scalable, high-performance, and ACID compliant KV operations at close to raw device speeds.

1 Introduction

Flash-based key-value (KV) stores are becoming mainstream, with the importance of the KV interface to storage and flash technology have been well established through a gamut of implementations [13, 17, 21, 22, 31]. However, best utilizing the high-performance flash-based storage to drive the new generation of key-value stores continues to remain a challenge. The majority of the existing KV stores use a logging-based approach which induces significant additional *write amplification* (WA) at the KV software layer in addition to the internal WA caused by the FTL while managing physical flash.

Modern FTLs offer new capabilities that enable compelling, new design points for KV stores [6, 9]. Integration with these advanced capabilities results in an optimized *FTL-aware* KV store [33]. First, writing to the flash can be optimized to significantly improve both device lifetime and workload I/O performance. Second, modern FTLs already perform many functions that are similar to the functionality built into many KV stores such as log-structuring, dynamic data remapping, indexing, transactional updates, and thin provisioning [29, 35, 37]. Avoiding such replication of functionality can offer significant resource and performance benefits.

In this paper, we present the design, implementation, and evaluation of NVMKV, an FTL-aware KV store. NVMKV has been designed from the ground up to utilize the advanced capabilities found in modern FTLs. It implements a hashing-based design that uses the FTLs sparse address-space support to eliminate all write amplification at the KV layer, improving flash device endurance significantly relative to current KV stores. It is able to achieve single I/O `get/put` operations with performance close to that of the raw device, representing a significant improvement over current KV stores. NVMKV uses the advanced FTL capabilities of *atomic multi-block write*, *atomic multi-block persistent trim*, *exists*, and *iterate* to provide strictly atomic and synchronous durability guarantees for KV operations.

Two complementary factors contribute to increased collocation requirements for KV stores running on a single flash device. First, given the increasing flash densities, the performance points of flash devices are now based on capacity with larger devices being more cost-effective [42]. Second, virtualization supports increases in collocation requirements for workloads. A recent study has shown that multiple independent instances of such applications can have a counterproductive effect on the underlying FTL, resulting in increased WA [42]. NVMKV overcomes this issue by offering a new *pools* abstraction that allows transparently running multiple KV stores within the same FTL. While similar features exist in other KV stores, the FTL-aware design and implementation within NVMKV enables both efficient FTL coupling and KV store virtualization. NVMKV's design also allows for optimized flash writing across multiple KV instances and as a result lowers the WA.

In the quest for performance, KV stores and other applications are trending towards an in-memory architecture. However, since flash is still substantially cheaper than DRAM, any ability to offset DRAM for flash has the potential to reduce Total Cost of Ownership (TCO). We demonstrate how accelerating KV store access to flash can in turn result in similar or increased performance with much less DRAM.

We evaluated NVMKV and compared its performance to LevelDB. We evaluated the scalability of pools, compared it to multiple instances of LevelDB, and also found that NVMKV's atomic writes outperform both async and sync variants of LevelDB writes by up to 6.5x and 1030x respectively. NVMKV reads are comparable to that of LevelDB even when the workloads fit entirely in the filesystem cache, a condition that benefits LevelDB exclusively. When varying the available cache space, NVMKV outperforms LevelDB and more importantly introduces a write amplification of 2x in the worst case, which is small compared to the 70x for LevelDB. Finally, NVMKV improves YCSB benchmark throughput by up to 25% in in comparison to LevelDB.

2 Motivation

In this section, we discuss the benefits of better integration of KV stores with the FTL's capabilities. We also motivate other key tenets of our architecture, in particular the support of multiple KV stores on the same flash device and the need to increase performance with smaller quantities of DRAM.

An FTL-aware KV store: A common technique for performance improvement in flash optimized KV stores is some form of log structured writing. New data is appended to an immutable store and reorganized over time to reclaim space [1, 10, 26, 31]. The reclamation process, also called garbage collection or compaction, generates *Auxiliary Write Amplification (AWA)* which is the application level WA above that which is generated by the FTL. Unfortunately, AWA and the FTL induced WA have a multiplicative effect on write traffic to flash [43]. Previous work highlighted an example of this phenomenon with LevelDB, where a small amount of user writes can be amplified into as much as 40X more writes to the flash device [33]. As another example, the SILT work describes an AWA of over 5x [31]. NVMKV entirely avoids AWA by leveraging native addressing mechanisms and optimized writing implemented within modern FTLs.

Multiple KV stores in the device: The most recent PCIe and SAS flash devices can provide as much as 4-6TB of capacity per device. As density per die increases with every generation of flash driven by the consumer market, the multiple NAND dies required to generate a certain number of IOPs will come with ever increasing capacities as well as reduced endurance [27]. Multiple KV stores on a single flash device become cost effective but additional complexities arise. For instance, recent work shows how applications that are log structured to be flash optimal can still operate in suboptimal ways when either placed above a file system or run as multiple independent instances over a shared FTL [42]. NVMKV provides the ability to have multiple independent KV workloads share a device with minimal AWA.

Frugal DRAM usage: The ever increasing need for performance is driving the in-memory computing trend [7, 11, 24]. However, DRAM cost does not scale linearly with capacity since high capacity DRAM and the servers that support it are more expensive per unit of DRAM (in GB) than the mid-range DRAM and servers. The efficacy of using flash to offset DRAM has also been established in the literature [16]. In the KV store context, similar arguments have been made showing the server consolidation benefits of trading DRAM for flash [1]. A KV store's ability to leverage flash performance contributes directly to its ability to trade off DRAM for flash. NVMKV operates with high performance and low WA in both single and multiple instance KV deployments.

3 Building an FTL-aware KV Store

NVMKV is built using the advanced capabilities of modern FTLs. In this section, we discuss its goals, provide an overview of the approach, and describe its API.

3.1 Goals

NVMKV is intended for use within single node deployments by directly integrating it into applications. While it is not intended to replace the scale out key-value functionality provided by software such as Dynamo and Voldemort [23, 39], it can be used for single node KV storage within such scale out KV stores. From this point onward, we refer to such single node KV stores simply as KV stores. We had the following goals in mind when designing NVMKV:

Deliver Raw Flash Performance: Convert the most common KV store operations, `GET` and `PUT` into a single I/O per operation at the flash device to deliver close to raw flash device performance. As flash devices support high levels of parallelism, the KV store should also scale with parallel requests to utilize the performance scaling capacity of the device. Finally, when multiple, independent KV instances are consolidated on a single flash device, the KV store should deliver raw flash performance to each instance.

Minimize Auxiliary Write Amplification: Given the multiplicative effect on I/O volume due to WA, it is important to minimize additional KV store writes, which in turn reduces the write load at the FTL and the flash device. Reducing AWA improves KV operation latency by minimizing the number of I/O operations per KV operation as well as improvement of flash device lifetime.

Minimize DRAM Consumption: Minimize DRAM consumption by *(i)* minimizing the amount of internal metadata, and *(ii)* by leveraging flash performance to offset the amount of DRAM used for caching.

Simplicity: Leverage FTL capabilities to reduce code complexity and development time for the KV store.

3.2 Approach

Our intent with NVMKV is to provide the rich KV interface while retaining the performance of a much simpler block based flash device. NVMKV meets its goals by leveraging the internal capabilities of the FTL where possible and complementing these with streamlined additional functionality at the KV store level. The high level capabilities that we leverage from the FTL include:

Dynamic mapping: FTLs maintain an indirection map to translate logical addresses into physical data locations. NVMKV leverages the existing FTL indirection map to the fullest extent to avoid maintaining any additional location metadata. Every read and write operation simply uses the FTL indirection map and thereby operates

Category	API	Description
Basic	get(...)	Retrieves the value associated with a given key.
	put(...)	Inserts a KV pair into the KV store.
	delete(...)	Deletes a KV pair from the KV store.
Iterate	begin(...)	Sets the iterator to the beginning of a given pool.
	next(...)	Sets the iterator to the next key location in a given pool.
	get_current(...)	Retrieves the KV pair at the current iterator location in a pool.
Pools	pool_exist(...)	Determines whether a key exists in a given pool.
	pool_create(...)	Creates a pool in a given NVMKV store.
	pool_delete(...)	Deletes all KV pairs from a pool and deletes the pool from NVMKV store.
	get_pool_info(...)	Returns metadata information about a given pool in a KV store.
Batching	batch_get(...)	Retrieves values for a batch of specified keys.
	batch_put(...)	Sets the values for a batch of specified keys.
	batch_delete(...)	Deletes the KV pairs associated with a batch of specified keys.
	delete_all(...)	Deletes all KV pairs from a NVMKV store in all pools.
Management	open(...)	Opens a given NVMKV store for supported operations.
	close(...)	Closes a NVMKV store.
	create(...)	Creates a NVMKV store
	destroy(...)	Destroys a NVMKV store.

Table 1: **NVMKV API** *The table provides brief descriptions for the NVMKV API calls.*

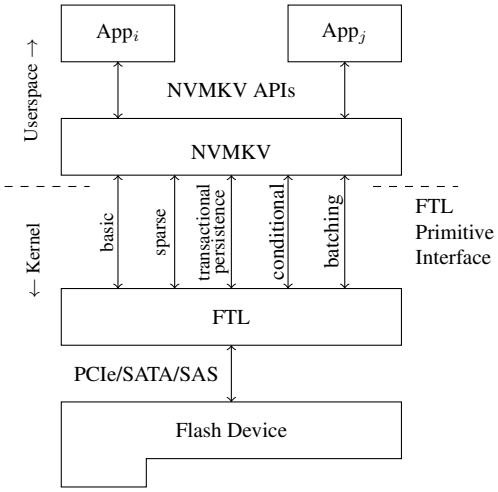


Figure 1: NVMKV System Architecture

at raw flash device performance by definition. This approach also reduces the additional DRAM overhead of NVMKV.

Persistence and transactional support: FTLs access data and metadata, and in particular maintain the persistence of indirection maps, at the speed of raw flash. NVMKV leverages this highly tuned capability to reduce the overhead for metadata persistence, logging, and checkpointing operations. Further, as FTLs operate as non-overwriting redirect-on-write stores, they can easily provide high performance transactional write semantics [35]. NVMKV leverages these capabilities to limit locking and journaling overheads.

Parallel operations: FTLs already implement highly parallel read/write operations while coordinating metadata access and updates. NVMKV leverages this FTL feature to minimize locking, thus improving scalability. We also define batch operations that are directly executed by the FTL to enable parallel KV store requests to be issued with lower I/O stack overhead [40].

3.3 NVMKV Architecture

NVMKV is a lightweight library in user space which interacts with the FTL through a *primitives interface* implemented as IOCTLs to the device driver that manages the flash device. Figure 1 shows the architecture of a system with NVMKV. Consumer applications, such as scale out KV stores, communicate with the NVMKV library using the NVMKV API. The NVMKV API calls are translated to underlying FTL primitives interface calls to be executed by the FTL.

3.4 NVMKV Consumer API

NVMKV's consumer applications interact with the library through the NVMKV API. We held discussions with the creators and vendors of several scale out KV stores to identify a set of operations commonly needed in a KV store. These operations formed the NVMKV API and they fall under five broad categories based on the functionality they provide. The categories are: *basic*, *iterate*, *pools*, *batching*, and *management*.

Table 1 presents the overview of the NVMKV API. We leverage the FTL's ability to provide enhanced operations such as *Atomic Writes* to provide transactional guarantees in NVMKV operations. Most existing KV stores do not offer such guarantees for their operations, and adopt more relaxed semantics such as eventual consistency to provide higher performance. On the other hand, we found that our approach enabled us to provide transactional guarantees with no loss of performance. We believe such guarantees can be of use to specific classes of applications as well as for simplifying the store logic

Category	API	Description
Basic	read(...)	Reads the data stored in the Logical Block Address (LBA).
	write(...)	Writes the data stored in buffer to destination LBA.
	trim(...)	Deletes (or discards) the mapping in FTL for the passed LBA range.
Sparse	exists(...)	Returns the presence of FTL mapping for the passed LBA.
	range_exist(...)	Returns the subset of LBA ranges that are mapped in the FTL.
	ptrim(...)	Persistently deletes the mapping in FTL for the passed LBA range.
	iterate(...)	Returns the next populated LBA starting from the passed LBA.
Transactional Persistence	atomic_read(...)	Executes read for a contiguous LBAs as an ACID transaction.
	atomic_exists(...)	Executes exists for a contiguous LBAs as an ACID transaction.
	atomic_write(...)	Executes write of a contiguous LBAs as an ACID transaction.
	atomic_ptrim(...)	Executes ptrim of a contiguous LBAs as an ACID transaction.
Conditional	cond_atomic_write(...)	Execute the atomic_write only if a condition is satisfied.
	cond_range_read(...)	Returns the data only from a subset of LBA ranges that are mapped in the FTL.
Batching		Operations within each category can be batched and executed in the FTL.

Table 2: **FTL Primitive Interface** *Enhanced FTL capabilities that NVMKV builds upon.*

contained within such applications. For instance, atomic KV operations imply that applications no longer need to be concerned with partial updates to flash.

The *Basic* and *Iterate* categories contain common features provided by many KV stores today. The *Pools* category interfaces allow for grouping KV pairs into buckets that can be managed separately within an NVMKV store. Pools provide the ability to transparently run multiple KV stores within the same FTL (discussed in more detail in § 6). The *Batching* category interfaces allow for group operations both within and across *Basic*, *Iterate*, and *Pools* categories, a common requirement in KV stores [18]. Finally, the *Management* category provides interfaces to perform KV store management operations.

4 Overview and FTL Integration

NVMKV's design is closely linked to the advanced capabilities provided by modern FTLs. Before describing its design in more detail, we provide a simple illustrative example of NVMKV's operation and discuss the advanced FTL capabilities that NVMKV leverages.

4.1 Illustrative Overview

To illustrate the principles behind NVMKV's design simply, we now walk through how a `get`, a `put`, and a `delete` operation are handled. We assume the sizes of keys and values are fixed and then address arbitrary sizes when we discuss design details (§5).

By mapping all KV operations to FTL operations, NVMKV eliminates any additional KV metadata in memory. To handle `put`s, NVMKV computes a hash on the key and uses the hash value to determine the location (i.e., LBA) of the KV pair. Thus, a `put` operation gets mapped to a write operation inside the FTL.

A `get` operation takes a key as input and returns the value associated with it (if the key exists). During a `get` operation, a hash of the key is computed first to determine the starting LBA of the KV pair's location. Using the computed LBA, the `get` operation is translated to a read operation to the FTL wherein the size of the read is equal to the combined sizes of the key and value. The stored key is matched with the key of the `get` operation and in case of a match, the associated value is returned.

To handle a `delete` operation, the given key is hashed to compute the starting LBA of the KV pair. Upon confirming that the key stored at the LBA is the key to be deleted, a discard operation is issued to the FTL for the range of LBAs containing the KV pair.

In this simplistic example, translating existing KV operations to FTL operations is straightforward and the KV store becomes a thin layer offloading most of its work to the underlying FTL with no in-memory metadata. However, additional work is needed to handle hash collisions in the LBA space and persisting discard operations.

4.2 Leveraging FTL Capabilities

We now describe the advanced FTL capabilities that are available and also extended to enable NVMKV. Many of these advanced FTL capabilities have already been used in other applications [20, 29, 35, 37, 43]. The FTL interface available to NVMKV is detailed in Table 2.

4.2.1 Dynamic Mapping

Conventional SSDs provide a dense address space, with one logical address for every advertised available physical block. This matches the classic storage model, but forces applications to maintain separate indexes to map items to the available LBAs. *Sparse address spaces* are available in advanced FTLs which allow applications to address the device via a large, thinly provisioned, virtual address space [35, 37, 43]. Sparse address entries are allocated physical space only upon a write. In the NVMKV context, a large address space enables simple mapping techniques such as hashing to be used with manageable collision rates.

Additional primitives are required to work with sparse address spaces. `EXISTS` queries whether a particular sparse address is populated. `PTRIM` is a persistent and atomic deletion of the contents at a sparse address. These primitives can be used for individual, or ranges of, locations. For example, `RANGE-EXISTS` returns a subset of a given virtual address range that has been populated. The `ITERATE` primitive is used to cycle through all populated virtual addresses, whereby `ITERATE` takes a virtual address and returns the next populated virtual address.

4.2.2 Transactional Persistence

The transactional persistence capabilities of the FTL are provided by `ATOMIC-WRITE` and `PTRIM` [20, 29]. `ATOMIC-WRITE` allows a sparse address range to be written as a single ACID compliant transaction.

4.2.3 Optimized Parallel Operations

The FTL is well-placed to optimize simultaneous device-level operations. Two classes of FTL primitives, *conditional* and *batch*, provide atomic parallel operations that are well-utilized by NVMKV. For example, `cond_atomic_write` allows for an atomic write to be completed only if a particular condition is satisfied, such as the LBA being written to is not already populated. This primitive removes the need to issue separate `exists` and `atomic_write` operations. Batch or vectored versions of all primitives are also implemented into the FTL (such as `batch_read`, `batch_atomic_write`, and `batch_ptrim`) to amortize lock acquisition and system call overhead. The benefits of batch (or vector) operations have been explored earlier [41].

5 Design Description

NVMKV implements novel techniques to make sparse addressing practical and efficient for use in KV stores and for providing low-latency, transactional persistence.

5.1 Mapping Keys via Hashing

Conventional KV stores employ two layers of translations to map keys to flash device locations, both of which need to be persistent [8, 10, 13]. The first layer translates keys to LBAs. The second layer (i.e., the FTL) translates the LBAs to physical locations in flash device. NVMKV leverages the FTLs sparse address space and encodes keys into sparse LBAs via hashing, thus collapsing an entire layer.

NVMKV divides the sparse address into equal sized virtual slots, each of which stores a single KV pair. More specifically, the sparse address space (with addressability through N bits) is divided into two areas: the *Key Bit Range* (KBR) and the *Value Bit Range* (VBR). This division can be set by the user at the time of creating the

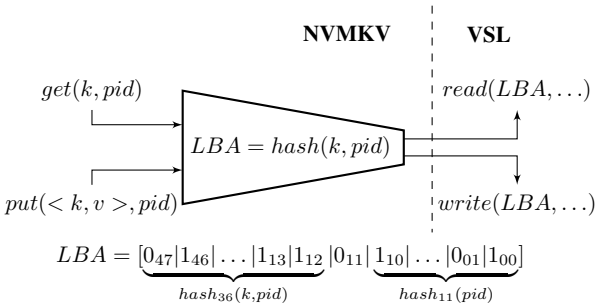


Figure 2: Hash model used in NVMKV. The arguments to the functions represent k:key, v:value, and pid:pool_id.

NVMKV store. The VBR defines the amount of contiguous address space (i.e., maximum value size or virtual slot size) reserved for each KV pair. The KBR determines the maximum number of such KV pairs that can be stored in a given KV store. In the expected use cases, the sparse virtual address range provided by the KBR will still be several orders of magnitude larger than the number of KV pairs as limited by the physical media.

The keys are mapped to LBAs through a simple hash model as shown in Figure 2. User supplied keys can be of variable length up to the maximum supported key size. To handle a `put` operation, the specified key is hashed into an address which also provides its KBR value. The maximum size of the information (Key, Value, metadata) that can be stored in a given VBR is half of the size addressed by the VBR. For example, if the VBR is 11 bits and each address represents a 512B sector, a given VBR value can address 2 MB.

The above layout guarantees the following two properties. First, each VBR contains exactly one KV pair, ensuring that we can quickly and deterministically search and identify KV pairs stored in the flash device. Second, no KV pairs will be adjacent in the sparse address space. In other words, there is always unpopulated virtual addresses between every KV pair. This deliberately wasted virtual space does not translate into unutilized storage since it is in virtual and physical space. These two properties are critical for NVMKV as the value size and exact start location of the KV pair are not stored as part of NVMKV metadata but are inferred via the FTL. Doing so helps in significantly reducing the in-memory metadata footprint of NVMKV. Non-adjacent KV pairs in the sparse address space help in determining the value size along with the starting virtual address of each KV pair. To determine the value size, NVMKV issues a `range_exist` call to the FTL.

A direct consequence of this design is that every access pattern becomes a random pattern, losing any possible order in the key space. The decision to not preserve sequentiality was shaped by two factors: metadata overhead and flash I/O performance. To ensure sequential

writes for contiguous keys, additional metadata would be required. This metadata would have to be consulted when reading, updated after writing, and cached in RAM to speed up the lookups. While straightforward to implement, doing so is unnecessary since the performance gap between random and sequential access for flash is ever decreasing; for current high performance flash devices, it is practically non-existent.

5.2 Handling Hash Collisions

Hashing variable sized keys into fixed size virtual slots could result in collisions. Since each VBR contains exactly one KV pair, hash conflicts only occur in the KBR. To illustrate the collision considerations, consider the following example. A 1TB Fusion-io ioDrive can contain a maximum of 2^{31} (2 billion) keys. Given a 48 bit sparse address space with 36 bits for KBR and 12 bits for VBR, NVMKV would accommodate 2^{36} keys with a maximum value size of 512KB. Under the simplifying assumption that our hash function uniformly distributes keys across the value ranges, for a fully-utilized 1TB ioDrive, the chances of a new key insertion resulting in a collision is $1/2^5$ or a little under 3 percent.

NVMKV implicitly assumes that the number of KBR values is sufficiently large, relative to the number of keys that can be stored in a flash device, so that the chances of a hash collision are small. If the KV pair sizes are increased, the likelihood of a collision reduces because the device can accommodate fewer keys while preserving the size of the key address space. If the size of the sparse address space is reduced, the chances of a collision will increase. Likewise, if the size of the flash device is increased without increasing the size of its sparse address space, the likelihood of a collision will increase.

Collisions are handled deterministically by computing alternate hash locations using either linear or polynomial probing. By default, NVMKV uses polynomial probing and up to eight hash locations are tried before NVMKV refuses to accept a new key. With this current scheme, the probability of a `put` failing due to hash failure is vanishingly small. Assuming that the hash function uniformly distributes keys, the probability of a `put` failing equals the probability of 8 consecutive collisions. This is approximately $(1/2^5)^8 = 1/2^{40}$, roughly one failure per trillion `put` operations. The above analysis assumes that the hash function used is well modeled by a uniformly distributing random function. Currently, NVMKV uses the FNV1a hash function [5] and we experimentally validated our modeling assumption.

5.3 Caching

Caching is employed in two distinct ways within NVMKV. First, a read cache speeds up access to frequently read KV pairs. NVMKV's read cache implementation is based on LevelDB's cache [26]. The read cache size is configurable at load time. Second, NVMKV uses a collision cache to improve collision handling performance. It caches the key hash (the sparse LBA) along with the actual key which is used during `put`s (i.e., inserts or updates). If the cached key matches the key to be inserted, the new value can be stored in the corresponding slot (the key's hash value). This significantly reduces the number of additional I/Os needed during collision resolution. In most cases, only a single I/O is needed for a `get` or a `put` to return or store the KV pair.

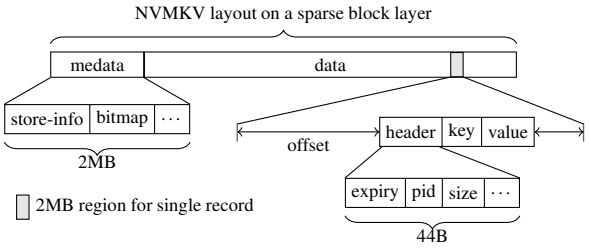


Figure 3: NVMKV layout

5.4 KV Pair Storage and Iteration

KV pairs are directly mapped to a physical location in the flash device and addressable through the FTL's sparse address space. In our current implementation, the minimum unit of storage is a sector and KV pairs requiring less than 512B will consume a full 512B sector. Each KV pair also contains metadata stored on media. The metadata layout is shown in Figure 3; it includes the length of the key, the length or the value, pool identifier (to be discussed further in §6), and other information.

To minimize internal fragmentation, NVMKV packs and stores the metadata, the key, and the value in a single sector whenever possible. If the size of the KV pair and the metadata is greater than a sector, NVMKV packs the metadata and key into the first sector and stores the value starting from the second sector. This layout allows for optimal storage efficiency for small values and zero-copy data transfer into the users buffer for larger values.

NVMKV supports unordered iteration through all KV pairs stored in the flash device. Key iteration is accomplished by iterating across the populated virtual addresses inside the FTL in order. The iterator utilizes the `ITERATE` primitive in the FTL, which takes in the previously reported start virtual address and returns the start address of the next contiguously populated virtual address segment in the sparse address space. Note that this approach relies on the layout guarantee that each KV pair is located contiguously in a range of virtual addresses, and that there are unpopulated virtual addresses in between each KV pair.

5.5 Optimizing KV Operations

NVMKV's design goal is one I/O for a `get` or a `put` operation. For `get`, this is achieved with the KV data layout and the `CONDITIONAL-RANGEREAD` primitive. The layout guarantees that individual KV pairs occupy a single contiguous section of the sparse address space, separated from other KV pairs by unpopulated virtual addresses. Given this, a `CONDITIONAL-RANGEREAD` can retrieve the entire KV pair in one operation without knowing the size of the value up front. Second, collisions induce a minimal number of additional operations. Since `get` and `put` operations for a given key map to hash addresses in a deterministic order, and since `put` places the new KV pair at the first available hash address (that is currently unused) in this order, subsequent `get`s are guaranteed to retrieve the most recent data written to this key. Finally, `DELETE` operations may require more than one I/O per operation, since they are required to read and validate the key before issuing a `PTRIM`. It also needs to check multiple locations to ensure that previous instances of a particular key have all been deleted.

NVMKV is intended to be zero copy and avoid memory comparison operations wherever possible. First, for any value that is large enough to start at its own sector, the data retrieved from a `get` (or written during a `put`) operation will be transferred directly to (or from) the user provided memory buffer. Second, no key comparisons occur unless the key hashes match. Given that the likelihood of collisions is small, the number of key comparisons that fail is also correspondingly small.

6 Multiple KV Instances Via Pools

Pools in NVMKV allow applications to group related keys into logical abstractions that can then be managed separately. Besides simplifying KV data management for applications, pools enable efficient access and iteration of related keys. The ability to categorize or group KV pairs also improves the lifetime of flash devices.

6.1 Need for Pools

NVMKV as described thus far, can support multiple independent KV stores. However, it would need to either partition the physical flash device to create multiple block devices each with its own sparse address space or logically partition the single sparse address space to create block devices to run multiple instances of KV stores.

Unfortunately, both approaches do not work well for flash. Since it is difficult to predict the number of KV pairs or physical storage needed in advance, static partitioning would result in either underutilization or insufficient physical capacity for KV pairs. Further, smaller capacity physical devices would increase pressure on the garbage collector, resulting in both increased write amplification and reduced KV store performance. Alternatively, partitioning the LBA space would induce higher key collision rates as the KBR would be shrunk depending on the number of pools that need to be supported.

6.2 Design Overview

NVMKV encodes pools within the sparse LBA to avoid any need for additional in-memory pool metadata. The encoding is done by directly hashing both the pool ID and the key to determine the hash location within the KBR. This ensures that all KV pairs are equally distributed across the sparse virtual address space regardless of which pool they are in. Distributing KV pairs of multiple pools evenly across the sparse address space not only retains the collision probability properties but also preserves the `get` and `put` performance with pools.

Pool IDs are also encoded within the VBR to optimally search or locate pools within the sparse address space. Encoding pool IDs within the VBR preserves the collision properties of NVMKV. The KV pair start offset within the VBR determines the Pool ID. The VBR size determines the maximum number of pools that can be addressed without hashing, while also maintaining the guarantee that each KV pair is separated from neighboring KV pairs by unpopulated sparse addresses. For example, with a 12 bit VBR, the maximum number of pools that can be supported without pool ID hashing is 1024. If the maximum number of pools is greater than 1024, the logic of `get` is modified to also retrieve the KV pair metadata that contains the pool ID now needed to uniquely identify the KV pair.

6.3 Operations

Supporting pools requires changes to common operations of the KV store. We now describe three important operations in NVMKV that have either been added or significantly modified to support pools.

Creation and Deletion: Pool creation is a lightweight operation. The KV store performs a one-time write to record the pool's creation in its persistent configuration metadata. On the other hand, pool deletion is an expensive operation since all the KV pairs of a pool are distributed across the entire LBA space, each requiring an independent `PTRIM` operation. NVMKV implements pool deletion as an asynchronous background operation. Upon receiving the deletion request, the library marks the pool as invalid in its on-drive metadata, and the actual deletion of pool data occurs asynchronously.

Iteration: NVMKV supports iteration of all KV pairs in a given pool. If no pool is specified, all key-value pairs on the device are returned by the iteration routines. Iteration uses the `ITERATE` primitive of the FTL to find the address of the next contiguous chunk of data in the sparse address space. During pool iteration, each contiguous virtual address segment is examined as before.

However, the iterator also examines the offset within the VBR of each starting address, and compares it to the pool ID, or the hash of the pool ID, for which iteration is being performed. Virtual addresses are only returned to the KV store if the pool ID match succeeds.

NVMKV guarantees that each KV pair is stored in a contiguous chunk, and that adjacent KV pairs are always separated by at least one empty sector, so the address returned by `ITERATE` locates the next KV pair on the drive (see §5.1). When the maximum number of pools is small enough that each pool ID can be individually mapped to a distinct VBR offset, the virtual addresses returned by the `ITERATE` primitive are guaranteed to belong to the pool currently being iterated upon. When the maximum number of pools is larger, the `ITERATE` uses the hash of the pool ID for comparison. In this case, the virtual addresses that match are not guaranteed to be part of the current pool, and a read of the first sector of the KV pair is required to complete the match.

7 Implementation

NVMKV is implemented as a stand-alone KV store written in C++ using 6300 LoC. Our current prototype works on top of ioMemory VSL and interacts with the FTL using the IOCTL interface [25]. The default subdivision for KBR and VBR used in the current implementation is 36 bits and 12 bits respectively, in a 48 bit address space. The KBR/VBR subdivision is also configurable at KV store creation time. To accelerate pool iteration, we implemented filters inside the `ITERATE/BATCH-ITERATE` FTL primitives. During the iteration of keys from a particular pool, the hash value of the pool is passed along with the IOCTL arguments to be used as a filter for the iteration. The FTL services `(BATCH-)ITERATE` by returning only populated ranges that match the filter. This reduces data copying across the FTL and NVMKV.

7.1 Extending FTL Primitives

We extended the FTL to better support NVMKV. `ATOMIC-WRITE` and its vectored forms are implemented in a manner similar to what has been described by Ouyang *et al.* [35]. Atomic operations are tagged within the FTL log structure, and upon restart, any incomplete atomic operations are discarded. Atomic writes are also not updated in the FTL map until they are committed to the FTL log to prevent returning partial results. `ITERATE` and `RANGE-EXISTS` are implemented as query operations over the FTL indirection map. `CONDITIONAL-READ and CONDITIONAL-WRITE` are emulated within NVMKV in the current implementation.

7.2 Going Beyond Traditional KV Stores

NVMKV provides new capabilities with strong guarantees relative to traditional KV stores. Specifically, it provides full atomicity, isolation, and consequently serializability for basic operations in both individual and batch submissions. Atomicity and serializability guarantees are provided for individual operations within a batch, not for the batch itself. The atomicity and isolation guarantees provided by NVMKV rely heavily on the `ATOMIC-WRITE` and `PTRIM` primitives from the FTL.

Each `put` is executed as a single ACID compliant `ATOMIC-WRITE`, which guarantees that no `get` running in parallel will see partial content for a KV pair. The `get` operation opportunistically retrieves the KV pair from the first hash location using `cond_range_read` to guarantee the smallest possible data transfer. In the unlikely event of a hash collision, the next hash address is used. Since the hash address order is deterministic, and every `get` or `put` to the same key will follow the same order, and every write has atomicity and isolation properties, `get` is natively thread safe requiring no locking.

When `ATOMIC-WRITE`s are used, `put` operations require locking for thread safety because multiple keys can map to the same KBR. When a `CONDITIONAL-WRITE` (which performs an atomic `EXISTS` check and `WRITE` of the data in question) is used, `put` operations can also be made natively thread safe. Individual iterator calls are thread safe with respect to each other and to `get`/`put` calls; thus, concurrent iterators can execute safely.

The `ITERATE` primitive is also supported in batch mode for performance. `BATCH-ITERATE` returns multiple start addresses in each invocation, reducing the number of IOCTL calls. For each LBA range returned, the first sector needs to be read to retrieve the key for the target KV pair.

8 Evaluation

Our previous work established performance of the basic approach used in NVMKV, contrasting it relative to block device performance [33]. Our evaluation addresses a new set of questions:

(1) How effective is NVMKV in supporting multiple KV stores on the same flash device? How well do NVMKV *pools* scale?

(2) How effective is NVMKV in trading off DRAM for flash by sizing its read cache?

(3) How effective is NVMKV in improving the endurance of the underlying flash device?

(4) How sensitive is NVMKV to the size of its collision cache?

8.1 Workloads and Testbed

We use LevelDB [26], a well-known KV store as the baseline for our evaluation of NVMKV. LevelDB uses a logging-based approach to write to flash and uses compaction mechanisms for space reclamation. Our eval-

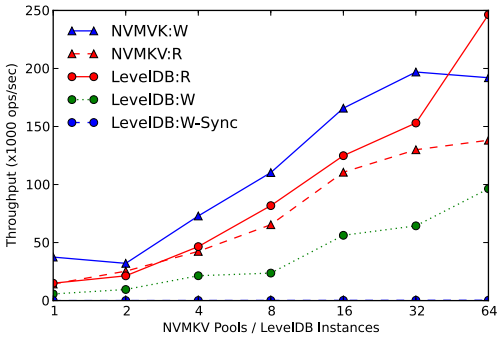


Figure 4: Comparing multiple identical instances of LevelDB with a single instance of NVMKV with equal number of pools.

uation consists of two parts. We answer question (1) above using dedicated micro-benchmarks for NVMKV and LevelDB. We then answer questions (2), (3), and (4) using the YCSB macro-benchmark [19]. We used the YCSB workloads A, B, C, D, and F with a data set size of 10GB; workload E performs short range-scans and the YCSB Java binding for NVMKV does not support this feature currently. We also used a raw device I/O micro-benchmark which was configured so that I/O sizes were comparable to the sizes of the key-value pairs in NVMKV. Our experiments were performed on two testbeds, I and II. Testbed I was a system with a Quad-Core 3.5 GHz AMD Opteron(tm) Processor, 8GB of DDR2 RAM, and a 825GB Fusion-io ioScale2 drive running Linux Ubuntu 12.04 LTS. Testbed II was a system with a 32 core Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz with 128 GB DDR3 RAM, and a 1.2TB Fusion-io ioDrive2 running Linux Ubuntu 12.04.2 LTS.

8.2 Micro-Benchmarks

Our first experiment using Testbed II answers question (1). We ran a single instance of NVMKV and measured the throughput of reads and writes as functions of the number of NVMKV pools. NVMKV also used as many threads as pools. We compared its performance against multiple instances of LevelDB. Both KV stores were configured to use the same workload, sizes of key-value pairs, and accessed a total of 500 MB of data. In addition, LevelDB used both its own user-level cache of size 1GB and the operating system's file system cache as well. On the other hand, NVMKV used neither. LevelDB provides two options for writes, a low-performing but durable *sync* and the high performing *async*, and we include them both here. NVMKV, on the other hand, performs all writes synchronously and atomically, and thus only a synchronous configuration is possible.

Figure 4 provides a performance comparison. Due to its low-latency flash-level operations, NVMKV almost equals LevelDB's primarily in-memory performance for up to 32 pools/instances. LevelDB continues scaling beyond 32 parallel threads; its operations continue to be memory-cache hits while NVMKV must perform flash-level accesses (wherein parallelism is limited) for each operation. When writing, NVMKV outperforms LevelDB's *sync* as well as *async versions* despite not using the filesystem cache at all. Even when LevelDB was configured to use async writes, it was about 2x slower than NVMKV in the best case, and about 6.5x slower at its worst. For synchronous writes, a more comparable setup, NVMKV outperforms LevelDB between 643x (64 pools) and 1030x faster (1 pool).

8.3 DRAM Trade-off and Endurance

This second experiment using Testbed I addresses questions (2) and (3). NVMKV uses negligible in-memory metadata and does not use the operating system's page cache at all. It implements a read cache whose size can be configured, allowing us to trade-off DRAM for flash, thus providing a tunable knob for trading off cost for performance. To evaluate the effectiveness of the collision cache, we evaluate two variants of NVMKV, one without the collision cache and the other when it uses 64MB of collision cache space. We used the YCSB benchmark for this experiment. We present the results from both the *load phase*, that is common to all workload personalities implemented in YCSB, and the *execution phase*, that is distinct across the workloads.

Figure 5 (top) depicts throughput as a function of the size of the application-level read cache available to LevelDB and NVMKV. Unlike NVMKV, LevelDB accesses use the file system page cache as well. Despite this, NVMKV outperforms LevelDB during both phases of the experiment, *load* and *execution*, by a significant margin. Further, the gap in performance increases as the size of the cache increases for every workload. This is because YCSB's workloads favor reads in general, varying from 50%, in the case of workload A, all the way to 100% in the case of workload C. Furthermore, the YCSB workloads follow skewed data access distributions, making even a small amount of cache highly effective.

To better understand these results, we also collected how much data was written to the media while the experiments were running. All workload were configured to use 10GB of data, so any extra data that is written to the media is overhead introduced by NVMKV or LevelDB. Figure 5 (bottom) depicts the results of the write amplification. By the end of each experiment, LevelDB has written anywhere from 42.5x to 70x extra data to the media. This seems to be a direct consequence of its internal design which migrates the data from one level to the next, therefore copying the same data multiple times as it ages. NVMKV on the other hand, introduces a write amplification of 2x in the worst case. We believe this to be

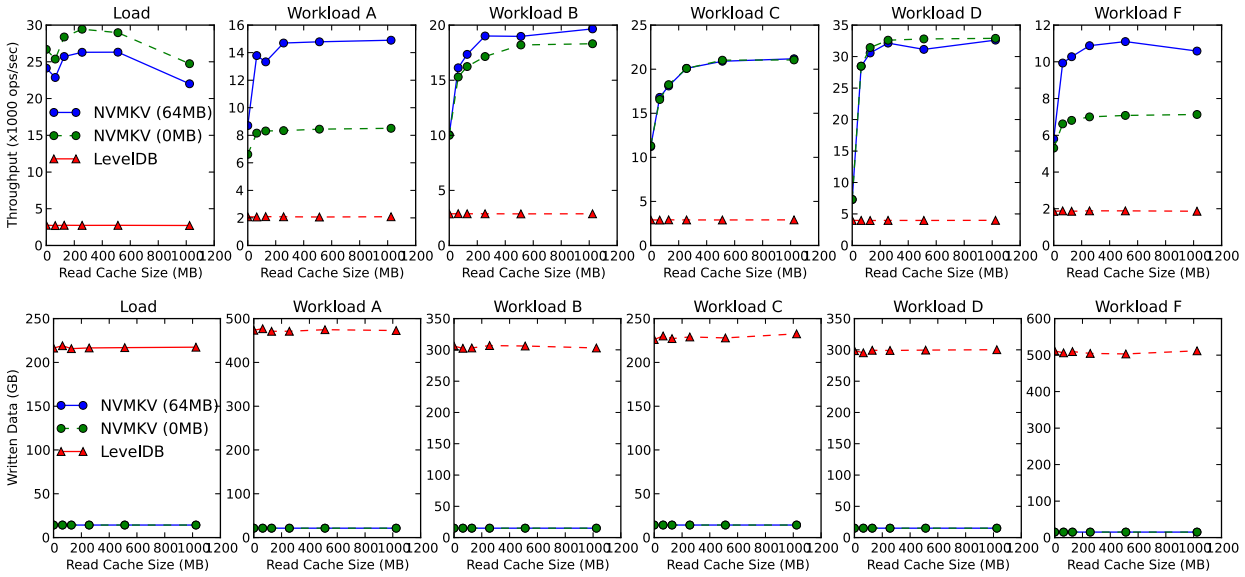


Figure 5: Throughput comparison between NVMKV and LevelDB using YCSB workloads (above). Write Amplification Comparison between NVMKV and LevelDB using YCSB workloads (below).

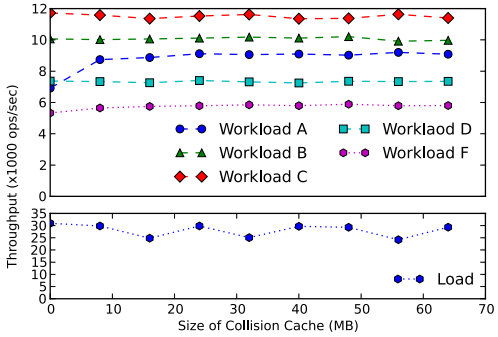


Figure 6: Collision cache impact on YCSB workloads.

the main reason for the performance difference between the two KV Stores.

Finally, the effect of NVMKV's collision cache is highly sensitive to the workload type. As expected, for read-intensive workloads (i.e., B, C, and D) the presence of collision cache has little to no impact at all. On the other hand, the update heavy workloads (i.e., A and F) benefit significantly from the collision cache, increasing the performance up to 76% and 56% respectively. Surprisingly the load phase is negatively affected by the collision cache, and performance decreases by up to 11%.

8.4 Effectiveness of the Collision Cache

We used Testbed II to address question (4). We measured YCSB workload throughput when varying the size of the collision cache in NVMKV. During this experiment, the read cache was disabled to eliminate other caching effects. As shown in Figure 6, the presence of the collision cache benefits workloads A and F with a throughput improvement of 28% and 10% respectively. Read-mostly workloads (B, C, and D) do not benefit from the collision cache since the probability of collision is low and a single flash-level read is necessary to service KV `GET` operations. A and F involve writes and these benefit from the collision cache. The collision cache optimizes the handling repeated writes to the same location by eliminating the reading of the location (to check for collisions) prior to the write. Finally, the loading phase does not demonstrate any benefit from the collision cache mainly because of YCSB's key randomization during inserts.

9 Discussion and Limitations

Through the NVMKV implementation, we were able to achieve a majority of our design goals of building a flash-aware lightweight KV store that leverages advanced FTL capabilities. We made several observations through the design and development process.

It is valuable for atomic KV operations, such as those described by Ouyang *et al.* [35], to be fully ACID compliant. The usage described in Ouyang *et al.*'s work only required the durable writes to have the atomicity property. We found that having *isolation* and *consistency* enables reduced locking and in some cases, fully lock free operation, at the application level. For example, updating multiple KV pairs atomically as a single batch can help provide application-level consistent KV store state without requiring additional locks or logs.

Many primitives required by NVMKV are the same as those required by other usages of flash. FlashTier, a primitives based solid state cache leverages the sparse

addressing model, and the EXISTS and PTRIM FTL primitives [37], as does DirectFS, a primitives based filesystem [4, 29].

NVMKV suffers from internal fragmentation for small KV pairs. Since we map individual KV pairs to separate sectors, NVMKV will consume an entire sector (512B) even for KV pairs smaller than the sector size. While this does not pose a problem for many workloads, there are those for which it does. For the second group of workloads, NVMKV will have poor capacity utilization. One way to manage efficient storage of small KV pairs is to follow a multi-level storage mechanism, as provided in SILT [31], where small items are initially indexed separately and later compacted into larger units such as sectors. We believe that implementing similar methods within the FTL itself can be valuable.

10 Related Work

Most previous work on FTL-awareness has focused on leveraging FTL capabilities for simpler and more efficient applications, focusing on databases [35], file systems [29] and caches [37]. NVMKV is the first to present the complete design and implementation of an FTL-aware KV store and explains how specific FTL primitives can be leveraged to build a lightweight and performant KV store. NVMKV is also the first to provide support for multiple KV instances (i.e., *pools*) on the same flash device. Further, NVMKV trades-off main memory for flash well as evidenced in the evaluation of a read cache implementation. Finally, NVMKV extends the use of the FTL primitives in a KV store to include conditional-primitives and batching.

There is substantial work on scale-out KV stores and many of the recent ones focus on flash. For example, Dynamo [23] and Voldemort [39] both present scale out KV stores with a focus on predictable performance and availability. Multiple local node KV stores are used underneath the scale out framework and these are expected to provide `get`, `put`, and `delete` operations. NVMKV complements these efforts by providing a lightweight, ACID compliant, and high-performance, single-node KV store.

Most flash-optimized KV stores use a log structure on block-based flash devices [10, 14, 17, 18, 21, 31]. FAWN-KV [14] focused on power-optimized nodes and uses an in-memory map for locating KV pairs at they rotate through the log. FlashStore [21] and SkimpyStash [22] take similar logging-based approaches to provide high-performance updates to flash by maintaining an in-memory map. SILT [31] provides a highly memory optimized multi-layer KV store, where data transitions between several intermediate stores with increasing compaction as the data ages. Unlike the above mentioned systems, NVMKV eliminates an entire additional layer of mapping along with in-memory metadata management by utilizing the FTL mapping infrastructure.

There are several popular disk optimized KV stores [3, 8, 26]. Memcachedb [3] provides a persistent back end to the in-memory KV store, memcached [2], using BerkeleyDB [34]. BerkeleyDB, built to operate on top a black-box block layer, caches portions of the KV map in DRAM to conserve memory and incurs read amplification on map lookup misses. MongoDB, a cross-platform document-oriented database, and LevelDB, a write-optimized KV store, are HDD based KV stores. Disk-based object stores can also provide KV capabilities [28, 30, 32, 36]. Disk-based solutions do not work well on flash because the significant AWA that they induce reduces the flash device lifetime by orders of magnitude [33].

Finally, we examine the role of consistency in KV stores in the literature. Anderson *et al.* analyze the consistency provided by different KV stores [15]. They observe that while many KV stores offer better performance by providing a weaker form of (eventual) consistency, user dissatisfaction when violations do occur is a concern. Thus, while many distributed KV stores provide eventual consistency, others have focused on strong transactional consistency [38]. NVMKV is a unique KV store that leverages the advanced capabilities of modern FTLs to offer strong consistency guarantees and high-performance simultaneously.

11 Conclusions

Leveraging powerful FTL primitives provided by a flash device allows for rapid and stable code development; application developers can exploit features present in the FTL instead of re-implementing their own mechanisms. NVMKV serves as an example of leveraging and enhancing capabilities of an FTL to build simple, lightweight but highly powerful applications. Through the NVMKV design and implementation, we demonstrated the impact to a KV store in terms of code and programming simplicity and the resulting scalable performance that comes from cooperative interaction between the application and the FTL. We believe that the usefulness of primitives for FTLs will only grow. In time, such primitives will fundamentally simplify applications by enabling developers to quickly create simple but powerful, feature-rich applications with performance comparable to raw devices.

Acknowledgments

We thank the many developers of NVMKV, an open source project [12]. We would like to thank Akshita Jain for her help with our experimental setup. And we thank our shepherd, Haryadi Gunawi and the anonymous reviewers for their insightful feedback.

References

[1] Aerospike: High performance KV Store use cases. http://www.aerospike.com/.

[2] memcached. http://memcached.org/.

[3] memcachedb. http://memcachedb.org/.

[4] Native Flash Support for Applications. http://www.flashmemorysummit.com/.

[5] FNV1a Hash. http://www.isthe.com/chongo/tech-/comp/fnv/index.html, 1991.

[6] SBC-4 SPC-5 Atomic writes and reads. http://www.t10.org/cgi-bin/ac.pl?t=d&f=14-043r2.pdf, 2013.

[7] MemSQL. http://www.memsql.com, 2014.

[8] MongoDB. http://mongodb.org, 2014.

[9] NVM Primitives Library. http://opennvm.github.io/, 2014.

[10] RocksDB. http://rocksdb.org, 2014.

[11] What is SAP HANA? http://www.saphana.com/community/about-hana, 2014.

[12] NVMKV. https://github.com/opennvm/nvmkv, 2015.

[13] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. of ACM SOSP*, 2009.

[14] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. Technical report, 2009.

[15] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What Consistency Does Your Key-Value Store Actually Provide? october 2010.

[16] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proc. of the USENIX NSDI*, 2011.

[17] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proc. of the USENIX NSDI*, 2009.

[18] Mateusz. Berezecki, Eitan. Frachtenberg, Michael. Paleczny, and Kenneth Steele. Many-Core Key-Value Store. In *Green Computing Conference and Workshops (IGCC)*, july 2011.

[19] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. June 2010.

[20] Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. NVM Compression—Hybrid Flash-Aware Application Level Compression. In *Proc. of the USENIX INFLOW)*, October 2014.

[21] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. of VLDB*, September 2010.

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proc. of the ACM SIGMOD*, 2011.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *Proc. of the ACM SIGOPS*, October 2007.

[24] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proc. of the ACM SIGMOD*, 2013.

[25] Fusion-io, Inc. ioMemory Virtual Storage Layer (VSL). http://www.fusionio.com/overviews/vsl-technical-overview.

[26] Sanjay Ghemawat and Jeff Dean. LevelDB. https://code.google.com/p/leveldb/, 2011.

[27] Laura M. Grupp, John D. Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *Proc. of the USENIX FAST*, 2012.

[28] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software Persistent Memory. In *Proc. of USENIX ATC*, 2012.

[29] William Josephson, Lars Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. In *Proc. of the USENIX FAST*, February 2012.

[30] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Commun. ACM*, 34:50–63, October 1991.

[31] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of the ACM SOSP*, October 2011.

[32] Barbara Liskov, Atul Adya, Miguel Castro, Sanjay Ghemawat, Umesh Gruber, Robert a nd Maheshwari, Andrew C. Myers, Mark Day, and Liuba Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of ACM SIGMOD*, 1996.

[33] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *Proc. of the USENIX HotStorage*, June 2014.

[34] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *Proc. of the USENIX ATC*, 1999.

[35] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/o: Rethinking Traditional Storage Primitives. In *High Performance Computer Architecture*, Feb 2011.

[36] Mahadev Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steer, and James J. Kistler. Lightweight Recoverable Virtual Memory. *Proc. of ACM SOSP*, December 1993.

[37] Mohit Saxena, Michael Swift, and Yiying Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *European Conference on Computer Systems*, April 2012.

[38] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. In *Proc. of the ACM SIGPLAN workshop on ERLANG*, 2008.

[39] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proc. of the USENIX FAST*, 2012.

[40] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proc. of the ACM SoCC*, 2012.

[41] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proc. of the ACM Symposium on Cloud Computing*, 2012.

[42] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *Proc. of the USENIX INFLOW*, October 2014.

[43] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proc. of the SYSTOR*, 2013.

Lightweight Application-Level Crash Consistency on Transactional Flash Storage

Changwoo Min[†*] Woon-Hak Kang[‡] Taesoo Kim[†] Sang-Won Lee[‡] Young Ik Eom[‡]
[†]*Georgia Institute of Technology* [‡]*Sungkyunkwan University*

Abstract

Applications implement their own update protocols to ensure consistency of data on the file system. However, since current file systems provide only a preliminary ordering guarantee, notably `fsync()`, these update protocols become complex, slow, and error-prone.

We present a new file system, CFS, that supports a native interface for applications to maintain crash consistency of their data. Using CFS, applications can achieve crash consistency of data by declaring code regions that must operate atomically. By utilizing transactional flash storage (SSD/X-FTL), CFS implement a lightweight mechanism for crash consistency. Without using any heavyweight mechanisms based on redundant writes and ordering, CFS can atomically write multiple data pages and their relevant metadata to storage.

We made three technical contributions to develop a crash consistency interface with SSD/X-FTL in CFS: selective atomic propagation of dirty pages, in-memory metadata logging, and delayed deallocation. Our evaluation of five real-world applications shows that CFS-based applications significantly outperform ordering versions: 2–5× faster by reducing disk writes 1.9–4.1× and disk cache flushing 1.1–17.6×. Importantly, our porting effort is minimal: CFS requires 317 lines of modifications from 3.5 million lines of ported applications.

1 Introduction

Preserving the consistency of application data is one of the foremost responsibilities of computer systems. Applications, ranging from a simple text editor to more complex relational DBMS, are designed to keep their data crash-consistent. Nevertheless, due to limited file system interfaces, primarily `fsync()`, update protocols of applications to achieve crash consistency are notoriously complex, inefficient, and ad-hoc. As a result, most applications still incur inconsistencies of data upon system crashes or random power failures [54].

Suppose that two database files need to be atomically updated as shown in Figure 1. In current file systems, a typical solution is to use multiple rollback journals as shown in Figure 2. To make a single database update crash-consistent, it first records the original state of the database to a journal, so that it can always restore the database to known state upon a system crash. To make multiple database files crash-consistent, it has to maintain another journal, the so-called master journal, which specifies the database files involved during updates.

```
1  + cfs_begin();
2    write(/db1, "new");
3    write(/db2, "new");
4  + cfs_commit();
```

Figure 1: An example code snippet to implement crash-consistent updates of two database files in SQLite by using CFS. In this pseudo code, `/db1` means a file descriptor of a database `db1` under the directory `/`, and "new" means new data (e.g., database entry) to be updated. Two API calls, `cfs_begin()` and `cfs_commit()`, are included at the beginning and end of two `write()` operations to denote an atomic update.

As a result, popular database systems, such as SQLite [7], end up maintaining three journal files and performing 11 `fsync()` operations for updating just two database files with crash-consistency [31]. Besides complexity, this ad-hoc update protocol imposes a huge performance overhead: under ext4 in ordered journal mode, it generates 48 page disk writes and eight disk cache flush operations to update just two data pages. In spite of the inherent performance overhead, such `fsync()`-based update protocols often can not guarantee crash consistency. This happens because some file systems, device drivers, and virtual machines deliberately ignore such flush requests to optimize runtime performance [12, 13, 50].

A significant amount of research has been done to provide consistency of file system structures (e.g., metadata) [16, 25, 29, 39, 43, 58, 64]. However, upon crashes or power failures, even when file system structures are consistent in a system-wide manner, each application's data can be left inconsistent.

One reason why file systems and applications resort to costly journaling or logging is that current storage devices do not guarantee the atomic write of multiple pages or even a single page. Though recently proposed transactional flash storage supports atomic multi-page writes [20, 33, 51, 53, 56] by extending their log-structured write mechanism, to the best of our knowledge, there is no study on how to use transactional flash storage for

*Some of this work was performed while Changwoo Min was at Sungkyunkwan University.

```
1  // init master journal
2  open(/master.jnl, O_CREATE);
3  write(/master.jnl, "/db1,/db2");
4  fsync(/master.jnl);  fsync(/);
5  // update db1
6  open(/db1.jnl, O_CREATE);
7  write(/db1.jnl, "old");
8  fsync(/db1.jnl);     fsync(/);
9  write(/db1.jnl, "/master.jnl");
10 fsync(/db1.jnl);
11 write(/db1, "new");
12 fsync(/db1);
13 // update db2: do the same as db1
14    ...
15 // clean up master journal
16 unlink(/master.jnl);
17 fsync(/);
18 // clean up db1/2 journal
19 unlink(/db1.jnl);
20 unlink(/db2.jnl);
```

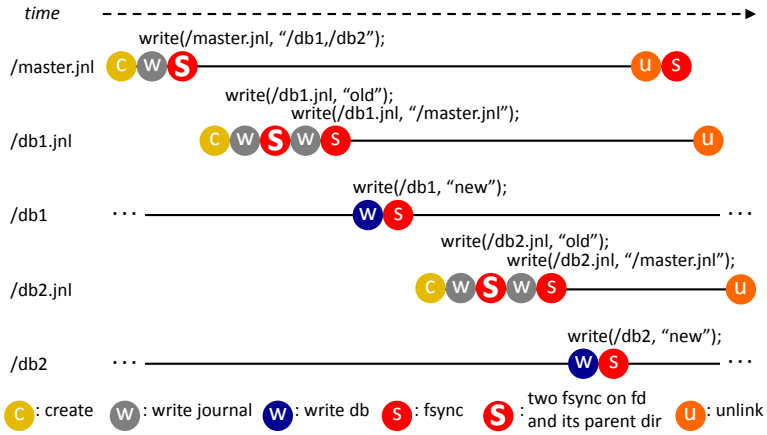


Figure 2: Crash-consistent updates of two database files in SQLite. File systems provide a minimal consistency guarantee upon a crash or a power failure: data and metadata of the latest sync-ed files will be preserved in order. To provide application-level consistency, an application has to carefully coordinate `fsync()` and `unlink()` in an ad-hoc manner. In this example, SQLite maintains a master journal (`master.jnl`) and two journals (i.e., `db1.jnl` and `db2.jnl`) with complex ordering of `fsync()` and `unlink()` calls. Because `fsync()` does not ensure that the entry in the directory containing the file has reached storage, creating a journal file entails two `fsync()` calls: one for the journal itself and another for its parent directory.

direct and efficient support of application-level crash consistency.

In this paper, we present CFS, a file system that natively supports application-level crash consistency on transactional flash storage. We make the following technical contributions:

- **Native Interface for Crash Consistency:** CFS provides a native interface for applications to describe *atomic code regions* which must operate atomically. For each region, CFS creates an *atomic propagation group*, which is a set of data and metadata pages modified in the region. An atomic propagation group is atomically written to storage regardless of system crash (*commit*), or is reverted either explicitly by a user (*abort*) or upon crash. Atomicity is guaranteed by atomic multi-page writes in transactional flash storage without using any journaling or logging.
- **In-Memory Metadata Logging:** The key design challenge is how to propagate metadata pages shared by multiple atomic propagation groups. Suppose that two inodes in different groups happen to be in the same metadata page; committing a group results in unintended propagation of another inode. We call this *false sharing of metadata pages* because irrelevant sub-page-sized metadata structures are in the same page. To resolve this, we introduce an *in-memory metadata logging* mechanism that keeps track of metadata changes for each group and selectively propagates changes of a committing group.
- **Delayed Deallocation:** All operations in an atomic code region must be safely abortable. However, it is tricky to revert deallocation of file system resources such as inodes and blocks. For example, suppose that block `B1`, which was released from atomic propagation group `A1`, was allocated to another group. If `A1` is aborted, it is impossible to revert `A1`'s deallocation of `B1`. To resolve this, we introduce a *delayed deallocation* technique, which defers actual resource deallocation until commit time.
- **Legacy Application Support:** CFS internally manages a system-wide atomic propagation group for legacy applications that do not use CFS system calls. Legacy applications can run with CFS-based ones without any modification.

Unlike transactional file systems [36, 42, 55, 60, 63] , which have been proposed to support DBMS-like ACID transactions, we have designed CFS primarily for crash consistency, and *not* for strong isolation. There are two reasons for this decision. First, modern applications like MariaDB [3, 38] or Kyoto Cabinet [2] already relax their isolation level for performance optimization, without compromising their correctness semantics. If strong isolation is enforced by transactional file systems, atomic updates in these applications may fail to progress efficiently due to frequent conflicts among transactions. We will discuss this in more detail in §8. Second, using rich semantics at the application level, it makes sense to give developers more freedom to choose appropriate synchronization primitives to achieve the required isolation level for their applications. We summarize isolation policies of popular applications in §6.

We have implemented CFS based on ext4 using transactional flash storage (SSD/X-FTL [33]). Our evaluation on real-world applications, including SQLite, MariaDB, and Kyoto Cabinet shows that CFS-based applications significantly outperform their original versions: 2–5× faster by

reducing disk writes 1.9–4.1× and disk cache flushing by 1.1–17.6×. Our porting experience shows that CFS can easily replace a variety of existing update protocols with its native interface.

In the rest of this paper, we will discuss the background (§2) and design principles (§3). Next, we will present the details of CFS design (§4) and implementation (§5). Then, we will show our application case studies (§6) and evaluate performance (§7). Finally, we discuss our limitations (§8) and related work (§9), and conclude (§10).

2 Background

2.1 Problems in Modern File Systems

Modern file systems cannot guarantee application-level crash consistency even with data journal mode for the four following reasons:

No Atomicity on Multiple Files: It is not uncommon that application data, which needs to be atomically updated, spans two or more files, as shown in Figure 2. However, while modern file systems provide three ordering primitives, namely `sync()`, `fsync()` and `msync()` for system-wide or file-wide flushing, they do not provide any primitive to flush multiple files selectively and atomically. As a result, developers have no option other than to implement their own complex update protocols with the given primitives.

Lack of Atomic Page Writes: Although `fsync()` ensures durability and ordering of writes, current storage devices do not guarantee the atomic write of a single page as well as multiple pages. Hence, file systems and applications resort to costly *journaling (or logging)* mechanisms or complicated *copy-on-write (CoW)* mechanisms [19]. However, disk cache flush operations, which are essential to implement such mechanisms, frequently become a performance bottleneck [18, 19, 46].

Shared Metadata Page: Even if an underlying storage device provides atomic page writes, modern file systems cannot directly support application-level crash consistency. Assume that two applications, `A1` and `A2`, are running. Suppose that `A1` finishes atomic updates of its changes and the system crashes before `A2` triggers its atomic updates. To achieve crash consistency of `A1`, all data pages and relevant metadata pages should be written. However, a metadata page can contain information of both `A1` and `A2`, so the *incomplete* metadata changes of `A2` can be accidentally propagated by `A1`. This is because a write unit in storage is a page, not an individual metadata structure. We call this *false sharing of metadata pages*. Depending on the unwanted metadata propagation of `A2`, a directory could have nonexistent files, or a file could have garbage blocks, or the free block counter in a superblock could be incorrect. In other words, `A1` hampers the consistency of `A2` upon a crash.

Steal Policy and Lack of Undo Mechanism: Modern file systems use the *steal policy*: due to page reclamation by the page flusher or `sync()` by applications, any metadata or data page can be written to storage at any time, although its corresponding application is still executing. Upon system recovery after a crash or an application's request to abort its changes, the stolen pages and in-memory data structures, such as metadata and inode cache, should be reverted. Unfortunately no existing file system provides a native undo mechanism. This is one of the main reasons why file systems cannot natively support application-level crash consistency.

2.2 Transactional Flash Storage

Transactional flash storage [20, 33, 51, 53, 56] supports atomic write of multiple pages by extending the log-structured nature of a *flash translation layer* (FTL). They defer the update of the mapping table for new data and achieve the atomicity of multi-page writes by atomically updating the mapping table in response to a commit request from the host. Since atomic writes achieve a high level of data integrity with fewer write commands, the storage industry is working on its standardization [61].

In this paper, we used SSD/X-FTL [33], which is a transactional flash storage providing extended SCSI interfaces such as `write(txid, page)`, `commit(txid)`, and `abort(txid)`. Each write operation is associated with `txid`, and the written pages with the same `txid` become atomically durable upon a `commit(txid)` request. Upon an `abort(txid)` request, they are reverted to their old copies. Though CFS is built on SSD/X-FTL, CFS does not fundamentally require SSD/X-FTL, and it can be built on any transactional storage devices [26, 51, 56, 61] (see §8).

3 Design Principles

For an application to be crash-consistent, a series of file system operations either *all* occur, or *nothing* occurs. From the perspective of file systems, this can be translated into the following technical axiom: "all data pages and their relevant metadata changes should be *atomically* propagated to storage." In this paper, file systems satisfying this technical requirement will be said to provide *application-level crash consistency*. In this section, we discuss four design principles which will lead to our key techniques: selective atomic propagation of dirty pages (§4.1) and in-memory metadata logging (§4.2).

Defining an Atomic Code Region: In CFS, instead of implementing complex update protocols, applications simply specify an *atomic code region*, in which file system operations must be atomically processed. An atomic code region starts with `cfs_begin()` and ends

with `cfs_commit()` (or is canceled with `cfs_abort()`). CFS automatically captures files modified by system calls, but for memory-mapped files, developers should explicitly specify corresponding file descriptors by using `cfs_add(fd)`. Naturally, there are one or more files in an atomic code region. After capturing all the modifications inside the atomic code region (the so-called *atomic propagation group*), `cfs_commit()` will make those changes persistent, and `cfs_abort()` will revert them by undoing all operations performed in the atomic code region.

Atomic Propagation of Data and Metadata Pages: Instead of resorting to costly journaling or complex CoW mechanisms, CFS exploits the atomic multi-page write feature of transactional flash storage. All data and relevant metadata pages modified in an atomic code region are grouped and sent to storage for an atomic write. Since transactional flash storage guarantees atomic durability, there is no need for journaling or CoW mechanisms.

No-Steal and Selective Propagation of Metadata: Even if CFS writes only relevant metadata pages modified in an atomic code region, metadata changes made by other in-progress atomic code regions can be propagated to storage due to the *false sharing of metadata pages*.

To avoid this anomaly, CFS delays writing the in-progress metadata changes to storage until commit time (*no stealing*). Also, to selectively propagate the changes in a metadata page, we propose a technique that logs in-memory metadata changes for each atomic propagation group and replays them at commit time.

Undoing Stolen Data Pages and In-Memory Structures: Unlike metadata pages, we support a *steal policy* for data pages, meaning that data pages do not need to be sent to storage, and rather can be stolen. This provides two benefits. First, effective management of limited page cache becomes possible, because data pages can be reclaimed under memory pressure. Second, the amount of writes at commit can be reduced, hence latency as well, because the page flusher can flush dirty pages during idle time.

To support a steal policy of data pages, CFS should be able to revert every stolen page of the aborted atomic propagation group to its old copy when system recovers from a crash or `cfs_abort()` is invoked. CFS relies on transactional flash storage to revert stolen pages. When a system crashes, transactional flash storage reverts all uncommitted writes to their old copies on system reboot. When `cfs_abort()` is called, CFS asks the transactional flash storage to revert written pages of the aborting group to their old copies. In addition, CFS reverts all in-memory metadata changes for the operating system after the abort operation. This is done by undoing the collected logs of the in-memory metadata structures.

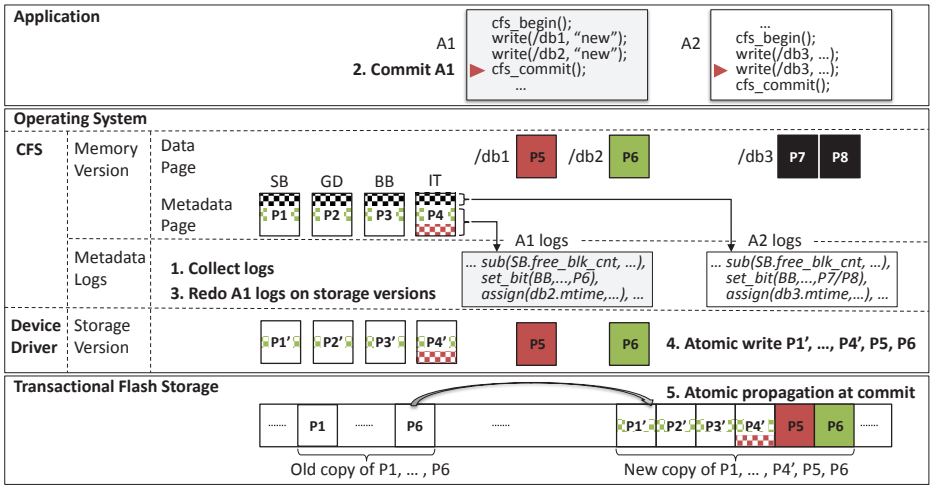
4 CFS Design

In this section, we present the design of CFS, an ext4-based file system that natively supports application-level crash consistency on transactional flash storage. To achieve crash consistency for a series of file system operations, developers define *atomic code regions* in source code and CFS guarantees atomic operations within the regions. For each region, CFS manages an *atomic propagation group* that is a set of data and metadata pages modified in each region. The pages in the group will be written atomically using atomic multi-page write features of transactional flash storage. CFS logs in-memory metadata changes made by each region. At commit time, CFS replays the collected logs of a committing group to selectively propagate changes made in the group to storage. To this end, CFS manages two versions of metadata: *memory version* and *storage version*. The data pages and the storage version of metadata pages are atomically written to storage.

Figure 3 illustrates a running example of CFS. Application `A1`, as in Figure 1, updates two database files, `db1` and `db2`, in its atomic code region. Application `A2` updates two pages of a database file `db3`. Upon `cfs_begin()`, a new atomic propagation group starts and an associated new `txid` is assigned for further interaction with transactional flash storage. CFS logs in-memory metadata changes made in each region (Step 1). Suppose that `A1` starts `cfs_commit()` while `A2` is still in-progress (Step 2). At this moment, the atomic propagation group of `A1` has data page `P5` and `P6`, and metadata pages `P1`–`P4`, which have metadata changes for `db1` and `db2`. The false sharing between `A1` and `A2` occurs in `P1`–`P4`. CFS replays the collected logs of `A1` on the storage versions, `P1'`–`P4'`, for selective propagation of metadata changes (Step 3). Thus, the storage versions `P1'`–`P4'` only contain the changes of `A1` without incorporating the changes of still-in-progress `A2`. All data pages and the storage version of metadata pages in the group are written to storage with the `txid` of `A1` (Step 4). Finally, CFS asks the storage device to make the written pages with the `txid` atomically persistent (Step 5). By using transactional flash storage, CFS can avoid redundant journalings and can significantly improve the performance of file systems. Not only that applications do not need complex and ad-hoc update protocols, but naturally gain better performance by using CFS. For example, SQLite's update protocols, which invokes 11 `fsync()` calls for updating two database files, can be replaced with two native calls to CFS and gain a 16.7× increase in performance compared to the original version, as shown in §7.2.

4.1 Managing Atomic Propagation Groups

For each atomic code region embraced with `cfs_begin()` and `cfs_commit()`, CFS keeps track of modified pages



NOTE. SB: superblock, GD: group descriptor, BB: block bitmap, IT: inode table

Figure 3: Two database applications running in CFS. Checkerboard rectangles denote metadata of files with the same color. Redoing logs of `A1` resolves the false sharing that occurred at `P1`–`P4` and creates storage version `P1'`–`P4'`. Only storage versions of metadata pages with no false sharing are written to storage. Thus, committing `A1` does not interfere with the consistency of `A2`.

as an atomic propagation group. All dirty data pages of the files and modified metadata pages in the same atomic propagation group are atomically and persistently written to storage using atomic write operations in transactional flash storage. An atomic propagation group is inherited by a child task but cannot be nested: a new task spawned inside an atomic code region automatically inherits its parent region (i.e., parent's `txid`) until it starts a new atomic code region. In SSD/X-FTL, a new `txid` is assigned to a task (i.e., `task_struct` in Linux) upon invocation of `cfs_begin()`. CFS then writes all pages with the same `txid` for the task.

4.2 In-Memory Metadata Logging

Memory Version vs. Storage Version: At the core of CFS is the in-memory metadata logging. CFS records changes of in-memory metadata structures, called the *memory version*, for each atomic propagation group. Upon a commit, CFS selectively propagates the changes made in the group to on-disk metadata structures, called the *storage version*, which are updated by *redoing* logs of a committing group, and then writing to storage. Upon an abort, CFS reverts the changes of in-memory metadata structures by *undoing* logs of the aborting group. Creating the storage version is straightforward if a metadata has separate in-memory and on-disk structures (e.g., superblock and inode). Otherwise, in the case that there is no separate structure (e.g., inode bitmap), CFS clones a memory version and uses the cloned structure as a storage version. The storage versions are what will be initially loaded when reading pages from storage.

Operational Logging: CFS uses operational logging, which records executed metadata change operations. Table 1 shows the specification of operations used to capture the metadata changes in CFS. Operations are composed of four primitive operations and one extended operation (`x_op`). Primitive operations directly modify metadata structures and an extended operation runs a registered callback function, noted as argument `f`. All operations have two arguments: `m` for memory version and `s` for storage version of a metadata structure.

Let us suppose that the free inode count in a superblock needs to be decremented when allocating a new inode. To capture this operation, CFS records a `sub` operation with an argument of the free inode count in superblock, so that the metadata change can be part of the logs in the current atomic propagation group. Upon a commit, the free inode count in on-disk superblock (i.e., `s`) will be decremented to reflect the change (*redoing*). Upon an abort, the free inode count of the in-memory superblock (i.e., `m`) will be incremented (*undoing*) to revert the change.

It is worth detailing how to undo `assign` operations (ϕ(`m`) in Table 1). For example, atomic propagation group `A1` creates a new file, thus the timestamp of the parent directory `D` is updated from `t0` to `t1`. After that, another atomic propagation group `A2` creates another file at the same directory so the timestamp is updated from `t1` to `t2`. Now, if `A1` aborts, to what should the timestamp of `D` be reverted? Since `A2` already updated the timestamp from `t1` to `t2`, it should remain `t2`. After that, when `A2` aborts, the timestamp should be reverted to `t0`. After all, CFS always reverts to *its most recent valid value*. For this purpose, CFS maintains a list of `assign` operations for a data entry in order of the operations. Upon an abort, CFS removes the aborting `assign` operation in the list and reverts the value to the head of the list (i.e., its most recent valid value).

Operation	REDO (commit)	UNDO (abort)	Description (example)
add(m,s,v)	s+=v	m-=v	Add v to m (e.g., increments free inode counts)
sub(m,s,v)	s-=v	m+=v	Subtract v from m (e.g., decrements free block counts)
assign(m,s,v)	s=v	m=ϕ(m)	Assign v to m (e.g., changes access mode of an inode)
toggle_bit(m,s,i)	s[i]=¬s[i]	s[i]=¬s[i]	Toggle i-th bit of m (e.g., toggles a bit in block bitmap)
x_op(m,s,f,a)	f(commit,m,s,a)	f(abort,m,s,a)	Run a function f (e.g., allocates a directory entry)

NOTE. m: memory version, s: storage version, m[i]: i-th bit of m, ϕ(m): the most recent valid value of m

Table 1: Specification of operations for in-memory metadata logging.

Extended Operations: To handle complex metadata structures and optimize the use of resources (e.g., caches), CFS introduced a special type of operation, called x_op(). We summarize its usage into three categories:

The first usage is to manipulate complex metadata structures. For example, each directory keeps its directory entries (or dentries) in a list or a hash tree [24]. Inserting or deleting a dentry must follow the semantics of such structures. When CFS allocates a dentry to create a new file, it registers a callback function. Upon a commit, the callback inserts the dentry into the storage version of the directory. Upon an abort, it deletes the dentry from the memory version of the directory.

Second, the extended operation is required to revert file system caches upon an abort. CFS maintains the inodes and dentry caches for efficient accesses as well as the buddy cache [17] for efficient disk block allocation. When allocating a file system resource (i.e., inode, dentry, or block), an associated cache is also updated. CFS needs to revert the changes in the cache if the resource allocation is aborted. To do this, CFS has to register a callback that reverts the cache updates that happened while allocating file system resources.

Lastly, the extended operation is required to correctly deallocate file system resources. For instance, given two atomic propagation groups A1 and A2, let us suppose that a block released from A1 was allocated to A2. After A2 is committed, it is impossible to abort A1 because there is no way to revert the block allocation of already committed A2. In order to prevent this scenario, we propose a technique named *delayed deallocation*. When CFS needs to release file system resources, it registers a callback function. Deallocation is deferred until the actual commit, at which point it finally deallocates the resource by executing the registered callback function.

A Running Example: As in Figure 3, suppose that db1, db2, and db3 are in the same block group [24]. If new data is overwritten in db1, and another new data is appended in db2 and db3, then the size of their database files grows. The last modified time of each database file is updated (P4) and CFS logs three assign operations. Growing the files incurs a series of metadata changes: three block use flags for P6, P7, and P8 in a block bitmap (P3) are turned on and CFS logs three toggle_bit operations; the block maps in the inode table (P4) are changed to refer to the new blocks and the file sizes in the inodes (P4) increase, thus CFS logs five assign operations; each free block count in the superblock (P1) and block group descriptor (P2) decreases, thus CFS logs two sub operations. Since block allocation incurs the changes in the buddy cache, CFS adds one x_op to revert the change in the cache upon abort.

4.3 Commit and Abort Procedures

Upon a commit, CFS first writes all dirty data pages of the files that belong to the committing atomic propagation group. Writing data pages could cause further metadata changes; for example, due to the delayed block allocation scheme [17], the actual block allocation happens when writing data blocks, changing metadata structures such as block bitmap and free block count. Then, CFS applies all of the group's collected logs to the storage version of metadata in the order of their generations and writes the storage version. It writes all pages with the txid issued at cfs_begin() and then asks SSD/X-FTL to make written pages with the txid durable.

Upon an abort, CFS rolls back the atomic propagation group by executing all the collected logs for the group in reverse order of their creation (*undoing*). Then, it also lets the storage revert the stolen data pages to their old copies. In SSD/X-FTL, CFS sends an abort command with the txid of the group. Finally, CFS forcefully drops all the dirty pages of the files in the group so that subsequent access to the page results in reading the reverted valid page from storage. If another application happens to access the aborted files, it could encounter an error depending on its correctness semantics. If this is the case, access to shared files must be coordinated using a synchronization primitive such as locking, or the shared files must be made public only after they are committed. For example, a transactional package manager needs to make new versions of shared libraries public after successful package installation to avoid applications reading the libraries, which are being installed and could be subject to an abort.

4.4 Dealing with Legacy Applications

It is highly desirable to be able to run the legacy applications without any modification while preserving their semantics. To this end, every update from legacy applications is treated as part of an atomic propagation group

in CFS. To be concrete, CFS maintains a *system-wide atomic propagation group*, to which every update from legacy applications belongs. CFS commits the system-wide atomic propagation group either when background flusher threads flush all dirty data and metadata pages or when a `sync()` is invoked. After the commit, CFS creates a new system-wide atomic propagation group for handling subsequent updates from legacy applications. Our current unoptimized `fsync()` simply performs `sync()`. We believe, however, this is not a fundamental limitation of our approach; for example, managing fine-grained (e.g., file-level) atomic propagation group and using group commit can be leveraged to optimize `fsync()` of legacy applications.

4.5 Consistency and Recovery

Despite various system or application failures, CFS guarantees application-level crash consistency as long as an application correctly specifies atomic code regions and a transactional flash storage guarantees atomic multi-page writes. Because CFS enforces durability of *all and only* the data pages and metadata changes of a committing atomic code region, it guarantees version consistency [19], that the metadata version matches the version of the referred data for each commit operation, and does not interfere with the consistency of other commit operations. Also, because updates from legacy applications are treated in the same manner, CFS guarantees file system-level crash consistency.

There are two types of common failures in CFS. First, if an application is terminated abnormally without the entire system failing, the OS kernel aborts all uncommitted atomic propagation groups of the terminating process and thus rolls back the changes of the application. To maintain the semantics of legacy applications, CFS never aborts the system-wide atomic propagation group. Second, if the entire system fails (e.g., a power outage), CFS relies on the recovery mechanism of transactional flash storage. On system reboot, for any incomplete commit at the time of failure, transactional flash storage will invalidate all uncommitted changes and thus roll back the storage to the last successful commit state.

5 Implementation

We implemented CFS in Linux Kernel 3.10.7 based on ext4, modifying about 5,800 lines of code. To capture the operational logs at runtime, we inserted 171 primitive operations and 11 extended operations. We performed experiments on a machine with a quad-core 2.1 GHz Intel Xeon E5606 processor and 4 GB memory. We used the OpenSSD development platform [10] with 8 GB storage capacity and the SATA 2 interface. We implemented two FTL schemes on the OpenSSD device: greedy FTL [35], which is a page-level FTL scheme with a greedy garbage collection policy, and X-FTL [33], which is an extended greedy FTL, to support atomic multi-page writes.

Application	Isolation
SQLite	Strong isolation
MariaDB	Four isolation levels in SQL standards [38]
Kyoto Cabinet	Intentionally no isolation
APT	No isolation
vim	Strong isolation or no isolation

Table 2: Isolation levels in five real-world applications.

In comparison to commercial SSDs, OpenSSD and its FTLs have several limitations: First, its capacity is too small to be considered as typical enterprise setting. Since SSDs use log-structured writing scheme, write performance under high disk utilization would be slower than that in low disk utilization. To avoid such performance anomaly and present fair comparison, we carefully choose the data set size for evaluation. For MariaDB, database size was set to 2.5 GB so there were around 70% free space in the SSD. Next, OpenSSD has a low degree of internal parallelism due to its architectural limitations. Due to this low degree of internal parallelism, performance degradation caused by a disk cache flush is limited in OpenSSD, even though it will be significant in high-end SSDs [32]. Finally, the size of atomic propagation group is limited by the transaction size of X-FTL. However, this limitation could be overcome by adopting other transaction representation schemes (e.g., cyclic representation [56]) to support unlimited (i.e., limited by only disk capacity) transaction size.

6 Application Case Studies

In this section, we show how CFS can simplify the complicated update protocols of existing applications. We choose five real-world applications, which have a variety of isolation levels from strong isolation (e.g., SQLite) to no isolation (e.g., KyotoCabinet), shown in Table 2. The CFS-enabled applications can simply reuse the existing concurrency control code to achieve the same isolation level without any additional overheads. As summarized in Table 3, porting existing applications to CFS is straightforward. For four applications, in which a file is the granularity of atomicity, we simply specified atomic code regions using CFS's native calls. For MariaDB, which uses physiological write-ahead-logging [44] and double-write [1], we replaced the double-write with CFS-protected atomic write of database files. Our experience confirms that CFS can easily replace various existing update protocols: for five real-world applications, we only needed to modify 317 lines of code out of 3.5 million in total.

SQLite: SQLite [6] is a library based DBMS widely used in smart devices. It relies on rollback journaling (RBJ) or write-ahead-logging (WAL) to guarantee crash consistency [7, 8]. In RBJ mode, the original content of a

Application	Lines of code		Mechanisms to guarantee application consistency	
	Original	Modified	Original	Modified
SQLite	217,313	38	Physical RBJ or WAL	Specifying an atomic code region
MariaDB	1,534,980	240	Physiological WAL & double-write	Physiological WAL & atomic database write
Kyoto Cabinet	162,606	26	Physical RBJ on a `mmap`-ed region	Specifying an atomic code region
APT	407,642	4	None	Specifying an atomic code region
vim	1,179,246	9	`rename`-based update	Specifying an atomic code region
Total	**3,501,787**	**317**		

NOTE. RBJ (rollback journaling), WAL (write-ahead-logging)

Table 3: Summary of our porting efforts in applying CFS to five real-world applications. CFS requires only 317 lines of modifications out of 3.5 million lines of ported applications, in order to support the crash consistency.

page is copied to the rollback journal before updating the page. In the WAL mode, the original content is preserved in the database and the modified page is appended to a write-ahead-log file. The change is then later propagated to the database by periodic checkpointing.

In version 3.8.3 of SQLite, the RBJ and WAL mode consist of about 14,500 lines of code. With CFS, we were able to implement the same level of crash consistency by adding just 38 lines of CFS system calls with journal mode off.

MariaDB: MariaDB [3] is a popular open source DBMS, and InnoDB is a popular transactional storage engine used in MariaDB. To preserve crash consistency, InnoDB uses an optimized logging technique known as ARIES-style physiological write-ahead-logging (WAL) [44]. Unlike the *physical WAL* mode in SQLite, in the *physiological WAL* of MariaDB, only changes made to data pages are written to the log device to minimize the amount of log writes. Since logs are directly applied to the data pages in-place, crash recovery is possible only if the data pages are not corrupted. To guarantee atomic update of data pages, InnoDB uses a redundant page write technique known as *double-write* [1]: it first synchronously writes data pages to the dedicated double-write area, then re-writes each page to its original location. In each step, `fsync()` calls are used to enforce ordering and durability.

CFS-based MariaDB directly updates database files in-place after writing the physiological log, and does not require the double-write. CFS can ensure the atomic updates of database files by simply guarding the update code using the CFS system calls.

Kyoto Cabinet: Kyoto Cabinet [2] is a library-based key-value store using a memory mapped region to manage its data. For crash recovery, it writes an unmodified copy to the dedicated *rollback journal* area when a data page becomes dirty. To guarantee that the old copy is flushed to storage ahead of its new copy, it calls `fsync()` upon every write to the rollback journal area.

We were able to achieve the same level of crash consistency by simply turning off the journaling and guarding the atomic update code using the CFS system calls.

APT Package Manager: For a successful software installation or update, numerous files can be created, modified, or deleted, and all these modifications should be carried out atomically. Surprisingly, due to the complexity of guaranteeing atomic package installation, most package managers, including the popular APT [23], do not provide atomic installation, leaving the responsibility to system administrators. We added an atomic installation feature to APT by guarding its package operation code using the CFS system calls.

Vim: When saving an updated file of *document-like data*, many applications, including vim [11], use `rename`-based update schemes: creating a new file and writing the updated document to the new file, then calling `fsync()` on the file to force it to disk, and finally replacing the original with the new one. With CFS, vim is modified to update the file in place and its atomicity is guaranteed by wrapping the update code with `cfs_begin()` and `cfs_commit()`.

7 Evaluation

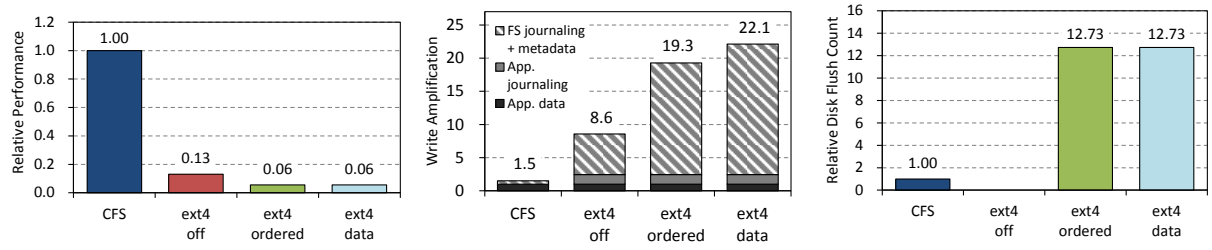
In this section, we present experiments that answer the following questions:

- Does CFS really guarantee application-level crash consistency? (§7.1)
- How do legacy update protocols and CFS-based atomic updates behave differently? (§7.2)
- What are the performance benefits of CFS-based applications? (§7.3)
- What is the performance impact on legacy applications that do not use CFS system calls? (§7.4)

Before running each experiment, we ran the workload independent preconditioning (WIPC) [62], so as to put the SSD in a steady state. The journal size of ext4 is set to 128 MB. We reported the average of three runs.

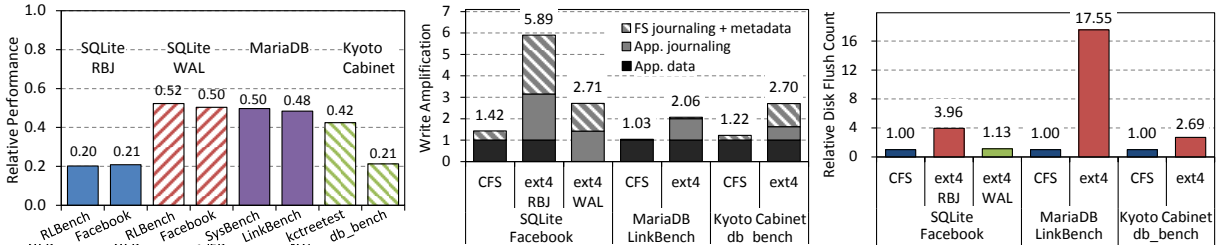
7.1 Consistency over Random Failures

We begin our evaluation of CFS by experimentally verifying whether it preserves application-level crash consistency across sudden power outages. We used MariaDB because it is the most mature and complicated among our test applications and it also provides the tool `mysqlcheck`,



(a) Performance **(b)** Write amplification **(c)** Disk cache flush count

Figure 4: Results of the microbenchmark. Atomic update of two database files in Figure 1 and 2. The CFS-based version is 16.7× faster than the original version in ext4 ordered mode. Because the CFS-based version does not rely on a complex update protocol, disk writes and cache flush operations are reduced by 12.9× and 12.7×, respectively.



(a) Performance **(b)** Write amplification **(c)** Disk cache flush count

Figure 5: Results of real-world applications. The original versions are run on ext4 in ordered journal mode. Compared to the CFS-based versions, the original versions relying on complex update protocols show significant overhead.

which checks for database corruption. While we ran LinkBench [14] on CFS-based MariaDB, we cut the power of the SSD/X-FTL to stress the full system software/hardware stack. After rebooting the test machine, we checked the consistency of CFS and the database using `fsck` and `mysqlcheck`, respectively. If both checks pass, then we conclude that CFS preserves application-level crash consistency. We repeated this test 100 times and passed the consistency check every time. To check the coverage of our test, we further analyzed pages recovered by SSD/X-FTL. In 90% of the tests, SSD/X-FTL recovered 10.3 pages on average. Types of recovered pages were CFS metadata (2.4%), LinkBench data table (95.1%), and InnoDB system table (2.5%). Though it is limited, this experiment is one practical way to validate CFS's correctness. From a theoretical point of view, it is hard to imagine a case where application consistency is vulnerable when data pages and their relevant metadata changes are atomically propagated to the storage.

7.2 Analysis of Atomic Update

To understand the performance characteristics of legacy update protocols and CFS-based atomic updates, we used the atomic update of two database files presented in Figure 1 and 2 as a microbenchmark. The original version was run on ext4 with three different journaling modes: off, ordered, and data journal mode.

In Figure 4, we first present performance comparisons, write amplification, and disk flush count for further analysis. The performance of each original version is normalized to the CFS-based version. Write amplification is the ratio of an application's writing of database files to the file system's writing to storage. It is split into three categories: application data, application journaling (e.g., SQLite RBJ), and file system overhead (i.e., metadata and journaling). We present the normalized disk flush count for CFS. In the case of CFS, we counted commit requests, upon which SSD/X-FTL flushes the disk cache. This data is obtained by instrumenting the microbenchmark and collecting block traces from the host using `blktrace`.

As Figure 4a shows, the CFS-based version significantly outperforms the original version—7.7× to ext4 off mode, and 16.7× to ext4 journaling modes—due to reduced disk writes and disk cache flush operations. The write amplification of the original version is surprisingly high (Figure 4b): 8.6, 19.3, and 22.1 in ext4 off, ordered, and data journal mode, respectively. Application-level journaling incurs significant metadata overhead. When combined with ext4 journaling, the amount of writes is amplified by 2–3× compared to ext4 off mode. As expected, the disk flush count of the original version is very high (Figure 4c). Since ext4 in off mode does not guarantee any consistency, it does not issue any disk cache flush operations. In the other modes, 12.7× more cache flush operations were issued. However, due to the low degree of internal parallelism of OpenSSD [10], performance is largely determined by the write amplification factors rather than disk cache flush count.

7.3 Performance of Real Applications

To see how CFS can improve performance of real-world applications, we evaluated three performance-sensitive applications: SQLite, MariaDB, and Kyoto Cabinet, studied in §6, using six workloads. In Figure 5, we compared performance, write amplification, and disk flush count of each application. As we expected, CFS-based applications significantly improve performance of their original versions, because they prevent the journaling of journal (JoJ) anomaly [31] fundamentally without any consistency compromise. In the rest of this section, we present the performance analysis of each application.

SQLite: We ran two SQL traces [33] collected from running the RL Benchmark [5] and Facebook applications on an Android 4.1.2 Jelly Bean SDK under its typical usage scenario.

The CFS-based version outperforms the original versions with the rollback journal (RBJ) and write-ahead logging (WAL) by approximately five-fold and two-fold, respectively. In RBJ mode, data is always written twice, one for the RBJ file and another for the database file. Moreover, since the RBJ file is created and deleted whenever a new transaction ends, SQLite in RBJ mode has very high file system metadata overhead. As a result, the original version generates 4.1× more writes and 3.9× more disk cache flush operations than the CFS-based version. In WAL mode, the modified data is appended to a WAL file and then the change is propagated to the database file by periodical checkpointing. Since the WAL file is reused by many transactions until the checkpoint occurs, the metadata overhead of SQLite in WAL mode is far lower than that of RBJ mode. As a result, the original version generates about 1.9× more writes and 10% more disk cache flush operations than the CFS-based version.

MariaDB: We used two popular database benchmarks: SysBench [9] and LinkBench [14]. SysBench in an OLTP mode stresses a 2.5 GB database (16 files) with 10 million rows for 10 minutes. LinkBench from Facebook is designed to benchmark performance of database operations with large-scale social graphs. In LinkBench, we ran 80,000 operations for a 2.5 GB database (18 files) after a two minute warm-up. In both experiments, MariaDB was configured to use 100 MB as a buffer pool with eight concurrent threads, and all under `O_DIRECT` I/O mode.

The CFS-based MariaDB performs 2× faster than the original MariaDB. This performance benefit primarily comes from the reduced number of write operations by replacing the double-write with CFS's native interface. While `fsync()` is required for each database file update and every double-write operation in the original MariaDB, the CFS-based MariaDB requires only one disk flush to update all database files. Thus, the CFS-based version invokes 17.6× fewer disk flush operations than the original

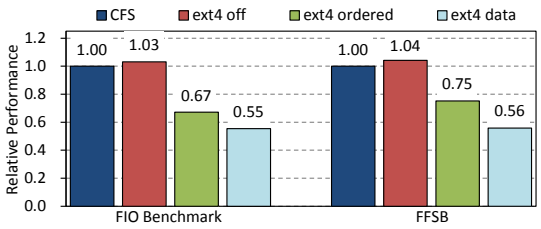


Figure 6: Performance comparison between CFS and ext4 for FIO and FFSB benchmarks

version. However, the performance results in Figure 5a are little affected by such frequent disk flush operations. This is because OpenSSD, as discussed in §5, has low degree of internal parallelism.

To emphasize the criticality of this problem that end-users will face, we ran LinkBench using a commercial SSD [30] with high-level of internal parallelism. Since the SSD does not support transactional interfaces yet, we used the modified MariaDB that flushes disk cache once at updating all database files without double-write. Write amount and cache flush count are the same as those in CFS and so the performance will be similar to that of the CFS-based one. For comparison, we ran the MariaDB only turning off the double-write mode without any other modifications; so the cache flush count is the same as the unmodified MariaDB but the write amount is the same as CFS-based one. The frequent disk cache flush operations in the latter degrades performance by 78%. It shows that frequent cache flush caused by ad-hoc update protocols is a serious performance bottleneck and CFS's native interface can be a solution to overcome this performance degradation problem.

Kyoto Cabinet: We used two workloads for Kyoto Cabinet. First, we ran Kyoto Cabinet's `kctreetest` [2] with eight concurrent test threads. Each test thread writes 10,000 arbitrary key-value pairs and then reads the keys 10,000 times. Second, we ran LevelDB's `db_bench` [28] with a single test thread. We measured the performance of 10,000 arbitrary writes of key-value pairs. We configured Kyoto Cabinet in synchronous transaction mode, guaranteeing consistency.

The CFS-based version significantly outperforms the original version. In `kctreetest`, where the read-to-write ratio is about one, the CFS version is 2.4× faster than the original version. In the write-intensive `db_bench`, the CFS version is 4.8× faster than the original version. The original version issues `sync` system calls three times for a write operation. As a result, it generates 2.2× more writes and 2.7× more disk cache flush operations.

7.4 Performance of Legacy Applications

To understand the performance of legacy applications, which were not designed to use CFS, we compare performances of ext4 and CFS by running two popular bench-

marks: Flexible I/O (FIO) benchmark [15] and Flexible File System Benchmark (FFSB) [59]. We used the FIO benchmark to simulate a data-heavy workload: it is configured to perform random writes to a 4 GB file with an 8 KB write unit while `fsync()` is called every 40 KB. We used the FFSB benchmark to simulate a metadata-heavy workload: it executes a combination of small file creates, writes, reads, and appends.

We ran the benchmarks on CFS and the three journaling modes of ext4. As Figure 6 shows, CFS provides similar performance to ext4 with journaling off while it guarantees the highest level of crash consistency. Results of ext4 in the other journaling modes show significant overhead.

8 Discussion and Future Work

Isolation and Concurrency Control: One may be curious about the difference between the *application-level crash consistency* of CFS and the *transaction* of transactional file systems [36, 42, 55, 60, 63]. In terms of concurrency control, we took an opposite design choice to transactional file systems. Transactional file systems are designed to natively support a DBMS-like ACID transaction, therefore they support strong isolation (i.e., the highest isolation level, *serializable isolation*). Under strong isolation, time-of-check-to-time-of-use (TOCTTOU) races can be easily prevented. In Table 4, we compare two representative transactional file systems, Valor [63] and TxOS [55], with CFS. To support strong isolation, they took two extreme design decisions: Valor uses pessimistic coarse-grained locking and TxOS uses optimistic multi-versioning based on software transactional memory. Suppose that two applications, `A1` and `A2`, create, write, and read files in a directory `D` in each transaction. Since the timestamp of `D` is updated upon every file creation, Valor locks `D` at the expense of concurrent execution of applications. Though TxOS maintains multiple versions of `D` for concurrent execution, due to the conflicting updates of the timestamp, only one application succeeds and the other should be re-executed.

In contrast, CFS does not provide isolation or concurrency control mechanisms. Applications must implement required concurrency control using existing synchronization primitives, such as file lock and mutex. Also, if there is a possibility of TOCTTOU races, applications should prevent the races themselves (for example, by using the `openat()` system call [4]). In fact, of the four ACID properties in DBMS, the isolation property is the most often relaxed. Popular DBMS implementations [38, 40, 48] provide at least four isolation levels for a user to choose. In the above example, if `A1` and `A2` need an isolated view of `D`, they must implement concurrency control themselves. Otherwise, no concurrency control is required resulting no overhead. The rationale behind our design is that isolation can be best implemented at the application level by exploiting correctness semantics of the applications.

	CFS	Valor [63]	TxOS [55]
Atomic update	Trans. flash	FS meta journal + logging	FS full journal
Isolation	None (app.)	Locking	Versioning
Performance	High	Low	Mid
Complexity	Low (5.8K)	Low (4.4K)	High (22.6K)

Table 4: Comparison among CFS and recent transactional file systems. Modified LOCs are in the parentheses on the bottom.

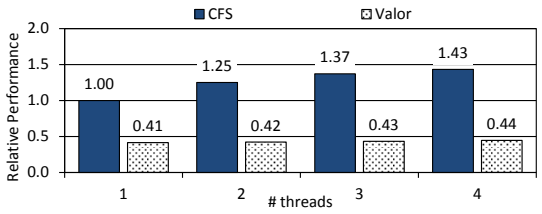


Figure 7: Multithreaded performance of CFS and Valor.

To verify the cost of isolation, we implemented the essential part of Valor in userspace and ran varying numbers of threads of the above example. As expected, performance of CFS increases as thread count increases and is constrained by storage bandwidth (Figure 7). However, due to conservative directory locking, performance of Valor is constant regardless of the number of threads. Also, since Valor uses logging, its single-thread performance is about 2× slower than CFS.

Running CFS on Non-transactional Storage: Another interesting question is whether CFS is applicable to non-transactional storage devices. The performance benefit of CFS is two-fold: simplified update protocols and no redundant writes that rely on transactional flash. Therefore, if atomic multi-page writes can be emulated on non-transactional storage, CFS is applicable and we can expect performance benefits from the simplified update protocols. Such atomic multi-page writes have long been studied (e.g., *atomic recovery unit* in Logical Disk [22]). The obvious design choice is to implement the atomic multi-page write using write-ahead-logging at the device mapper layer [57], which is a higher-level virtual block device on top of physical block devices in the Linux Kernel. To see its potential performance benefit, we ran the code in Figure 1 on ext4 data journal mode replacing `cfs_commit()` with two `fsync` calls. Though it is 2.9× slower than the CFS version, it is still 6.2× faster than the version using ext4 ordered journal (Figure 4). As future work, we will design and implement virtual transactional storage supporting CFS on non-transactional storage devices.

9 Related Work

Crash consistency is critical to operating system design, and many different approaches have been explored.

File System Consistency: To guarantee system-wide consistency of file system data structures, a variety of techniques have been proposed: journaling [39, 64], soft updates [25], copy-on-write [16, 29, 43, 58], and using a DBMS as a file system [27, 34, 45, 47, 49]. However, due to the lack of file system interfaces to support application-level crash consistency, applications had no choice but to implement complex update protocols using `fsync()`. Although several techniques [18, 19, 46] have been proposed to mitigate the performance penalty of `fsync()`, they cannot help to simplify these update protocols. A recent study revealed that widely-deployed applications, such as PostgreSQL, LevelDB, and HDFS, implemented their own ad-hoc update protocols, and thus still remained vulnerable to crashes [54]. We believe CFS is the first principled and practical way to change this landscape.

Transactional File Systems: There have been steady efforts to natively provide transactions with ACID properties to applications via file systems [36, 42, 55, 60, 63]. As we discussed in §8, transactional file systems and applications relying on them support only strong isolation. Considering that relaxation of isolation according to applications' correctness semantics is a key optimization technique, it is the critical limitation in practice. Also, complexity and overheads for strong isolation is not negligible; the most commonly used locking technique limits concurrent execution of multiple transactions [42, 60, 63]; sophisticated multi-versioning still shows non-negligible overhead (14% in TxOS [55]). Moreover, to achieve atomic and durable updates, transactional file systems rely solely on file system journaling [42, 63], or additionally maintain another write-ahead log for transactions [55, 60]. As a result, it was recommended to maintain transactions to small, mostly metadata operations [41].

Transactional Storage Devices: Several interesting approaches [20, 33, 51, 53, 56] have been proposed to support the transactional atomicity inside NAND flash storage devices. They exploit the log-structured mapping in FTL and atomically update the mapping table to achieve transactional atomicity. However, none of them resolve the false sharing of metadata pages. Thus they cannot support atomic update of multiple applications due to this lack of generality. For non-volatile memory storage, MARS [21] supports application transactions. But, it does not mention how MARS can be used to support file system consistency.

Atomic Update of Application Data: Recently, several techniques [37, 52, 65] have been proposed to protect application data from failures without supporting isolation like CFS. None of them handle the false sharing of metadata pages. Thus they cannot support atomic updates for arbitrary file system operations as CFS does. Failure-atomic `msync()` [52] atomically updates the changes of a `mmap`-ed file using REDO journaling. However, it only supports atomic update of a single `mmap`-ed file, and, due to the lack of the UNDO mechanism, the dirty data pages can not be stolen.

10 Conclusion

CFS is the first file system that natively supports application-level crash consistency on transactional flash storage. To guarantee crash consistency, applications can simply specify code regions that need atomic file system operations instead of implementing complex, slow, and error-prone update protocols by themselves. CFS guarantees the atomic propagation of data and metadata pages changed in the code region without relying on journaling through the use of the atomic multi-page write functionality of SSD/X-FTL. Our application case studies confirm that a variety of existing applications can be easily ported to CFS. Our experimental results show that CFS-based applications are 2–5× faster than the original versions.

Acknowledgments

The authors wish to thank our shepherd, Liuba Shrira, and the anonymous reviewers for their helpful comments. We thank to Jin-Soo Kim for motivational discussion, Gihwan Oh for helping us with SSD/X-FTL, and Sangman Kim for his feedback and comments. We also thank the various members of our operations staff who provided proofreading of this paper. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020730). This work was supported by ICT R&D program of MSIP/IITP [10041244, SmartTV 2.0 Software Platform]. Changwoo Min and Taesoo Kim are partly supported by ETRI MSIP/IITP [B0101-15-0644] and ONR N00014-15-1-2162.

References

[1] InnoDB Disk I/O in MySQL 5.7 Reference Manual. http://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html.

[2] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

[3] MariaDB An enhanced, drop-in replacement for MySQL. https://mariadb.org/.

[4] openat(2) - Linux man page. http://linux.die.net/man/2/openat.

[5] RL Benchmark:SQLite. http://redlicense.com/.

[6] SQLite. http://www.sqlite.org/.

[7] SQLite: Atomic Commit In SQLite. http://www.sqlite.org/wal.html.

[8] SQLite: Write-Ahead Logging. http://www.sqlite.org/wal.html.

[9] SysBench: a system performance benchmark. http://sysbench.sourceforge.net/.

[10] The OpenSSD Project. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[11] Vim the editor. http://www.vim.org/index.php.

[12] Apple. BSD System Calls Manual: FCNTL(2). https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fcntl.2.html.

[13] Apple. BSD System Calls Manual: FSYNC(2). https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fsync.2.html.

[14] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. LinkBench: a Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13.

[15] Axboe, J. FIO (Flexible IO Tester). http://git.kernel.dk/?p=fio.git;a=summary.

[16] Btrfs. http://btrfs.wiki.kernel.org.

[17] Cao, M., Santos, J. R., and Dilger, A. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium* (2008).

[18] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[19] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency Without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST '12.

[20] Choi, H. J., Lim, S.-H., and Park, K. H. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage 4*, 4 (feb 2009).

[21] Coburn, J., Bunker, T., Schwarz, M., Gupta, R., and Swanson, S. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[22] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993), SOSP '93, ACM.

[23] Debian. Apt. https://wiki.debian.org/Apt.

[24] Ext4 Wiki. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[25] Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. Generalized File System Dependencies. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07.

[26] Fusion-io. NVM Primitives API Specification 1.0. http://opennvm.github.io/nvm-primitives-documents/, feb 2014.

[27] Gehani, N. H., Jagadish, H. V., and Roome, W. D. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), VLDB '94.

[28] Google. LevelDB Benchmarks. http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html, July 2011.

[29] Hitz, D., Lau, J., and Malcolm, M. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference* (1994), WTEC '94.

[30] Intel. Intel® SSD 330 Series (120GB, SATA 6Gb/s, 25nm, MLC). http://ark.intel.com/products/67287/Intel-SSD-330-Series-120GB-SATA-6Gbs-25nm-MLC.

[31] Jeong, S., Lee, K., Lee, S., Son, S., and Won, Y. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), ATC '13.

[32] Kang, W.-H., Lee, S.-W., Moon, B., Kee, Y.-S., and Oh, M. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), SIGMOD '14.

[33] Kang, W.-H., Lee, S.-W., Moon, B., Oh, G.-H., and Min, C. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13.

[34] Kashyap, A. File System Extensibility and Reliability Using an in-Kernel Database. Master's thesis, Stony Brook University, December 2004. Technical Report FSL-04-06, http://www.fsl.cs.sunysb.edu/docs/kbdbfs-msthesis/kbdbfs.pdf.

[35] Kawaguchi, A., Nishioka, S., and Motoda, H. A Flash-memory Based File System. In *Proceedings of the 1995 USENIX Technical Conference Proceedings* (1995), ATC '95.

[36] Kim, S., Lee, M. Z., Dunn, A. M., Hofmann, O. S., Wang, X., Witchel, E., and Porter, D. E. Improving Server Applications with System Transactions. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, ACM.

[37] MariaDB. Fusion-io DirectFS atomic write support. https://mariadb.com/kb/en/mariadb/documentation/getting-started/mariadb-performance-advanced-configurations/fusion-io/fusion-io-directfs-atomic-write-support/.

[38] MariaDB. Isolation level. https://mariadb.com/kb/en/sql-99/37-sql-transaction-concurrency/isolation-level/.

[39] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007).

[40] Microsoft. Isolation Levels in the Database Engine. https://technet.microsoft.com/en-us/library/ms189122%28v=SQL.105%29.aspx.

[41] Microsoft. Performance Considerations for Transactional NTFS. http://msdn.microsoft.com/en-us/library/windows/desktop/ee240893(v=vs.85).aspx.

[42] Microsoft. Transactional NTFS (TxF). http://msdn.microsoft.com/en-us/library/bb968806(v=vs.85).aspx.

[43] Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y. I. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST '12.

[44] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transaction Database System 17*, 1 (Mar. 1992).

[45] Murphy, N., Tonkelowitz, M., and Vernal, M. The design and implementation of the database file system, 2002.

[46] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI '06.

[47] Olson, M. A. The Design and Implementation of the Inversion File System. In *Proceedings of the 1993 USENIX Winter Technical Conference* (1993).

[48] Oracle. Database Concepts: 9. Data Concurrency and Consistency. https://docs.oracle.com/cd/E11882_01/server.112/e40540/consist.htm.

[49] Oracle. Oracle Internet File System Setup and Administration Guide. http://docs.oracle.com/cd/A97336_01/cont.102/a81197/toc.htm.

[50] Oracle Corporation. Oracle VM VirtualBox® User Manual. http://www.virtualbox.org/manual/ch12.html#idp59653904.

[51] Ouyang, X., Nellans, D. W., Wipfel, R., Flynn, D., and Panda, D. K. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *17th International Conference on High-Performance Computer Architecture (HPCA)* (2011), pp. 301–311.

[52] Park, S., Kelly, T., and Shen, K. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13.

[53] Park, S., Yu, J. H., and Ohm, S. Y. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics (ISCE 2005)* (june 2005), pp. 155 – 160.

[54] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (2014), OSDI '14.

[55] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[56] Prabhakaran, V., Rodeheffer, T. L., and Zhou, L. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08.

[57] Red Hat Software. Device-mapper Resource Page. https://www.sourceware.org/dm/.

[58] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (Feb. 1992).

[59] Santos, J., and Rao, S. Flexible File System Benchmark). http://sourceforge.net/projects/ffsb/.

[60] Seltzer, M. I. Transaction Support in a Log-Structured File System. In *Proceedings of the 9th International Conference on Data Engineering* (1993).

[61] SNIA. NVM Programming Model (NPM) Version 1. Tech. rep., December 2013.

[62] SNIA. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise Version 1.1. Tech. rep., September 2013.

[63] Spillane, R. P., Gaikwad, S., Chinni, M., Zadok, E., and Wright, C. P. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proccedings of the 7th Conference on File and Storage Technologies* (2009), FAST '09.

[64] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (1996), ATC '96.

[65] Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., and III, C. B. M. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX conference on File and Storage Technologies* (2015), FAST'15, USENIX Association.

WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly

Wongun Lee,[*] Keonwoo Lee,[*] Hankeun Son,[*] Wook-Hee Kim,[†] Beomseok Nam[†] and Youjip Won[*]
[*]Dept. of Computer Software, Hanyang University, Seoul, Korea
[†]Ulsan National Institute of Science and Technology, Ulsan, Korea

Abstract

This work is dedicated to resolve the Journaling of Journal Anomaly in Android IO stack. We orchestrate SQLite and EXT4 filesystem so that SQLite's file-backed journaling activity can dispense with the expensive filesystem intervention, the *journaling*, without compromising the file integrity under unexpected filesystem failure. In storing the logs, we exploit the direct IO to suppress the filesystem interference. This work consists of three key ingredients: (i) *Preallocation with Explicit Journaling*, (ii) *Header Embedding*, and (iii) *Group Synchronization*. Preallocation with Explicit Journaling eliminates the filesystem journaling properly protecting the file metadata against the unexpected system crash. We redesign the SQLite B-tree structure with Header Embedding to make it direct IO compatible and block IO friendly. With Group Synch, we minimize the synchronization overhead of direct IO and make the SQLite operation NAND Flash friendly. Combining the three technical ingredients, we develop a new journal mode in SQLite, the `WALDIO`. We implement it on the commercially available smartphone. `WALDIO` mode achieves 5.1× performance (insert/sec) against `WAL` mode which is the fastest journaling mode in SQLite. It yields 2.7× performance (inserts/sec) against the `LS-MVBT`, the fastest SQLite journaling mode known to public. `WALDIO` mode achieves 7.4× performance (insert/sec) against `WAL` mode when it is relieved from the overhead of explicitly synchronizing individual log-commit operations. `WALDIO` mode reduces the IO volume to 1/6 compared against the `WAL` mode.

1 Introduction

Smart device, e.g. smartphone, smart TV, and smart pad, firmly position themselves as mainstream computing device. The mobile DRAM and mobile NAND Flash sales for smart device account for 30% [41] and 40% [8] of the world DRAM sales and NAND Flash sales, respectively. In the smartphone, the storage subsystem is arguably the main governing factor for performance [23].

Android IO stack suffers from the excessive IO behavior. Sending two character message, 'Hi', through the text messaging application yields at least 48 KByte of writes to the storage device. This anomalous amplification is due to the uncoordinated interaction between SQLite and EXT4 filesystem. The broken relationship between the EXT4 filesystem and SQLite is caused by the fact that SQLite synchronizes each change in the database file or in rollback journal file through `fsync()/fdatasync()` and that each call to `fsync()/fdatasync()` triggers the bulky EXT4 journal module to log the updated metadata. This phenomenon is called Journaling of Journal Anomaly [19].

There have been a number of efforts to mitigate the Journaling of Journal anomaly [19, 27, 35, 25, 38]. These works either modify SQLite to reduce the number of `fsync()` calls [19, 27] or modify the filesystem to mitigate the overhead of a single `fsync()` [19, 35, 25, 38]. While the overheads may vary, these works still need to journal the metadata of the SQLite journal file for each database transaction.

In this work, we dedicate our effort in resolving Journaling of Journal anomaly. We orchestrate EXT4 filesystem and SQLite so that SQLite can dispense with the expensive filesystem journaling in maintaining its journal file without compromising the file integrity under the unexpected system failure. We successfully eliminate the root cause for Journaling of Journal anomaly, the *filesystem journaling*. In our optimization effort, SQLite exploits "direct IO" in updating its journal file. Our work consists of three key technical ingredients: (i) Block Preallocation with Explicit Journaling, (ii) Header Embedding and (iii) Group Synch.

- **Preallocation with Explicit Journaling**: We preallocate the data blocks to the SQLite journal file and explicitly journal the file metadata. The subse-

quent direct IO based log-commit operation does not incur any metadata update and the filesystem journaling can be eliminated. Via explicit journaling, the SQLite journal file is protected by the underlying filesystem against the unexpected system failure.

- **Header Embedding**: We develop Header Embedding and re-design the journal file structure. With Header Embedding, the fragmented SQLite journal file structure becomes 4 KByte aligned. The Header Embedding makes the SQLite journaling operation direct IO compatible and block IO friendly.
- **Group Synchronization**: With Group Synchronization (Group Synch in short), we aggregate multiple logs and to synchronize them as a single unit. Group synch effectively reduces the overhead of synchronizing the direct IO based log-commit operation to the storage surface. It significantly improves the performance via aligning the IO with NAND Flash page size.

Combining all these techniques, we develop a new SQLite journal mode, the `WALDIO`. We implement the `WALDIO` mode in commercially available smartphone model (Samsung Galaxy S5). `WALDIO` mode with persistent direct IO exhibits 5.1× performance (insert/sec) and 2.7× performance (insert/sec) against `WAL` mode which is the fastest stock SQLite journal mode and `LS-MVBT` [27] mode which is the fastest SQLite journal mode known to public, respectively. Smartphone with non-removable battery can potentially make the direct IO a persistent operation for practical purpose, in which case `WALDIO` yields 7.4× performance (insert/sec) against `WAL` mode and 4.0× performance (insert/sec) against `LS-MVBT` mode, respectively. The improvement in `update` and `delete` follow the similar trend. `WALDIO` mode reduces the write volume of SQLite to 1/6 compared to `WAL` mode.

2 Background

2.1 SQLite

SQLite is a serverless embedded DBMS. SQLite is *the* way of maintaining the records in various smartphone platforms, e.g. Android, iOS, Tizen and etc. and is widely used as the embedded DBMS for desktop applications, e.g. Chrome web browser, Firefox, Adobe Acrobat reader, Skype [40].

SQLite adopts B-tree for its database. The size of the B-tree node is power of two ranging from 512 Byte to 64 KByte. Default node size is 1024 Byte. Fig. 1 illustrates the leaf node structure. The B-tree node consists of the page header, the index array and the cell array. The page header resides at the beginning of the node. Next to the page header, there exists an index array. Each index points the variable size record. The record, which is called *cell* in SQLite, is allocated from the end of the node. The index array and the records grow in the opposite direction. When a cell is deleted, the space occupied by the deleted record is marked as dead. The page header maintains the number of the deleted cells. The deleted cells are weaved together as a linked list. SQLite allocates a new node when there is no more free space in the page or the deleted area.

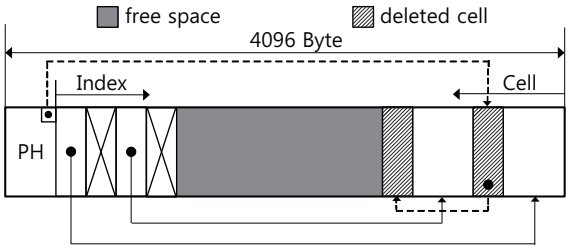


Figure 1: SQLite B-tree node structure, node size = 4 KByte, PH: Page header

Different from the large scale DBMS [39, 29, 10], SQLite does not have its own storage management module. SQLite heavily relies on the underlying filesystem to persistently manage its information and to protect it against unexpected system failure. SQLite uses *file* to maintain the log for crash recovery. For transactional guarantee, SQLite explicitly synchronizes, i.e. `fdatasync()`, the log file and the database file after committing the log or after updating the database, respectively.

SQLite provides six journal modes: `DELETE`, `TRUNCATE`, `PERSIST`, `WAL`, `MEMORY` and `OFF`. As the name suggests, `MEMORY` mode and `OFF` mode maintain the journal information in memory and does not maintain the journal information, respectively. The remaining four journal modes can be categorized into two: rollback journal and rollforward journal. `DELETE`, `TRUNCATE` and `PERSIST` modes are for rollback and `WAL` mode is for rollforward journaling, respectively.

In the rollback journal mode, the SQLite operation, e.g. `INSERT`, `DELETE` and `UPDATE`, consists of three phases: (i) logging, (ii) database update and (iii) log reset. The three SQLite journal modes in rollback journaling share the first and the second phase. In logging phase, SQLite updates the journal header and logs the old database pages (undo log) in the journal file. In database update phase, the updated database pages are written to the database file. The objective of the log-reset phase is to mark that a given transaction has successfully completed. In the third phase (log reset), there is

minor difference among the three SQLite journal modes. In DELETE mode, SQLite deletes the journal file. In TRUNCATE mode, SQLite truncates the journal file to 0. In PERSIST mode, SQLite puts special mark at the beginning of the journal file to denote that the transaction has completed. While the difference is subtle, it bears the profound implication on the filesystem journaling overhead. In DELETE mode, SQLite always needs to create the new journal file in the logging phase. Creating a file accompanies the large amount of metadata updates; the directory block, inode table, block bitmap and etc. All these metadata need to be journaled by the filesystem when fsync()/fdatasync() is called.

TRUNCATE mode retains the inode and deallocates the file blocks. Since TRUNCATE mode does not create the journal file, it yields smaller amount of metadata update compared to DELETE mode. As a result, when fsync()/fdatasync() is called in the logging phase, it yields smaller amount of EXT4 journal IO compared to DELETE mode.

PERSIST mode recycles not only the inode but also the file blocks. In PERSIST mode, the "logging" updates only the time related fields in the file metadata, e.g. mtime. Compared against TRUNCATE mode, the PERSIST mode further reduces the amount of metadata to be fsync()'ed in the logging phase. Via replacing the fsync() with fdatasync(), PERSIST mode achieves more reduction on the amount of metadata journaled in the "logging" phase [19].

In WAL mode, SQLite appends the header and a set of updated database pages to the log file (redo log). We call this file as WAL file for convenience's sake. When the database table is closed or the number of committed database pages in WAL file reaches the predefined maximum, the committed database pages in WAL file are checkpointed to the database file. SQLite provides two options to synchronize the committed logs: Full Sync and Normal Sync. In Full Sync, SQLite calls fsync()/fdatasync() after each log-commit to persistently store the logs. In Normal Sync, SQLite calls fsync()/fdatasync() after each checkpoint. In Normal Sync option, the committed logs reside in the buffer cache till they are either checkpointed by SQLite or flushed by OS. The logs in the buffer cache are subject to loss in case of unexpected system failure, e.g. power failure, or operating system crash [6] and the durability of a transaction can be compromised. The default option is Full Sync.

2.2 EXT4 Journaling

EXT4 filesystem provides three journal modes; Journal, Ordered and Writeback. The Ordered mode is the most widely used one. In Ordered mode, the filesystem logs only the updated metadata. When logging the metadata, the filesystem flushes all the data blocks related to the updated metadata and then it logs the updated metadata. EXT4 journaling module is bulky. An EXT4 journal transaction consists of a 4 KByte journal header, a set of 4 KByte journal records each of which corresponds to the updated filesystem block and a 4 KByte journal commit block. EXT4 journaling module is activated either on regular basis, e.g. in 5 sec interval, or via an explicit call to fsync() or fdatasync().

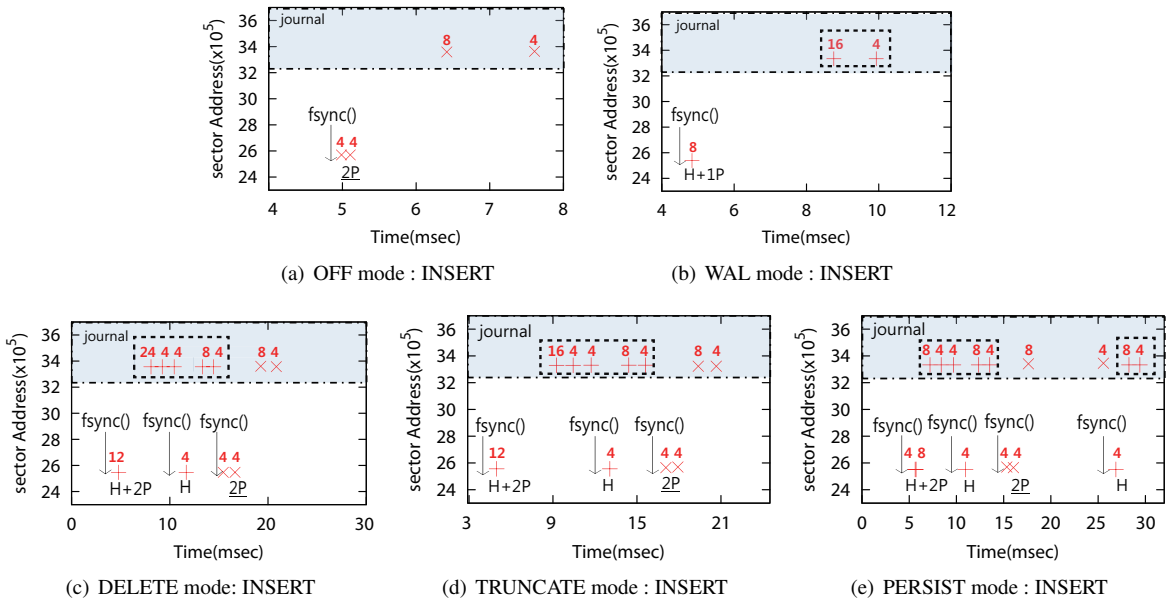
EXT4 journaling module functions efficiently when it is triggered in sufficiently large interval, e.g. in every 5 sec. With the large interval, the journal descriptor and the journal commit block pair carries sufficiently large amount of journal records in a single journal transaction. The overhead of journal descriptor and journal commit block is insignificant. SQLite drives the EXT4 filesystem in a way which, we carefully believe, has not been foreseen before and brings unacceptable inefficiency in Android IO stack. SQLite calls fdatasync() very frequently, typically after very few number of 4 KByte writes [19]. In fdatasync(), appending a 4 KByte block to a file accompanies at least 12 KByte of EXT4 journal writes.

3 Analysis of Journaling of Journal Anomaly

We overhaul the interaction between the SQLite and EXT4. SQLite inserts one 100 Byte record into an empty database table and we examine the block level IO behavior of the underlying filesystem. We use open-source benchmark, Mobibench [31] and MOST [18] to generate the workload and to analyze the IO trace, respectively. We examine the IO behavior under five SQLite journal modes: OFF, WAL, DELETE, TRUNCATE, and PERSIST.

Fig. 2 illustrates the block access patterns of SQL INSERT operations under five SQLite journaling modes. We mark SQLite journal related IO's and SQLite database related IO's with '+' and 'x', respectively. Each '+' and 'x' mark is annotated with the respective IO size in KByte unit. In the X-Y plane, the EXT4 journal region is marked with the light-grey background. The '+' marked IO's in the light-grey region correspond to the EXT4 journal writes for the SQLite journal file; *Journaling of Journal* overhead. We annotate each write in EXT4 data region with its type; the writes to journal file can be for journal header (H) or for journal record (P), respectively.

In OFF mode, SQLite synchronizes only the database file and does not accompany any SQLite journaling related IO (Fig. 2(a)). EXT4 filesystem writes two data blocks for database file and journals the respective metadata. The total 3 blocks are written in EXT4 journal re-



(a) OFF mode : INSERT (b) WAL mode : INSERT

(c) DELETE mode: INSERT (d) TRUNCATE mode : INSERT (e) PERSIST mode : INSERT

Figure 2: SQLite Block IO pattern (H: Journal Header or WAL Frame Header, P: DB page)

gion: one page of journal descriptor, one page for updated metadata and one page of journal commit mark.

In `WAL` mode (Fig. 2(b)), SQLite writes the redo log to WAL file and synchronizes the WAL file via `fdatasync()`. As a result of calling `fdatasync()`, EXT4 journals the updated metadata of the journal file, separately synchronizing the journal descriptor and journal commit mark. Since the committed database pages are checkpointed to the database file in batched manner, we do not observe any IO on the database file in Fig. 2(b).

Fig. 2(c), Fig. 2(d), and Fig. 2(e) illustrate block access patterns for rollback journal modes; `DELETE`, `TRUNCATE` and `PERSIST`, respectively. The rollback journal modes synchronize both the rollback journal file and the database file after they are updated. Compared to `WAL` mode, the filesystem journaling overhead doubles. In all these rollback journal modes (Fig. 2(c), Fig. 2(d), and Fig. 2(e)), the first and the second `fsync()` are for synchronizing the rollback journal file (phase 1: logging). The third `fsync()` is for synchronizing the updated database file (phase 2: update the database). `PERSIST` mode carries an additional `fsync()` to persistently store the reset mark in the log file (phase 3: log-reset).

The Journaling of Journal overhead for `DELETE`, `TRUNCATE` and `PERSIST` mode corresponds to 44 KByte, 36 KByte and 40 KByte, respectively. These differences are due to the way in which the SQLite journal mode resets the log file. The `WAL` mode yields the smallest JOJ overhead, 20 KByte.

Table 1 summarizes the traffic volume for five SQLite journal modes. In all SQLite journal modes, the filesystem intervention is overly excessive; the filesystem journaling activity accounts for more than 50% of the IO. While `WAL` mode yields the smallest amount of total IO, it is still subject to extreme IO inefficiency. In WAL, the filesystem journaling accounts for 70% of the entire IO traffic (20 KByte out of 28 KByte). `WAL` mode yields the smallest IO overhead and in the mean time, bears the largest room for improvement when the filesystem journaling overhead is eliminated.

	IO type (Write, KB)			
Mode	Data	Journal	JOJ	Total
`OFF`	8	12	0	20
`WAL`	8	20	20	28
`DELETE`	24	56	44	80
`TRUNCATE`	24	48	36	72
`PERSIST`	28	52	40	80

Table 1: IO Volume in inserting 100 Byte (DATA: EXT4 Data region, Journal: EXT4 Journal region, JOJ: EXT4 journal writes for SQLite journal file, and Total)

4 Direct IO and SQLite

4.1 Direct IO

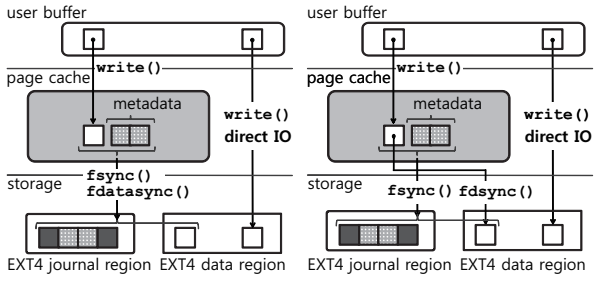
Direct IO is a filesystem feature which allows the user to read and to write the data directly from and to the storage device. In direct IO, the data block is immediately written to the storage device bypassing the page cache. Direct IO based write, DIO write for short, returns when the data blocks reach the writeback cache of the storage device.

DBMS [13, 3] and Virtual Machine Monitor [28, 33] use direct IO to manage the storage device with minimum file system intervention.

SQLite maintains the logs in a journal file. It uses buffered write in storing the log to the journal file and explicitly synchronizes the journal file via `fsync()/fdatasync()` for durability guarantee. Flushing the logs in the buffer cache can accompany the expensive filesystem journaling. If SQLite uses direct IO to write the log to the journal file, it can save the filesystem from expensive filesystem journaling activity.

4.2 Writing a Block to the Storage

We examine the three ways to write a block to the storage device: (i) `write()` followed by `fsync()`, (ii) `write()` followed by `fdatasync()` and (iii) DIO `write()`. These approaches differ in a way in which the filesystem handles the updated metadata. In `fsync()`, EXT4 filesystem journals the updated metadata for the respective file. In `fdatasync()`, EXT4 filesystem journals the updated metadata only when the file block is allocated (or deallocated). DIO write by itself does not entail any filesystem journaling. We can categorize the write operations into two types: *allocating write* and *non-allocating write*. Allocating write is a write system call which requires an allocation of a new filesystem block. Allocating write updates the various metadata, e.g. the block bitmap, inode table, intermediate node block and etc. Non-allocating write does not entail the allocation of a file block. It updates only access time related fields and possibly the `initialized` flag in the metadata.



(a) Allocating Write (b) Non-Allocating Write

Figure 3: Writing a block with `fsync()`, `fdatasync()` and direct IO

Fig. 3(a) and Fig. 3(b) schematically illustrate the IO paths of three different ways of writing 4 KByte to the storage device, for allocating and non-allocating write, respectively. For both allocating and non-allocating write, `fsync()` journals the updated metadata. In allocating write, `fdatasync()` exhibits the identical behavior as `fsync()`. In non-allocating write, `fdatasync()` does not journal any metadata. For both allocating and non-allocating write, direct IO does not accompany the filesystem journaling. In direct IO, the updated metadata, if there is any, can be subject to loss.

SQLite provides two options to synchronize the database (or journal) file: via `fsync()` and via `fdatasync()`. Android platform legitimately uses `fdatasync()` in SQLite to reduce the filesystem journaling overhead. Overhauling the IO behavior, we find an important caveat to resolve the Journaling of Journal anomaly, the *EXT4 journaling overhead*. In non-allocating write, "DIO write" yields the same behavior, though not precisely identical, with the "buffered write followed by `fdatasync()`" from the filesystem journaling's point of view; in delivering the data blocks to the storage, they both are free from the filesystem journaling.

5 Eliminating Filesystem Journaling in Android IO

We propose to use direct IO based write operation for committing the logs to the SQLite journal file so that the logs are directly written to the storage and the activity of committing the logs does not accompany any updates in the page cache entries; neither the data block nor the metadata. With this approach, the synchronization activity of SQLite, e.g. `fdatasync()`, does not trigger any filesystem journaling related IO. Our scheme consists of three key technical ingredients: (i) Preallocation with Explicit Journaling, (ii) Header Embedding, and (iii) Group Synch. Combining all these, we develop a new SQLite journal mode, `WALDIO`.

5.1 Preallocation with Explicit Journaling

The prime concern is to eliminate the interference of the EXT4 journaling in the log-commit operation and at the same time to protect the metadata of the journal file. We develop *Preallocation with Explicit Journaling*, where (i) we preallocate a certain amount of *initialized* blocks for a WAL file and (ii) journal the metadata for the created WAL file via explicitly calling `fdatasync()`. In this approach, we do rely on filesystem journaling to protect the metadata of the SQLite journal file, but suppress the every log-commit operation to accompany filesystem journaling. Fig. 4 schematically illustrates the detailed process; (i) WAL file is preallocated with the initialized blocks (labeled as 1), (ii) the metadata of the WAL file is synchronized to disk via `fdatasync()` (labeled as 2), and (iii) the logs are committed to WAL file via direct IO (labeled as 3, 4 and 5).

EXT4 filesystem maintains an `initialized` flag for each data block. A file block is said to be initialized when this flag is set. Any attempts to read the uninitialized

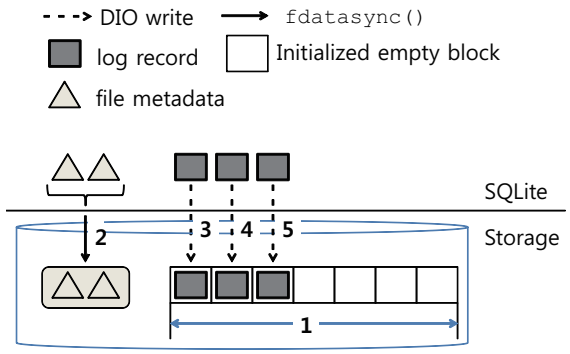


Figure 4: Preallocation with Explicit Journaling, 1: Preallocate the initialized blocks, 2: Journal the updated metadata, `fdatasync()`, 3, 4, and 5: Commit the logs with DIO

block are returned all 0's. The primary reason for this mechanism is to avoid exposing the stale data.

Being pre-allocated with the file blocks, the direct IO based log-commit operation becomes non-allocating write, where both the page cache entries and the metadata of the WAL file remain intact. The log-commit operation in `WALDIO` mode leaves no room for filesystem journaling module to interfere with. `WALDIO` mode saves the SQLite from the expensive filesystem journaling. When the WAL file is created or extended, `WALDIO` calls `fdatasync()` to synchronize the file metadata. With Explicit journaling, the WAL file becomes robust against the system failure.

In Preallocation, a special care needs to be taken to initialize the allocated blocks. Otherwise, the logs written in `WALDIO` mode may not be readable after unexpected system failure. Let us explain why. The `fallocate()` system call of EXT4 returns the uninitialized blocks. They are initialized when they are written for the first time. When a block is written with direct IO, the filesystem sets the respective `initialized` flag if it has not been initialized yet. However, since DIO write does not accompany the filesystem journaling, the updated flag is subject to loss under the unexpected system failure. While the dirty page cache entries and the updated metadata are synchronized to the storage in every few seconds, e.g. 5 sec, the contents in the writeback cache of the storage device are written to the storage surface in much shorter interval, e.g. in typically a few msec. Under the unexpected system failure, therefore, the logs written with direct IO may become unreadable even when they actually exist in the storage due to the unavailability of the `initialized` flag.

We propose three approaches to initialize the allocated blocks and subsequently to guard the stale contents in the allocated blocks against the exposure. The first and the easiest approach is to zero-fill the allocated blocks prior to use. In the second and the third approaches, we exploit the `discard` (or `trim`) command in the eMMC storage [1] to guard the stale content against the exposure. The `discard` command takes the list of the logical block addresses as an input and asks the eMMC storage to remove the mapping table entries for the respective logical blocks. In the second approach, we mount the filesystem with `discard` option and modify `fallocate()` to allocate the blocks with `initialized` flag set. When a filesystem uses `discard` mount option, it issues a `discard` command when the file blocks are deallocated. To force the `fallocate()` to return the blocks with `initialized` flags set, we port the existing `NO_HIDE_STALE` patch [34] to Linux source for Samsung Galaxy S5. In the third approach, we modify the `fallocate()` to allocate the blocks with `initialized` flag set and to discard the allocated blocks. We embed the `discard` command to the `NO_HIDE_STALE` patch [34] developed for the second approach and we develop a new flag `NO_HIDE_STALE_DISCARD` for `fallocate()`. The main difference between the second and the third approach is the time when the blocks are unmapped. In the second and third approaches, the file blocks are unmapped when they are deallocated and when they are allocated, respectively. In the second approach, the filesystem issues `discard` command for all deallocated blocks. Meanwhile, in the third approach, the filesystem discards only the file blocks allocated to WAL file. The third approach yields the smaller overhead than the second one.

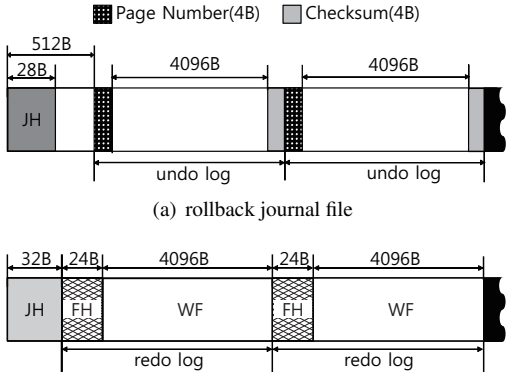
Each of these three approaches has pros and cons. The zero-fill operation accompanies IO overhead. Using `discard` mount option may slow down the filesystem [37]. Many recent smartphone devices including our test platform Galaxy S5 mount the filesystem with `discard` option. We implement all these schemes in our test platform. Via implementing all three schemes, one can choose the right scheme to initialize the allocated blocks subject to the available features of the underlying filesystem and storage device.

There exists an important implementation specific issue which deserves further attention. The `discard` command is designed to make the garbage collection more efficient [21]. It is not designed to hide the stale content. The eMMC standard [1] does not define what needs to be read when the discarded blocks are accessed. Some eMMC products, e.g. the one used by Samsung Galaxy S5 (Part No. MBC4GC), return all 0's when the discarded block is accessed. To use the `discard` command to hide the stale content, one needs to assure that the given eMMC product does not leak the stale content, i.e. is guaranteed to return all 0's or all 1's when the discarded block is accessed. Otherwise, one needs to take the resort to use `trim` command even though it is subject to larger overhead. The `trim` command is defined

to return all 0's or all 1's when the trimmed block is accessed [1]. On the same token, one also needs to assure that the given eMMC product does not ignore the `discard` command from the host in any circumstances, e.g. when the eMMC device is busy for performing background garbage collection.

5.2 Header Embedding

The direct IO operation fails when the IO size is not sector aligned. In SQLite, neither the redo log nor the undo log structures are sector aligned. Full fledged DBMS to align its database and cache organization for efficient block device interaction [10, 29, 39]. We reorganize the structure of SQLite WAL file and the database page to integrate direct IO into SQLite. Figures in Fig. 5 illustrate the redo and undo log structures of SQLite, respectively. B-tree node size is 4 KByte. The undo log of SQLite consists of 4 Byte prefix (page number), the 4 KByte journal record, and 4 Byte checksum (Fig. 5(a)). Each of these components is separately written with `write()`. In WAL file, a redo log consists of 24 Byte frame header and 4 KByte WAL frame (Fig. 5(b)). They are written separately as well. This fragmented data structure of SQLite bars the use of direct IO in managing its journal file.



(a) rollback journal file

(b) WAL file

Figure 5: Journal file structure of SQLite, node size: 4 KByte

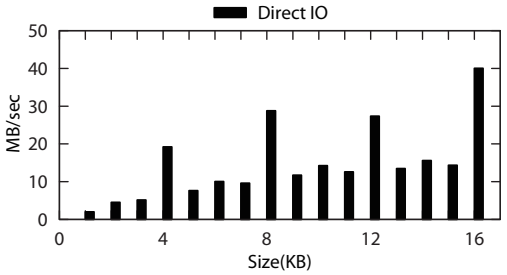


Figure 6: Effect of IO size: Sequential Write, Direct IO (GalaxyS5 eMMC Part No.: MBG4GC)

We examine the DIO write performance under varying IO size from 512 Byte to 16 KByte (Fig. 6). The sequential write performance of Samsung Galaxy S5 reaches over 70 MByte/sec. With 512 Byte IO size, the sequential write is subject to extreme inefficiency yielding mere 2 MByte/sec. This is because the IO size is not aligned with the block size. We observe larger degree of performance improvement when the IO size is aligned with the filesystem block size, 4 KByte, or NAND Flash page size, 8 KByte, respectively. This trend persists beyond 16 KByte IO size. This simple experiment provides an important direction for our optimization effort; Align the IO with the filesystem block size and with the NAND Flash page size.

We develop *Header Embedding* to align the SQLite IO with the filesystem block size. Instead of maintaining the header outside the log record, we embed the WAL header and the frame header into the header page and the WAL frame, respectively. In WALDIO, we set the B-tree node size to 4 KByte. WAL header is placed at the free space between database header and schema table in the root node of database B-tree. We harbor the 24 Byte frame header at the end of WAL-frame. Fig. 7 illustrates the log structure with header embedding. Embedding the frame header into the B-tree node, the available space in the B-tree node decreases. We physically examine the free spaces in B-tree nodes of SQLite. With few exceptions, there exists sufficient room to harbor 24 Byte field. Reserving 24 Byte for Header in the node, therefore, will not increase the number of nodes in the B-tree.

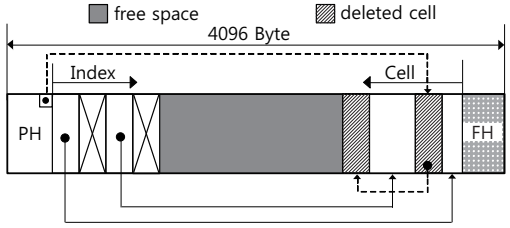


Figure 7: SQLite database page structure with Header Embedding (PH: Page Header, FH: Frame Header)

We examine the efficiency of the different aligning schemes for redo log structure. In page padding, SQLite pads the WAL header and WAL frame header to make them 4 KByte aligned. We examine four schemes: WAL (the original one), WAL with page padding, WALDIO with page padding, and WALDIO with header embedding. We perform 1,000 `INSERT` operations and examine the total IO volume. Fig. 8 illustrates the result. In `WAL` mode, total 29 MByte is written. Among 29.0 MByte, file data and EXT4 journal writes account for 12.3 MByte and 16.7 MByte, respectively. The size of WAL log record is 4120 Byte (24 Byte header and 4096 Byte frame). In committing 1,000 log entries of 4120

Byte each, total 12.3 MByte is written to filesystem data region. The fragmented log structure puts unnecessary stress on the storage device. Via properly aligning the log

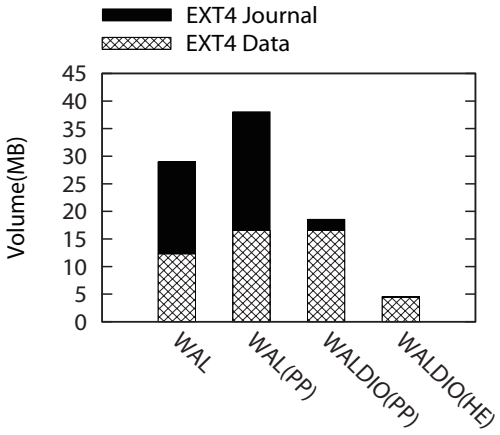


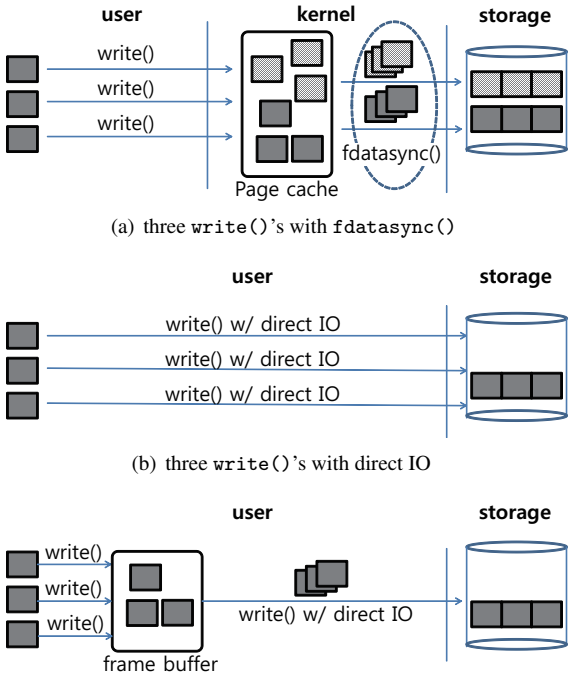
Figure 8: Total IO volume: 1000 INSERT in SQLite (PP: page pading, HE: header embedding)

structure, we reduce the IO volume to EXT4 Data region to 1/3 from 12.3 MByte to 4.4 MByte. With simple modification on the log structure accompanied by direct IO, the total IO volume decreases to 1/6 from 29.0 MByte to 4.6 MByte. The performance result will be dealt with in section 6.3.

5.3 Group Synchronization

Each layer in the IO stack, e.g. DBMS, filesystem, and block device layer, aggregates the IO's on its own way to remove the IO bottleneck [9, 11, 15, 5]. While `WALDIO` mode is successful in eliminating the filesystem journaling overhead, the individual log-commit operations are separately issued to the storage device. This nature of direct IO bars the underlying Operating System from aggregating and coalescing the IO's. If the logs are immediately synchronized to the storage surface after they are written with direct IO, the storage behavior is subject to further inefficiency since the storage device loses the opportunity to exploit its writeback cache. In this situation, the IO size plays a rather critical role in the storage performance. When the IO size is not properly aligned with the NAND Flash page size, it may cause read-modify-write problem [4, 24], proper handling of which requires complicated firmware technique such as subpage mapping [22].

We develop *Group Synchronization* (Group Synch) to mitigate the synchronization overhead of the direct IO based log-commit operation. In Group Synch, we employ *frame buffer* and *grouping interval*. All log records which have been written during a grouping interval are maintained at the frame buffer. When the grouping interval expires or when the frame buffer is full, SQLite flushes the frame buffer with DIO write. The Group Synch shares much of its idea with the prior arts; Group Commit from DBMS [9, 11] and Anticipatory disk scheduling from Operating System [17].



(a) three `write()`'s with `fdatasync()`

(b) three `write()`'s with direct IO

(c) three `write()`'s in Group Synch

Figure 9: Writing three blocks to storage: `fdatasync()`, direct IO vs. Group Synch

Figures in Fig. 9 schematically illustrates the IO behavior in writing three blocks to the storage. In Fig. 9(a), each of three blocks is written with separate `write()`'s (buffered IO) and then `fdatasync()` is called to synchronize them. We observe two sets of writes: dark gray ones and the light gray ones. The dark gray blocks correspond to data blocks. They are flushed to the disk with a single IO request. The set of light gray blocks correspond to EXT4 journal writes. In Fig. 9(b), each of three blocks is separately written via direct IO. Each of the IO requests is synchronously delivered to disk, yielding significant overhead. Fig. 9(c) illustrates the IO behavior in the Group Synch. The IO requests are first accumulated at the frame buffer and then flushed to the disk as a single DIO write. With Group Synch, the three blocks are written with single IO without accompanying the filesystem journaling. The benefit of Group Synch is twofold: reduce the the overhead for synchronizing the DIO write's and align the IO with NAND Flash page size.

Group synch provides weaker transactional guarantee than the other SQLite journaling modes since larger number of logs may get lost under system failure. However, we carefully conjecture that the difference may be

less than significant if the frame buffer size is properly set. In our limited empirical study, we find that a single SQLite transactions updates a number of database tables and indexes. For example, in contact manager application of Android, inserting an address book entry yields more than 8 logs to to the WAL file. We currently set the frame buffer size to four pages (16 KByte).

5.4 Durability

WALDIO should use Full Sync option to make the result of log-commit operation durable. When the WALDIO mode is used with Full Sync option, the log commit operation consists of two phases: (i) writing a log to the storage via DIO write and (ii) flush the writeback cache of the storage device via calling fdatasync() (Fig. 10). For better performance, we exploit Reliable Write command of eMMC standard [1] and implement *persistent direct IO*, PDIO. With PDIO write, WALDIO mode writes the logs directly to the storage surface bypassing the writeback cache of the Flash storage (Fig. 10). The blocks written with *persistent direct IO* are guaranteed to survive the power crash. With PDIO write, we can dispense with Full Sync option since each log-commit becomes immediately durable.

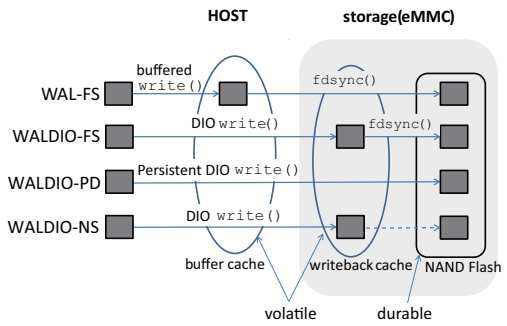


Figure 10: Making the log-commit durable: WAL with Full Sync, WALDIO with Full Sync, WALDIO with PDIO, WALDIO with Normal Sync

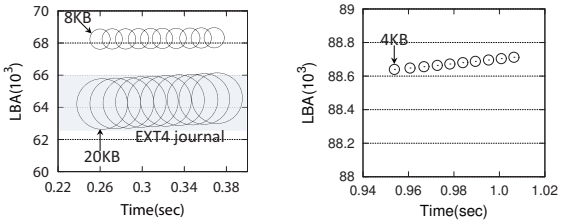
Non-removable battery in the smartphone can potentially make the DIO write a persistent one. Different from the logs in the buffer cache (Fig. 10), the logs in the writeback cache of the storage can survive the warm failure, e.g. Operating System crash [6] or kernel-panic [14]. It is very unlikely that the software bugs power off the device unexpectedly; Lue et.al. [35] reported that only 0.05% of AOSP software defect reports are related to the unexpected power failure. Also, as long as the power supply leaves some slack for eMMC to flush its writeback cache (typically a few msec), the content in the writeback cache will eventually be written to the storage surface. Given the rarity of the occasion, some application developers may prefer trading the perfect durability guarantee with the *almost* perfect durability guarantee with performance boost. For the device with non-removable battery, we carefully argue that WALDIO makes the Normal Sync option as one of the feasible choices for transactional guarantee for practical purpose.

6 Experiment

We examine the performance of WALDIO mode. We compare the behavior of the six SQLite journal modes: DELETE, TRUNCATE, PERSIST, WAL, LS-MVBT [27] and WALDIO. We implement these techniques in the recent smartphone model (Galaxy S5, Samsung, Android 4.4.2 (KitKat), Qualcomm MSM8974 Quadcore 2.5 GHz, 2 GByte DRAM, 32 GByte eMMC with 8 Kbyte page). We examine the performance of SQLite operations; INSERT, UPDATE and DELETE. We use Mobibench [31] and MOST [18] to generate the workload and to analyze the trace, respectively. We use NO_HIDE_STALE_DISCARD flag in Preallocation.

6.1 IO Access Pattern

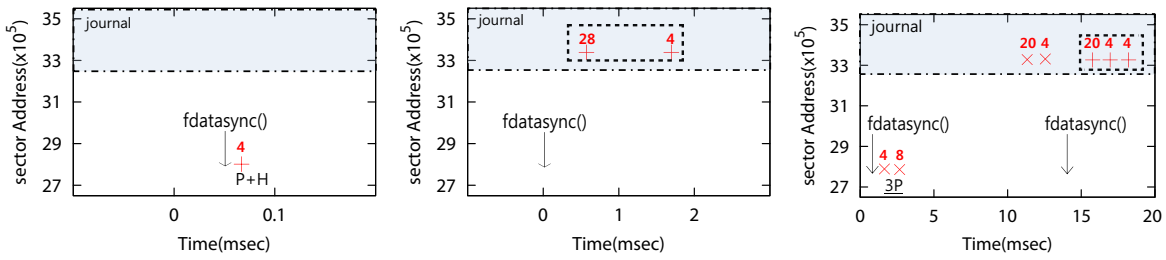
We first examine the IO access pattern of the newly proposed journal mode, WALDIO. With WALDIO mode, we insert a single 100 Byte record. Fig. 11 illustrates the result. In WALDIO, INSERT operation generates single page write (Fig. 11(a)). Be reminded that a single INSERT of 100 Byte record yields 80 KByte page writes and 28 KByte page writes in DELETE mode and WAL mode, respectively (Table 1). For illustrative purpose, we also show the IO accesses when the journal file is created (Fig. 11(b)) and when the journal file is extended (Fig. 11(c)), respectively.



(a) WAL mode (b) WALDIO mode

Figure 12: IO trace for ten INSERTs: WAL mode vs. WALDIO mode (Full Sync option)

To visualize the improvement on IO volume, we examine the IO trace for 10 INSERT operations in WAL mode (Fig. 12(a)) and in WALDIO mode (Fig. 12(b)), respectively. In this figure, the center and the radius of each circle denote the start address and the size of an IO, respectively. The circle radius is linearly proportional to the actual IO size. In WAL mode, each log-commit writes



(a) INSERT (b) Preallocation with Explicit Journaling (c) Checkpoint

Figure 11: IO accesses in WALDIO mode (P+H: Header embedding WAL Frame, 3P: three DB Pages)

8 KByte to the SQLite journal file and 20 KByte to EXT4 journal region. Due to the fragmented log structure, SQLite writes two 4 KByte blocks in committing the 4120 Byte log (24 Byte header and 4096 Byte page). In WALDIO, each log-commit yields 4 KByte IO since the frame header is embedded within the B-tree node. It does not entail any EXT4 journal IO since the log is committed with direct IO. An INSERT operation writes 28 KByte and 4 KByte to the storage in WAL and WALDIO, respectively. WALDIO successfully eliminates the filesystem journaling overhead and brings significant reduction on the total IO volume written to the storage.

6.2 Performance of Header Embedding

We examine the performance of four different page aligning schemes in WALDIO: sector padding, 4 KByte page padding, 8 KByte page padding and Header Embedding. We include the SQLite performance in WAL mode as the baseline. Fig. 13 illustrates the result. When the WAL file is sector padded (sector aligned), employing direct IO barely brings any performance gain against the WAL mode. When IO size is not aligned with the filesystem block size, the overhead of synchronously writing each data block offsets the benefit of eliminating the filesystem journaling overhead. When the WAL file structure is aligned with block size (4 KByte), the performance increases by 60% against the WAL. When the WAL file structure is aligned with NAND Flash page size (8 KByte), the SQLite performance increases by 100% against WAL. Via embedding the frame header information into the WAL frame, the SQLite yields 2.1× performance against WAL mode from 587 insert/sec to 1239 insert/sec.

6.3 WALDIO, the performance

We discuss the performance impact of WALDIO journal mode against existing SQLite journal modes: DELETE, TRUNCATE, PERSIST, WAL and LS-MVBT [27]. In WALDIO, we examine the performance under three synchronization options: (i) Full Sync, WALDIO-FS, (ii) Persistent Direct IO, WALDIO-PD and (iii) Normal Sync, WALDIO-NS. We examine the WALDIO performance with and without Group Synch. In Group Synch, the frame buffer size is set to 16 KByte with 2 msec grouping interval.

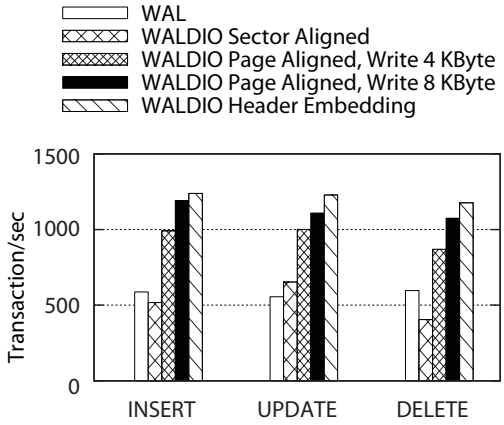


Figure 13: Page Aligning Schemes: Sector aligning, 4 KByte aligning, 8 KByte Aligning vs. Header Embedding (Full Sync)

We perform each of INSERT, UPDATE and DELETE operations 1,000 times and measure the performance. We put everything together in Fig. 14. The results in Fig. 14 are categorized into five groups: stock SQLite journal modes, LS-MVBT, WALDIO with Full Sync option, WALDIO with persistent direct IO and WALDIO with Normal Sync option. In stock SQLite journal modes, WAL mode yields the best performance (587 insert/sec). With LS-MVBT, SQLite yields 1083 insert/sec performance. With LS-MVBT, the SQLite performance increases by 80% from the WAL mode. In all these journals modes, SQLite issues fdatasync() after every log-commit.

In Full Sync option, WALDIO achieves 1219 insert/sec in the absence of Group Sync. With Group Synch with 16 KByte frame buffer, WALDIO performance leaps to 2729 insert/sec. It corresponds to 4.6× performance

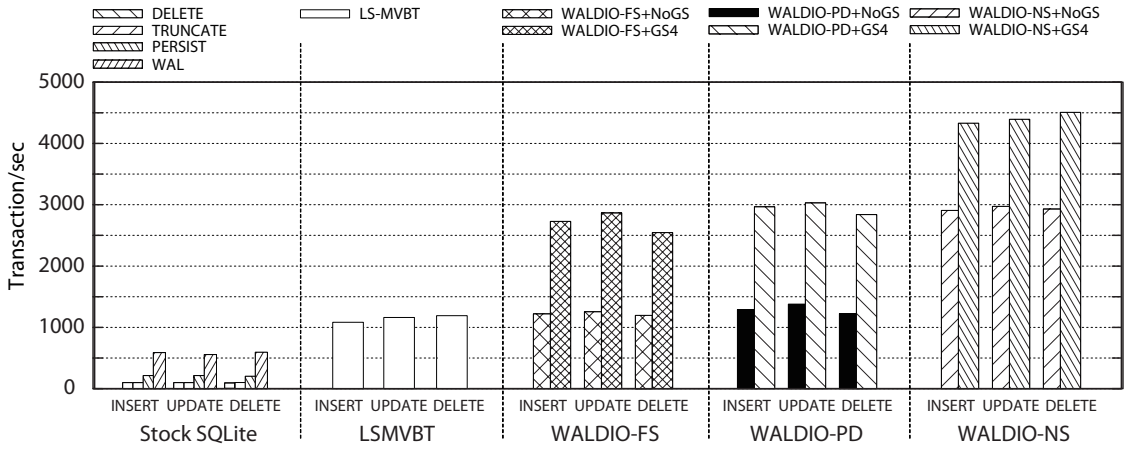


Figure 14: Performance Summary, NoGS: without Group Synch, GS4: Group Sync size = 4 pages

against stock WAL mode and 2.5× performance against LS-MVBT, respectively. Group Sync improves the performance by 120% from 1219 insert/sec to 2729 insert/sec. Group Synch is successful in eliminating the overhead of guaranteeing the durability.

With persistent DIO, WALDIO performance increases by 10% against WALDIO with Full Sync option. Bypassing the writeback cache at the storage device brings significant improvement. The performance under persistent DIO corresponds to 5.1 × performance against stock WAL mode and 2.8× performance against LS-MVBT, respectively.

In Normal Sync, the individual DIO based log-commit operations are relieved from the burden of calling the expensive fdatasync(). With Group Synch with 16 KByte frame buffer, WALDIO achieves 4332 insert/sec. The performance increases by more than 35% against the case where individual log-commits are persistently written to the storage surface; from 2967 insert/sec (WALDIO-PD) to 4332 insert/sec (WALDIO-NS). The WALDIO exhibits dramatic 7.4× and 4.0× performance compared against the WAL and the LS-MVBT, respectively. DELETE and UPDATE operations exhibit the similar performance gain with the INSERT operation. Table 2 illustrates the performance numbers for individual modes.

6.4 IO Volume

We examine the total IO volume for performing 1,000 SQLite operations, insert, update and delete. Fig. 15 illustrates the result. We limit our discussion to insert operation due to the space limit. In rollback journal modes (DELETE, TRUNCATE and PERSIST), as much as total 90 MByte is written to disk and 60% of which are for EXT4 journal writes. Via using WAL mode, the total page writes decreases to 29 MByte. LS-MVBT further de-

Journal Mode	INS	UPD	DEL
DELETE	98	97	96
TRUNCATE	99	98	97
PERSIST	213	212	203
WAL	587	556	596
LS-MVBT	1083	1161	1191
WALDIO-FS + NoGS	1219	1254	1197
WALDIO-FS + GS	**2729**	**2867**	**2546**
WALDIO-PD + NoGS	1290	1380	1224
WALDIO-PD + GS	**2967**	**3030**	**2839**
WALDIO-NS + NoGS	2907	2973	2930
WALDIO-NS + GS	**4332**	**4395**	**4507**

Table 2: SQLite performance (WALDIO-FS: WALDIO with Full Sync, WALDIO-PD: WALDIO with Persistent Direct IO, WALDIO-NS: WALDIO with Normal Sync, NoGS: Without Group Synch, GS: Group Synch with 4 pages)

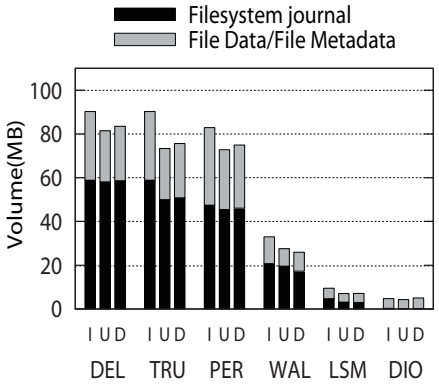


Figure 15: IO volume for 1000 INSERTs, DEL: DELETE, TRU: TRUNCATE, PER: PERSIST, WAL: WAL, LSM: LS-MVBT, DIO: WALDIO

creases the write volume to 7 MByte. In WALDIO, SQLite generates 4.6 MByte in executing an INSERT operation 1000 times. Compared to WAL mode, the total volume decreases to 1/6 from 29 MByte to 4.6 MByte.

Limited erase/write cycle of the NAND Flash storage is one of the main governing factors for the lifespan and the performance of the smartphone. The SQLite is responsible for dominant fraction of entire IO volume written to the storage. Reducing the IO volume to 1/6, the WALDIO technique can potentially allow the smartphone vendors to adopt the NAND Flash device with smaller Erase/Write cycle, e.g. TLC NAND Flash or NAND device with finer process technology, as the storage for their smartphone.

7 Related Work

While not everybody entirely agrees [35], the performance of the smartphone is governed by the performance of the storage device, not by the performance of the airlinks [23]. In Android, it is reported that more than 70% page writes generated by the smartphone application are for filesystem journal and dominant fraction of which are generated by SQLite DBMS [32]. The excessive filesystem journaling activity is due to the fact that the SQLite maintains a separate rollback journal file and synchronizes the every update in the rollback journal file via fsync() [19]. Tizen [16] also suffers from JOJ anomaly [26].

Jeong et. al. applied various IO optimization techniques, e.g. WAL mode, F2FS [30], external journaling and polling based IO and achieved 300% performance improvement against stock Android IO stack with DELETE journal mode [19]. Shen et. al. modified the EXT4 journal module and achieved 7% performance improvement against WAL mode [38]. Kim et. al. proposed to use LS-MVBT (Multiversion B-tree with Lazy Split) instead of B-tree in SQLite database [27]. LS-MVBT weaves the crash recovery information into the database file so that SQLite does not have to maintain separate file for crash recovery. LS-MVBT brings 80% performance gain against WAL mode in SQLite.

There are a number of benchmark programs for Android IO performance [12, 18, 2]. Kim et. al. [25] proposed to maintain the EXT4 journal region at NVRAM and to exploit its byte-granularity accessibility. Lue et. al. proposed to maintain the SQLite rollback journal file at DRAM [35] in the smartphone. Chidambaram et. al. proposed OPTFS to reduce the fsync() overhead involved in EXT4 journaling [7]. Kang et. al. proposed a transactional API for block device so that filesystem operations are free from the journaling overhead [20]. Piernas et. al. proposed to maintain the data and the metadata on the different blocks and maintains only single copy of metadata [36].

8 Conclusion

In this work, we successfully resolve the Journaling of Journal Anomaly in Android IO stack. We remove the root cause for excessive IO behavior in Android IO stack: the filesystem journaling. We develop a novel SQLite journal mode, WALDIO. In WALDIO mode, SQLite uses direct IO for log-commit operation so that it does not entail the expensive filesystem journaling. We develop *Preallocation with Explicit Journaling*, *Header Embedding* and *Group Synch* to enable the SQLite to exploit the direct IO semantics without compromising the filesystem integrity optimizing its performance for NAND Flash storage. The proposed features are implemented on the commercially available smartphone. WALDIO achieves as much as 7.4× increase against WAL mode and as much as 4.0× increase against LS-MVBT, respectively. With WALDIO mode, SQLite generates only 1/6 of the IO volume generated by SQLite in WAL mode.

Despite the dramatic improvement, WALDIO mode does not cost any major changes on the existing interface definitions of SQLite or of the filesystem, nor the introduction of the new ones. It is achieved by the minimal set of right modifications.

The contribution of this work should be viewed not only from the performance perspective but also from the NAND Flash endurance point of view. We carefully believe that via decreasing the IO volume generated by SQLite to 1/6, WALDIO can make the TLC NAND Flash not an infeasible choice for storage device in Android platform. Adoption of TLC NAND Flash in Android device can significantly reduce the cost of the smartphone and can make it available to wider community in the world.

9 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and feedback. Special thanks go to our shepherd Theodore Ts'o whose constructive comment and advice have made our work further mature and rigorous. We also would like to thank Yongseok Jo at EFTECH, Dongjun Shin, Seunghwan Hyun, Dongil Park and Heegyu Kim at Samsung Electronics for their advice on revising this paper. Finally, we like to thank our colleague Seongjin Lee for his help in preparing the manuscript. This work is sponsored by IT R&D program from MKE/KEIT (No. 10041608, Embedded system Software for New-memory based Smart Device) and by ICT R&D program of MSIP/IITP (No.1I2221-14-1005).

References

[1] emmc electrical standard 5.0. http://www.jedec.org/sites/default/files/docs/JESD84-B50.pdf/.

[2] http://www.antutu.com.

[3] Mysql homepage. http://www.mysql.com/.

[4] BOUGANIM, L., JONSSON, B., AND BONNET, P. uFLIP: Understanding Flash IO Patterns. In *Proc. of CIDR 2009* (Asilomar, CA, USA, Jan 2009).

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[6] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The rio file cache: Surviving operating system crashes. In *Proc. of ASPLOS 1966* (Cambridge, MA, USA, Sep 1996).

[7] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proc. of ACM SOSP 2013* (Farminton, PA, USA, 2013).

[8] CHINAFLASHMARKET.COM. 2013 NAND Flash Market annual report.

[9] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD 1984* (Boston, MA, USA, 1984).

[10] EFFELSBERG, W., AND HAERDER, T. Principles of Database buffer management. *ACM Trans. on Database Systems 9*, 4 (Dec. 1984), 560–595.

[11] GAWLICK, D., AND KINKADE, D. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull. 8*, 2 (1985), 3–10.

[12] GINGRICH, A. The Great Android Police Storage Benchmark: 11 Modern Devices Compared In 13 Tests. http://www.androidpolice.com/tags/rl-benchmark/.

[13] GRANCHER, E. Oracle and storage IOs, explanations and experience at CERN. In *Proc. of JPCS CHEP 2009* (Prague, Czech, Mar 2010).

[14] GU, W., KALBARCZYK, Z., IYER, R. K., AND YANG, Z. Characterization of linux kernel behavior under errors. In *Proc. of DSN 2003* (San Francisco, CA, USA, Jun 2003).

[15] HAGMANN, R. Reimplementing the Cedar File System Using Logging and Group Commit. *SIGOPS Oper. Syst. Rev. 21*, 5 (Nov. 1987), 155–162.

[16] HTTP://WWW.TIZEN.ORG.

[17] IYER, S., AND DRUSCHEL, P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of ACM SOSP 2001* (Banff, Alberta, Canada, Oct 2001).

[18] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Framework for Analyzing Android I/O Stack Behavior: From Generating the Workload to Analyzing the Trace. *Future Internet 5*, 4 (2013), 591–610.

[19] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC 2013* (San Jose, CA, USA, Jun 2013).

[20] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: Transactional FTL for SQLite Databases. In *Proc. of ACM SIGMOD 2013* (New York, NY, USA, Jun 2013).

[21] KIM, B., KANG, D. H., MIN, C., AND EOM, Y. I. Understanding implications of trim, discard, and background command for emmc storage device. In *Proc. of IEEE GCCE 2014* (Tokyo, Japan, Oct 2014).

[22] KIM, D., AND KANG, S. Partial page buffering for consumer devices with flash storage. In *Proc. of IEEE ICCE-Berlin* (Berlin, Germany, 2013).

[23] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proc. of USENIX FAST 2012* (San Jose, CA, USA, Feb 2012).

[24] KIM, H., AND AHN, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proc. of USENIX FAST 2008* (San Jose, CA, USA, Feb 2008).

[25] KIM, J., MIN, C., AND EOM, Y. I. Reducing Excessive Journaling Overhead with Small-Sized NVRAM for Mobile Devices. *IEEE Transactions on Consumer Electronics 6*, 2 (June 2014).

[26] KIM, M., LEE, S., AND WON, Y. IO Workload Characterization Of Tizen Based Consumer Electronics. In *Proc. of IEEE ISCE 2014* (Jeju, Korea, June 2014).

[27] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proc. of USENIX FAST 2014* (Santa Clara, CA, Feb 2014).

[28] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the Linux virtual machine monitor. In *Proc. of the Linux Symposium* (Ottawa, Ontario, Canada, Jun 2007).

[29] LANG, T., WOOD, C., AND FERNANDEZ, E. B. Database Buffer paging in virtual storage systems. *ACM Trans. on Database Systems 2*, 4 (Dec. 1977), 339–351.

[30] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proc. of USENIX FAST 2015* (San Jose, CA, US, Feb 2015).

[31] LEE, K. Mobile Benchmark Tool (MOBIBENCH). https://github.com/ESOS-Lab/Mobibench.

[32] LEE, K., AND WON, Y. Smart Layers and Dumb Result: IO Characterization of an Android-based Smartphone. In *Proc. of EMSOFT 2012* (Tampere, Finland, Oct 2012).

[33] LI, D., LIAO, X., JIN, H., ZHOU, B., AND ZHANG, Q. A New Disk I/O Model of Virtualized Cloud Environment. *IEEE Transactions on Parallel and Distributed Systems 24*, 6 (June 2013), 1129–1138.

[34] LIU", Z. vfs: add falloc_fl_no_hide_stale flag in fallocate. http://patchwork.ozlabs.org/patch/153251/.

[35] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi Non-Volatile RAM for Low Overhead Persistency Enforcement in Smartphones. In *Proc. of USENIX HotStorage 2014* (Philadelphia, PA, USA, Jun 2014).

[36] PIERNAS, J., CORTES, T., AND GARCÍA, J. M. DualFS: A New Journaling File System Without Meta-data Duplication. In *Proc. of ICS 2002* (New York, NY, USA, 2002).

[37] "REDHAT". "performance of trim command on ext4 filesystem".

[38] SHEN, K., PARK, S., AND ZHU, M. Journaling of Journal Is (Almost) Free. In *Proc. USENIX FAST 2014* (Santa Clara, CA, Feb 2014).

[39] SILBERSCHATZ, A. S., KORTH, H. F., AND SUDARSHAN, S. Database System Concepts. McGraw-Hill.

[40] "SQLITE". www.sqlite.org/famous.html.

[41] TRENDFORCE. Global mobile dram revenue rises 6%, Nov 2014. http://www.dramexchange.com/WeeklyResearch/Post/5/3904.html.

SpanFS: A Scalable File System on Fast Storage Devices

Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma and Jinpeng Huai

SKLSDE Lab, Beihang University, China

{*kangjb, woty, yuwr, dulian, mashuai*}*@act.buaa.edu.cn*, *zblgeqian@gmail.com*, *huaijp@buaa.edu.cn*

Abstract

Most recent storage devices, such as NAND flash-based solid state drives (SSDs), provide low access latency and high degree of parallelism. However, conventional file systems, which are designed for slow hard disk drives, often encounter severe scalability bottlenecks in exploiting the advances of these fast storage devices on many-core architectures. To scale file systems to many cores, we propose SpanFS, a novel file system which consists of a collection of micro file system services called domains. SpanFS distributes files and directories among the domains, provides a global file system view on top of the domains and maintains consistency in case of system crashes.

SpanFS is implemented based on the Ext4 file system. Experimental results evaluating SpanFS against Ext4 on a modern PCI-E SSD show that SpanFS scales much better than Ext4 on a 32-core machine. In micro-benchmarks SpanFS outperforms Ext4 by up to 1226%. In application-level benchmarks SpanFS improves the performance by up to 73% relative to Ext4.

1 Introduction

Compared to hard disk drives (HDDs), SSDs provide the opportunities to enable high parallelism on many-core processors [9, 15, 29]. However, the advances achieved in hardware performance have posed challenges to traditional software [9, 27]. Especially, the poor scalability of file systems on many-core often underutilizes the high performance of SSDs [27].

Almost all existing journaling file systems maintain consistency through a centralized journaling design. In this paper we focus on the scalability issues introduced by such design: (1) The use of the centralized journaling could cause severe contention on *in-memory* shared data structures. (2) The transaction model of the centralized journaling serializes its internal I/O actions *on devices* to ensure correctness, such as committing and checkpointing. These issues will sacrifice the high parallelism provided by SSDs. An exhaustive analysis of the scalability bottlenecks of existing file systems is presented in Section 2 as the motivation of our work.

Parallelizing the file system service is one solution to file system scalability. In this paper, we propose SpanFS, a novel journaling file system that replaces the centralized file system service with a collection of independent micro file system services, called *domains*, to achieve scalability on many-core. Each domain performs its file system service such as data allocation and journaling independently. Concurrent access to different domains will not contend for shared data structures. As a result, SpanFS allows multiple I/O tasks to work in parallel without performance interference between each other.

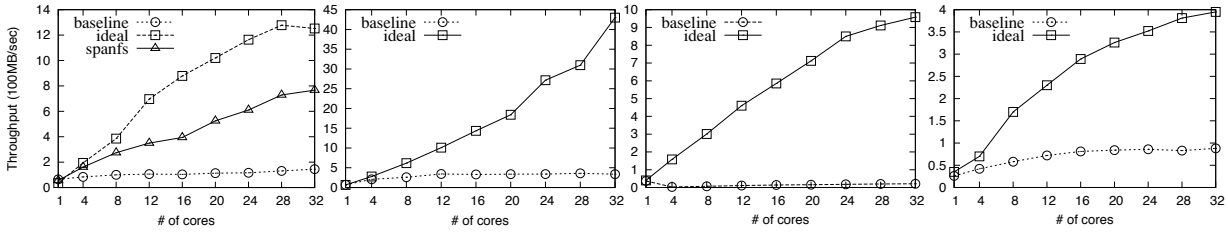
Apart from performance, consistency is another key aspect of modern file systems. Since each domain is capable of ensuring the consistency of the on-disk structures that belong to it, the key challenge to SpanFS is to maintain crash consistency on top of multiple domains. SpanFS proposes a set of techniques to distribute files and directories among the domains, to provide a global file system view on top of the domains and to maintain consistency in case of system crashes.

We have implemented SpanFS based on the Ext4 file system in Linux kernel 3.18.0 and would demonstrate that SpanFS scales much better than Ext4 on 32 cores, thus bringing significant performance improvements. In micro-benchmarks, SpanFS outperforms Ext4 by up to 1226%. In application-level benchmarks SpanFS improves the performance by up to 73% relative to Ext4.

The rest of the paper is organized as follows. Section 2 analyzes the scalability issues of existing file systems. Section 3 presents the design and implementation of SpanFS. Section 4 shows the performance result of SpanFS. We relate SpanFS to previous work in Section 5 and present the conclusion and future work in Section 6.

2 Background and Motivation

Most modern file systems scale poorly on many-core processors mainly due to the *contention* on shard data structures in memory and *serialization* of I/O actions on device. Our previous work [27] has identified some lock bottlenecks in modern file systems. We now provide an in-depth analysis of the root causes of the poor scalability. We first introduce the file system journaling mechanism to facilitate the scalability analysis. Then, through a set of experiments we will 1) show the scalability issues in existing modern file systems, 2) identify the scalability bottlenecks and 3) analyze which bottlenecks can be eliminated and which are inherent in the centralized file system design.



(a) Ext4 (b) XFS (c) Btrfs (d) ZFS

Figure 1: **Scalability Evaluation.** *We carry out this experiment in Linux 3.18.0 kernel on a RAM disk. We preallocate all the pages of the RAM disk to avoid contention within the RAM disk for the baselines and SpanFS. The total journaling sizes of SpanFS, Ext4, XFS and the ideal file systems based on Ext4 and XFS are all set to 1024 MB, respectively. For ZFS, we compile the recently released version (0.6.3) on Linux. This test is performed on a 32-core machine. For some lines the better than linear speedup is probably due to the Intel EIST technology.*

Ext4				XFS			
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)	Percent	Lock Name	Bounces	Total Wait Time (Avg. Wait Time)	Percent
journal->j_wait_done_commit	11845 k	1293 s (103.15 µs)	27%	cil->xc_push_lock	8019 k	329 s (37.26 µs)	13.8%
journal->j_list_lock	12713 k	154 s (11.34 µs)	3.2%	iclog->ic_force_wait	2188 k	87.4 s (39.94 µs)	3.7%
journal->j_state_lock-R	1223 k	7.1 s (5.19 µs)	0.1%	cil->xc_ctx_lock-R	1136 k	80.1 s (70.02 µs)	3.4%
journal->j_state_lock-W	956 k	4.3 s (4.29 µs)	0.09%	pool->lock	3673 k	34.1 s (9.28 µs)	1.4%
zone->wait_table	925 k	3.1 s (3.36 µs)	0.06%	log->l_icloglock	1555 k	25.8 s (16.18 µs)	1%

Table 1: **The top 5 hottest locks.** *We show the top 5 hottest locks in the I/O stack when running 32 Sysbench instances. These numbers are collected in a separated kernel with lock stat compiled. As* lock stat *introduces some overhead, the numbers does not accurately represent the lock contention overhead in Figure 1. "Bounces" represents the number of lock bounces among CPU cores. We calculate the percent of the lock wait time in the total execution time by dividing the lock wait time divided by the number of instances (32) by the total execution time.*

2.1 File System Journaling

Our discussion is based on the Ext3/4 journaling mechanism [39], which adopts the group commit mechanism [23] for performance improvements. Specifically, there is only one running transaction that absorbs all updates and at most one committing transaction at any time [39]. As a result, one block that is to be modified in the OS buffer does not need to be copied out to the journaling layer unless that block has already resided within the committing transaction, which largely reduces the journaling overhead caused by dependency tracking [39].

Ext4 [13] adopts JBD2 for journaling. For each update operation Ext4 starts a JBD2 handle to the current running transaction to achieve atomicity. Specifically, Ext4 passes the blocks (refer to metadata blocks in ordered journaling mode) to be modified associated with the handle to the JBD2 journaling layer. After modifying these blocks, Ext4 stops the handle and then the running transaction is free to be committed by the JBD2 journaling thread. These modified block buffers will not be written back to the file system by the OS until the running transaction has been committed to the log [39]. For simplicity, we refer to the above process as wrapping the blocks to be modified in a JBD2 handle in the rest of the paper.

2.2 Scalability Issue Analysis

We use Sysbench [1] to generate update-intensive workloads to illustrate the scalability bottlenecks. Multiple single-threaded benchmark instances run in parallel, each of which issues 4KB sequential writes and invokes *fsync()* after each write. Each instance operates over 128 files with a total write traffic of 512MB. We vary the number of running instances from 1 to 32 and the number of used cores is euqal to the number of instances. We measure the total throughput.

Four file systems are chosen as baseline for analysis: Ext4, XFS [38], Btrfs [35] and OpenZFS [2], of which Ext4 and XFS are journaling file systems while Btrfs and ZFS are copy-on-write file systems. An ideal file system is set up by running each benchmark instance in a separated partition (a separated RAM disk in this test) managed by the baseline file system, which is similar to the disk partition scenario in [31]. It is expected to achieve linear scalability since each partition can perform its file system service independently.

Figure 1 shows that all the four baseline file systems scale very poorly on many-core, resulting in nearly horizontal lines. The "ideal" file systems exhibit near-linear scalability. We add the result of SpanFS with 16 domains in Figure 1(a), which brings a performance improvement of 4.29X in comparison with the stock Ext4 at 32 cores.

2.3 Scalability Bottleneck Analysis

To understand the sources of the scalability bottlenecks, we collect the lock contention statistics using *lock stat* [7]. Due to space limitation, we show the statistics on top

SpanFS-16		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
journal->j_wait_done_commit	3333 k	38.5 s (11.13 µs)
journal->j_state_lock-R	4259 k	16.6 s (3.70 µs)
journal->j_state_lock-W	2637 k	10.5 s (3.84 µs)
journal->j_list_lock	5042 k	10.2 s (2.00 µs)
zone->wait_table	226 k	0.5 s (2.06 µs)

Table 2: **The top 5 hottest locks.** *The top 5 hottest locks in the I/O stack when running the sysbench benchmark on SpanFS.*

hottest locks of Ext4 and XFS in Table 1. Table 2 shows the top 5 hottest locks when running the same benchmark on SpanFS at 32 cores. Ext4 and XFS spend substantial wait time acquiring hot locks and the average wait time for these hot locks is high. In contrast, SpanFS reduces the total wait time of the hot locks by around 18X (76 s vs 1461 s).

Btrfs also has a few severely contended locks, namely eb->write_lock_wq, btrfs-log-02 and eb->read_lock_wq. The total wait time of these hot locks can reach as much as 14476 s, 5098 s and 2661 s respectively. We cannot collect the lock statistics for ZFS using *lock stat* due to the license compatible issue.

2.3.1 Contention on shared data structures

Now we look into Ext4 and discuss the causes of scalability bottlenecks in depth, some of which are also general to other file systems. As is well known, shared data structures can limit the scalability on many-core [10, 11]. JBD2 contains many shared data structures, such as the journaling states, shared counters, shared on-disk structures, journaling lists, and wait queues, which can lead to severe scalability issues.

(a) Contention on the journaling states. The journaling states are frequently accessed and updated, and protected by read-write lock (i.e., *j_state_lock*). The states may include the log tail and head, the sequence numbers of the next transaction and the most recently committed transaction, and the current running transaction's state. The lock can introduce severe contention. The RCU lock [33] and Prwlock [30] are scalable for read-mostly workloads while JBD2 have many writes to these shared data structures in general as shown in Table 1. Hence, they are not effective to JBD2.

(b) Contention on the shared counters. The running transaction in JBD2 employs atomic operations to serialize concurrent access to shared counters, such as the number of current updates and the number of buffers on this transaction, which can limit the scalability. Sometimes, JBD2 needs to access the journaling states and these shared counters simultaneously, which can cause even more severe contention. For instance, to add updates to the running transaction, JBD2 needs to check whether there is enough log free space to hold the running transaction by reading the number of the buffers on the running transaction and on the committing transaction and reading the log free space. We have confirmed the contention on shared counters using *perf*, which will partly cause the JBD2 function *start_this_handle* to account for 17% of the total execution time when running 32 Filebench Fileserver instances. Adopting per-core counters such as sloppy counter [11] and Refcache [18] will introduce expensive overhead when reading the true values of these counters [18].

(c) Contention on the shared on-disk structures. Although the on-disk structures of Ext4 are organized in the form of block groups, there is also contention on shared on-disk structures such as block bitmap, inode bitmap and other metadata blocks during logging these blocks. These were not manifested in Table 1 since *lock stat* does not track the bit-based spin locks JBD2 uses.

(d) Contention on the journaling lists. JBD2 uses a spin lock (i.e., *j_list_lock*) to protect the transaction buffer lists and the checkpoint transaction list that links the committed transactions for checkpointing, which can sabotage scalability. Replacing each transaction buffer list with per-core lists may be useful to relieve the contention. However, using per-core lists is not suitable for the checkpoint transaction list as JBD2 needs to checkpoint the transactions on the list in the order that the transactions are committed to the log.

(e) Contention on the wait queues. JBD2 uses wait queues for multi-thread cooperation among client threads and the journaling thread, which will cause severe contention when a wait queue is accessed simultaneously. The wait queue is the most contended point in Ext4 during our benchmarking. Simply removing this bottleneck, i.e. per-core wait queue [30], cannot totally scale Ext4 as other contention points will rise to become the main bottlenecks, such as *j_state_lock* and shared counters. The most contended point in XFS is not the wait queue as shown in Table 1. Hence, we need a more thorough solution to address the scalability issues.

2.3.2 Serialization of internal actions

The centralized journaling service usually needs to serialize its internal actions in right order, which also limits the scalability. Here we give two examples.

Parallel commit requests are processed sequentially in transaction order by the journaling thread [39], which largely sacrifices the high parallelism provided by SSDs. Enforcing this order is necessary for correctness when recovering the file system with the log after a crash due to the dependencies between transactions [16].

Another example is when the free space in the log is not enough to hold incoming update blocks. JBD2 performs a checkpoint of the first committed transaction on the checkpoint transaction list to make free space. Parallel checkpoints also have to be serialized in the order that

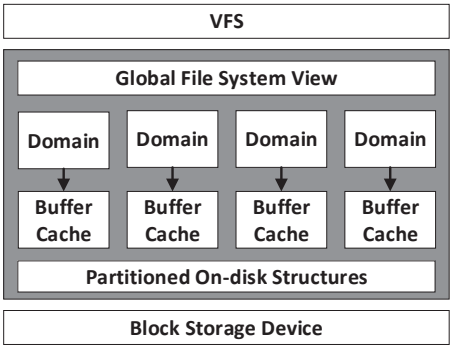


Figure 2: **SpanFS Architecture.**

transactions are committed to the log.

2.3.3 Summary

We analyzed the scalability bottlenecks of Ext4, which are mainly caused by the contention on the shared data structures in memory and the serialization of internal actions on devices. The use of shared data structures are inherent in the centralized journaling design. The serialization of journaling would also need to access and update the shared data structures, such as log head and tail.

To address the scalability bottlenecks, file systems should be restructured to reduce the contention on shared data structures and to parallelize the file system service.

3 Design and Implementation

We present the design and implementation of SpanFS and introduce the key techniques to provide a global file system view and crash consistency.

3.1 SpanFS Architecture

Figure 2 shows the architecture of SpanFS, which consists of multiple micro file system services called domains. SpanFS distributes files and directories among multiple domains to reduce contention and increase parallelism within the file system. Each domain has its independent on-disk structures, in-memory data structures and kernel services (e.g., the journaling instance JBD2) at runtime. As there is no overlap among the domains' on-disk blocks, we allocate a dedicated buffer cache address space for each domain to avoid the contention on the single device buffer cache. As a result, concurrent access to different domains will not cause contention on shared data structures. Each domain can do its journaling without the need of dependency tracking between transactions and journaled buffers that belong to different domains, enabling high parallelism for logging, committing and checkpointing.

SpanFS provides a global file system view on top of the domains by building global hierarchical file system namespace and also maintains global consistency in case of system crashes.

3.2 Domain

The domain is the basic independent function unit in SpanFS to perform the file system service such as data allocation and journaling. During mounting, each domain will build its own in-memory data structures from its on-disk structures and start its kernel services such as the JBD2 journaling thread. In the current prototype SpanFS builds the domains in sequence. However, the domains can be built in parallel by using multiple threads.

3.2.1 SpanFS on-disk layout

In order to enable parallel journaling without the need of dependency tracking, we partition the device blocks among the domains. SpanFS creates the on-disk structures of each domain on the device blocks that are allocated to the domain. The on-disk layout design of each domain is based on the Ext4 disk layout [5]: each domain mainly has a super block, a set of block groups, a root inode and a JBD2 journaling inode (i.e., log file).

Initially, the device blocks are evenly allocated to the domains. Our architecture allows to adjust the size of each domain online on demand in the unit of block groups. Specifically, the block groups in one domain can be reallocated to other domains on demand. To this end, we should store a block group allocation table (BAT) and a block group bitmap (BGB) on disk for each domain. The BGB is maintained by its domain and is used to track which block group is free by the file system. When the free storage space in one domain drops to a predefined threshold, the file system should reallocate the free block groups in other domains to this domain. To avoid block group low utilization each domain should allocate inodes and blocks from the block groups that have been used as far as possible. As the global block group reallocation can cause inconsistent states in case of crashes, we should create a dedicated journaling instance to maintain the consistency of the BATs. Each domain should first force the dedicated journaling to commit the running transaction before using the newly allocated block groups. This ensures the reallocation of block groups to be persisted on disk and enables the block groups to be correctly allocated to domains after recovery in case of crashes. We leave the implementation of online adjusting of each domain's size on demand as our future work.

In our current implementation SpanFS only supports static allocation of block groups. Specifically, we statically allocate a set of contiguous blocks to each domain when creating the on-disk structures by simply storing the first data block address and the last data block address in each domain's super block. Each domain's super block is stored in its first block group. In order to load all the domains' super blocks SpanFS stores the next super block address in the previous super block. Each domain adopts the same policy as Ext4 for inode and block allo-

cation among its block groups.

3.2.2 Dedicated buffer cache address space

The Linux operating system (OS) adopts a buffer cache organized as an address space radix tree for each block device to cache recently accessed blocks and use a spin lock to protect the radix tree from concurrent inserts. Meanwhile, the OS uses another spin lock to serialize concurrent access to each cache page's buffer list. As a result, when multiple domains access the single underlying device simultaneously, the above two locks will be contended within the buffer cache layer .

The device block range of each domain does not overlap with those of other domains and the block size is the same with the page size in our prototype. Concurrent accesses to different domains should not be serialized in the buffer cache layer as they can commute [19, 18].

We leverage the Linux OS block device architecture to provide a dedicated buffer cache address space for each domain to avoid lock contention. The OS block layer manages I/O access to the underlying device through the block device structure, which can store the inode that points to its address space radix tree and pointers to the underlying device structures such as the device I/O request queue and partition information if it is a partition. SpanFS clones multiple block device structures from the original OS block device structure and maps them to the same underlying block device. SpanFS assigns a dedicated block device structure to each domain during mounting so that each domain can have its own buffer cache address space.

Under the block group reallocation strategy, the device block range of each domain may be changed over time. We should remove the pages in the domain's buffer cache address space corresponding to the block groups that are to be reallocated to other domains. This process can be implemented with the help of the OS interface (*invalidate_mapping_pages()*). We leave the implementation of buffer cache address space adjusting as our future work.

3.3 Global Hierarchical Namespace

In order to distribute the global namespace, SpanFS chooses a domain as the root domain to place the global root directory and then scatters the objects under each directory among the domains. Specifically, SpanFS distributes the objects (files and directories) under each directory using a round-robin counter for the directory. The use of per directory counter can avoid global contention.

To support this distribution, SpanFS introduces three types of directory entries (dentries): normal dentry, shadow dentry and remote dentry, as illustrated in Figure 3. We call an object placed in the same domain with its parent directory a normal object. The domain where the parent directory lies is referred to as the local domain. SpanFS creates an inode and a *normal dentry* pointing

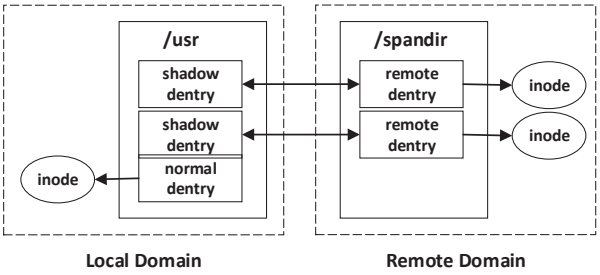


Figure 3: **The connection between domains.** *This figure illustrates how SpanFS distributes a object to a remote domain.*

to this inode under the parent directory in the local domain. We call an object placed in the different domain from its local domain a distributed object, the different domain is referred to as remote domain. SpanFS creates an inode and a *remote dentry* pointing this inode in the remote domain and then creates a *shadow dentry* under its parent directory in the local domain for look up. The reason why SpanFS creates a remote dentry and an inode rather than only an inode is due to consideration of maintaining global consistency, which will be described in Section 3.4. In this paper, the dentry and inode refer to the on-disk structures rather than the VFS structures.

To store remote dentries from other domains, each domain creates a set of special directories called *span directories* which are invisible to users. The number of span directories in each domain is set to 128 by default and the span directories are evenly allocated to each CPU core. When a thread is to create a remote dentry, SpanFS selects a span directory randomly from the ones allocated to the CPU core.

SpanFS constitutes the *bidirectional index* by embedding enough information in both the shadow dentry and remote dentry to make them point to each other. For each shadow dentry, SpanFS stores in the shadow dentry fields the remote dentry's domain ID, the inode number of the span directory where the remote dentry stays, the remote inode number and the name. For each remote dentry, SpanFS stores the shadow dentry's domain ID, the inode number of the parent directory where the shadow dentry lies, the remote inode number and the name.

3.4 Crash Consistency Model

As each domain is capable of ensuring the consistency of its on-disk structures in case of system crashes through journaling, the most critical challenge to SpanFS is how to maintain global consistency across multiple domains.

The following example will demonstrate the consistency challenges. To create a distributed object, one approach is to create an inode in the remote domain, and to add a dentry in the local domain which points to the remote inode, which can result in two possible inconsistent states in case of crashes. The first case is that the inode

reaches the device but the dentry is lost. Then the newly created inode becomes storage garbage silently, which cannot be reclaimed easily. Another case is that the dentry reaches the device but the inode fails. As a result the dentry points to a non-existent inode.

Another approach to address the above problem is adding dependency information to the running transactions of the two domains. However, this approach will end up with many dependencies between multiple domains, with the cost of dependency tracking and serializing the journaling actions among the domains.

Based on *bidirectional index*, SpanFS adopts two mechanisms *stale object deletion* and *garbage collection* to address the problem. As discussed in Section 3.3, SpanFS creates a remote dentry and inode in the remote domain and then adds a shadow dentry under the parent directory in the local domain. SpanFS wraps the blocks to be modified in two JBD2 handles of the remote domain and local domain to achieve atomicity in the two domains respectively.

The consistency of a distributed object is equivalent to the integrity of the bidirectional index: (1) A shadow dentry is valid only if the remote dentry it points to exists and points back to it too. (2) A remote object (the remote dentry and inode) is valid only if its shadow dentry exists and points to it.

3.4.1 Stale object deletion.

The stale object deletion will validate any shadow dentry by checking whether its remote dentry exists when performing *lookup* and *readdir*. Specifically, for a distributed object SpanFS first locks the span directory via the mutex lock, then looks up the remote dentry under the span directory using the embedded index information and unlocks the span directory in the end. If the remote dentry does not exist, SpanFS deletes the shadow dentry. Note that the parent directory would be locked by the VFS during the above process.

3.4.2 Garbage collection (GC)

The GC mechanism deals with the scenario when the remote dentry and inode exist while the shadow dentry is lost. Under this circumstance, the file system consistency is not impaired since the remote object can never be seen by applications. But the remote dentry and inode will occupy storage space silently and should be collected.

During mounting if SpanFS finds out that it has just gone through a system crash, SpanFS will generate a background garbage collection thread to scan the span directories. The GC thread verifies the integrity of each remote dentry in each span directory at runtime silently, and removes the remote objects without shadow dentries.

Two-phase validation. In order to avoid contention with the normal operations, the GC thread performs two-phase validation: the scan phase and the integrity validation phase. During the scan phase the background GC thread locks the span directory via the mutex lock, reads the dentries under it and then unlocks the span directory. Then the GC thread validates each scanned remote dentry's integrity. To avoid locking the span directory for a long time, the GC thread reads a small number of remote dentries (4 KB by default) each time until all the remote dentries have been scanned and validated.

3.4.3 Avoiding deadlocks and conflicts

To avoid deadlocks and conflicts, for all operations that involve the creation or deletion of a distributed object we should first lock the parent directory of the shadow dentry and then lock the span directory of the remote dentry. By doing so, SpanFS can guarantee that new remote objects created by the normal operations will not be removed by the background GC thread by mistake and can avoid any deadlocks.

During the integrity validation phase, the GC thread first locks the parent directory (read from the bidirectional index on the remote dentry) and then looks up the shadow dentry under it. If the shadow dentry is found and points to the remote dentry, the GC thread does nothing and unlocks the parent directory. Otherwise, the GC thread locks the span directory and then tries to delete the remote object. If the remote object does not exist, it might be deleted by the normal operation before the integrity validation phase. For such case, the GC thread does nothing. The GC thread unlocks the span directory and the parent directory in the end.

For normal operations such as *create()* and *unlink()*, SpanFS first locks the span directory of the remote dentry, then creates or deletes the remote object and the shadow dentry, and unlocks the span directory in the end. Note that the parent directory would be locked by the VFS during the above process. For *unlink()* the inode will not be deleted until its link count drops to zero.

3.4.4 Parallel two-phase synchronization

The VFS invokes *fsync()* to flush the dirty data and corresponding metadata of a target object to disk. As the dentry and its corresponding inode may be scattered on two domains, SpanFS should persist the target object and its ancestor directory objects, their shadow/remote dentries if distributed, along the file path.

In order to reduce the distributed *fsync()* latency, we propose a parallel two-phase synchronization mechanism: the committing phase and the validating phase. During the committing phase SpanFS traverses the target object and its ancestor directories except for the SpanFS root directory. For each traversed object, SpanFS wakes up the journaling thread in its parent directory's domain to commit the running transaction and then records the committing transaction id in its VFS inode's field. Note that if there does not exist a running transaction, SpanFS

does nothing. This situation may occur when the running transaction has been committed to disk by other synchronization actions such as periodic transaction commits in JBD2. Then, SpanFS starts to commit the target object. In the end SpanFS traverses the target object and its ancestor directories again to validate whether the recorded transaction commits have completed. If some of the commits have not completed, SpanFS should wait on the wait queues for their completion.

The synchronization mechanism utilizes JBD2 client-server transaction commit architecture. In JBD2, the client thread wakes up the journaling thread to commit the running transaction and then waits for its completion. In SpanFS, we leverage the journaling thread in each domain to commit the running transactions in parallel.

In order to avoid redundant transaction commits, we use flags (*ENTRY_NEW*, *ENTRY_COMMIT* and *ENTRY_PERSISTENT*) for each object to record its state. Ext4 with no journaling has the counterpart of our committing phase but does not have validating phase, which could potentially lead to inconsistencies.

During the committing phase SpanFS will clear the *ENTRY_NEW* flag of each traversed object. If cleared, SpanFS stops the committing phase. The *ENTRY_COMMIT* flag of the object would be set after the transaction has been committed. If not set, SpanFS would commit the transaction in its parent directory's domain and wait for the completion during the validating phase. During the validating phase, SpanFS will set the *ENTRY_PERSISTENT* flags of the traversed objects when all the recorded transaction commits have been completed. If set, SpanFS stops the validating phase.

3.4.5 Rename

The rename operation in Linux file systems tries to move a source object under the source directory to the destination object with the new name under the destination directory. SpanFS achieves atomicity of the rename operation through the proposed *ordered transaction commit mechanism*, which controls the commit sequence of the JBD2 handles on multiple domains for the rename operation. SpanFS ensures the commit order by marking each handle with *h_sync* flag, which would force JBD2 to commit the corresponding running transaction and wait for its completion when the handle is stopped.

For the case that the destination object does not exist, three steps are needed to complete a rename operation. Due to space limitation, we only demonstrate the case where the source object is a distributed object. The shadow dentry of the source object resides in Domain A. The inode of the source directory that contains the shadow dentry of the source object also resides in Domain A. The remote dentry and the inode of the source object resides in Domain B. The inode of the destination directory resides in Domain C. SpanFS starts a JBD2 handle for each step.

Step 1: SpanFS adds a new shadow dentry, which points to the remote dentry of the source object, to the destination directory in Domain C. If system crashes after this handle reaches the disk, the bidirectional index between the old shadow dentry and the remote dentry of the source object is still complete. The newly added dentry will be identified as stale under the destination directory and be removed at next mount.

Step 2: The remote dentry of the source object is altered to point to the newly added shadow dentry in Step 1. Then the bidirectional index between the old shadow dentry and the remote dentry of the source object becomes unidirectional while the bidirectional index between the new shadow dentry and the remote dentry is built. As long as the handle reaches disk, the old shadow dentry of the source object in Domain A is turned stale and the rename operation is essentially done.

Step 3: Remove the old shadow dentry of the source object under the source directory.

During the above process, JBD2 handles could be merged if they operate on the same domain. If the step needs to lock the span directory, it must start a new handle to avoid deadlocks, esp. step 2.

For the case that the destination object already exists, SpanFS first has the existing shadow dentry of the destination object tagged with *TAG_COMMON*, then adds another new shadow dentry tagged with both *TAG_NEWENT* and *TAG_COMMON* in the destination directory in Step 1. Moreover, two extra steps are needed to complete the rename: Step 4 to remove the inode and the remote dentry of the destination object and step 5 to delete the existing shadow dentry of the destination object and untag the newly added shadow dentry under the destination directory. As the existing shadow dentry has the same name as the newly added dentry under the destination directory, we use the tags to resolve conflicts in case of system crashes. Specifically, a dentry tagged with *TAG_COMMON* should be checked during *lookup()*. If there exist two tagged dentries with the same name under a directory, SpanFS will remove the one without integral bidirectional index. If their bidirectional indices are both integral, the one with *TAG_NEWENT* takes precedence and the other is judged as stale and should be removed.

3.5 Discussion

The approach introduced in this paper is not the only way to scale file systems. Another way to providing parallel file system services is running multiple file system instances by using disk partitions (virtual block devices in [26]), stacking a unified namespace on top of them and maintaining crash consistency across them, which is similar to Unionfs [42] in the architecture. We previously

adopt this approach as an extension to MultiLanes [27] to reduce contention inside each container [26]. However, this approach has several drawbacks: First, managing a number of file systems would induce administration costs [31]. Second, adjusting storage space among multiple file systems and partitions on demand also introduces management cost. Although we can leverage the virtual block device of MultiLanes to support storage space overcommitment, it comes with a little cost of lightweight virtualization [27]. Third, the cost of namespace unification will increase with the increasing number of partitions [42].

4 Evaluation

We evaluate the performance and scalability of SpanFS against Ext4 using a set of micro and application-level benchmarks.

4.1 Test Setup

All experiments were carried out on an Intel 32-core machine with four Intel(R) Xeon(R) E5-4650 processors (with the hyperthreading capability disabled) and 512 GB memory. Each processor has eight physical cores running at 2.70 GHZ. All the experiments are carried out on a Fusion-IO SSD (785 GB MLC Fusion-IO ioDrive). The experimental machine runs a Linux 3.18.0 kernel. We compile a separated kernel with *lock stat* enabled to collect the lock contention statistics.

We use 256 GB of the SSD for evaluation. We evaluate SpanFS with 16 domains and 4 domains in turn. We statically allocate 16 GB storage space to each domain for the 16 domain configuration and 64 GB storage space to each domain for the 4 domain configuration. For SpanFS with 16 domains, each domain has 64 MB journaling size, yielding a total journaling size of 1024 MB. To rule out the effects of different journaling sizes, the journaling sizes of both SpanFS with 4 domains and Ext4 are all set to 1024 MB, respectively. Both SpanFS and Ext4 are mounted in ordered journal mode unless otherwise specified.

Kernel Patch. The VFS uses a global lock to protect each super block's inode list, which can cause contention. We replace the super block's inode list with per-core lists and use per-core locks to protect them. We apply this patch to both the baseline and SpanFS.

4.2 Performance Results

4.2.1 Metadata-Intensive Performance

We create the micro-benchmark suite called *catd*, which consists of four benchmarks: *create*, *append*, *truncate* and *delete*. Each benchmark creates a number of threads performing the corresponding operation in parallel and we vary the number of threads from 1 to 32.

Create: Each thread creates 10000 files under its private directory.

Append: Each thread performs a 4 KB buffered write and a *fsync()* to each file under its private directory.

Truncate: Each thread truncates the appended 4 KB files to zero-size.

Delete: Each thread removes the 10000 truncated files.

We run the benchmark in the order of *create-append-truncate-delete* in a single thread and multiple threads concurrently. Figure 4 shows that SpanFS performs much better than Ext4 except for the *create* benchmark.

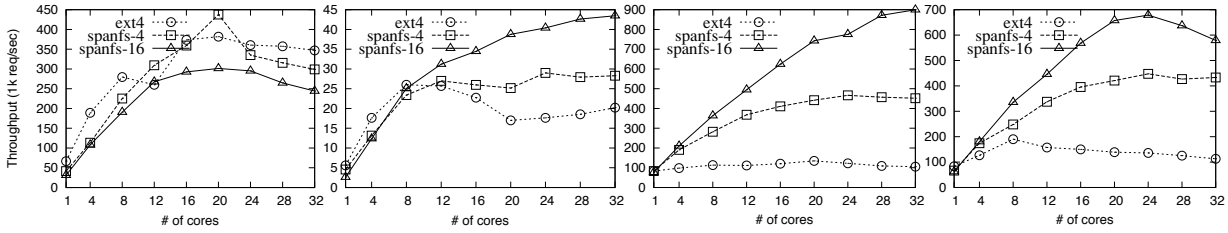
For the *create* benchmark SpanFS performs worse than Ext4 for two reasons: Ext4 has not encountered severe scalability bottlenecks under this workload, and SpanFS introduces considerable overhead as it needs to create two dentries for each distributed object. Ext4 is 113% and 42% faster than SpanFS with 16 domains at one core and at 32 cores, respectively. Note that the 4 domain configuration performs better than the 16 domain configuration mainly due to that the percentage of the distributed objects in SpanFS with 4 domains is lower.

Ext4		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
sbi->s_orphan_lock	478 k	534 s (1117.32 µs)
journal->j_wait_done_commit	845 k	100.4 s (112.10 µs)
journal->j_checkpoint_mutex	71 k	56.5 s (789.70 µs)
journal->j_list_lock	694 k	10.5 s (14.64 µs)
journal->j_state_lock-R	319 k	9.8 s (28.58 µs)
SpanFS-16		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
journal->j_checkpoint_mutex	27 k	15.1 s (557.96 µs)
inode_hash_lock	323 k	8.1 s (25.07 µs)
sbi->s_orphan_lock	124 k	4.3 s (34.51 µs)
journal->j_wait_done_commit	287 k	3.4 s (11.07 µs)
ps->lock (Fusionio driver)	789 k	2.4 s (2.87 µs)

Table 3: **The top 5 hottest locks**

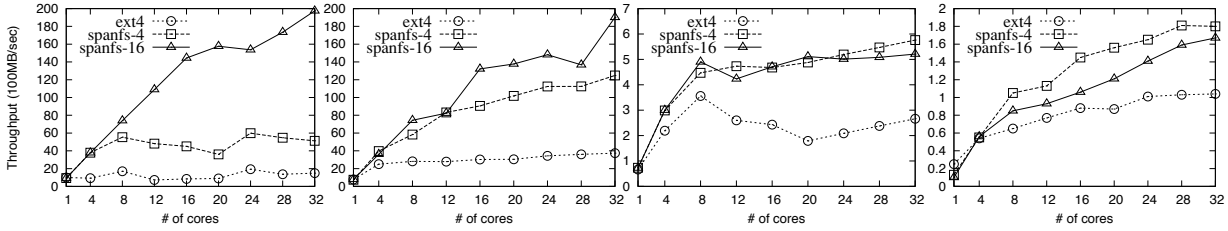
As shown in Figure 4, for the *append*, *truncate*, *delete* benchmark, SpanFS significantly outperforms Ext4 beyond a number of cores due to the reduced contention and better parallelism. As the *fsync()* in *append* may span several domains to persist the objects along the path and the *delete* benchmark involves the deletion of two dentries for each distributed object, there exists some overhead for these two benchmarks. Specifically, SpanFS is 113% and 33% slower than Ext4 for these two benchmarks at one single core. However, due to the reduced contention, SpanFS with 16 domains outperforms Ext4 by 1.15X, 7.53X and 4.13X at 32 cores on the *append*, *truncate* and *delete* benchmark, respectively.

To understand the performance gains yielded by SpanFS, we run the *catd* benchmark at 32 cores in a separated kernel with *lock stat* enabled. Table 3 shows that Ext4 spends substantial time acquiring the hottest locks during the benchmarking. In contrast, the total wait time



(a) create (b) append (c) truncate (d) delete

Figure 4: **Catd.** *This figure depicts the overall throughput (operations per second) with the benchmark create, append, truncate and delete, respectively.*



(a) sequential buffered write (b) random buffered write (c) direct write (d) synchronous write

Figure 5: **IOzone.** *This figure shows the total throughput of IOzone on SpanFS against Ext4 under sequential buffered writes, random buffered writes, sequential direct writes and sequential synchronous writes (open with O_SYNC), respectively.*

of the hot locks in SpanFS has been reduced by 20X.

4.2.2 Data-Intensive Performance

IOzone. The IOzone [3] benchmark creates a number of threads, each of which performs 4KB writes to a single file which ends up with 512 MB. Figure 5 shows that SpanFS scales much better than Ext4, leading to significant performance improvements. Specifically, SpanFS with 16 domains outperforms Ext4 by 1226%, 408%, 96% and 60% under the four I/O patterns at 32 cores, respectively. For direct I/O, Ext4 scales poorly due to the contention when logging the block allocation.

Sysbench. We run multiple single-threaded sysbench instances in parallel, each of which issues 4 KB writes. Each instance operates over 128 files with a total write traffic of 512 MB. Figure 6 shows that SpanFS scales well to 32 cores, bringing significant performance improvements. Specifically, SpanFS with 16 domains is 4.38X, 5.19X, 1.21X and 1.28X faster than Ext4 in the four I/O patterns at 32 cores, respectively.

4.2.3 Application-Level Performance

Filebench. We use Filebench [6] to generate application-level I/O workloads: the Fileserver and Varmail workloads. The Varmail workload adopts the parameter of 1000 files, 1000000 average directory width, 16 KB average file size, 1 MB I/O size and 16 KB average append size. The Fileserver workload adopts the parameter of 10000 files, 20 average directory width, 128 KB average file size, 1 MB I/O size and 16 KB average append size. We run multiple single-threaded Filebench instances in parallel and vary the number of instances from 1 to 32. Each workload runs for 60 s.

As shown in Figure 7(a) and Figure 7(b) , for the Fileserver and Varmail workloads SpanFS in all the two configurations scales much better than Ext4. SpanFS with 16 domains outperforms Ext4 by 51% and 73% under the Fileserver and Varmail workloads at 32 cores, respectively. We have also evaluated the performance of SpanFS against Ext4 in data journal mode under the Varmail workload. Figure 7(c) shows that SpanFS with 16 domains outperforms Ext4 by 88.7% at 32 cores.

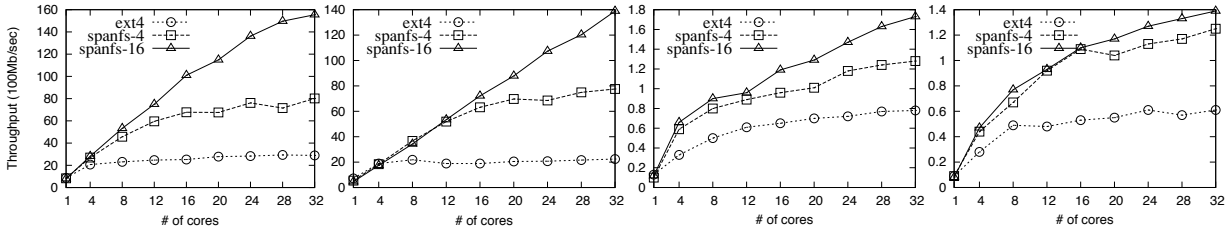
Dbench. We use Dbench [4] to generate I/O workloads that mainly consist of *creates, renames, deletes, stats, finds, writes, getdents* and *flushes*. We choose Dbench to evaluate SpanFS as it allows us to illuminate performance impact of the rename operation overhead on a realistic workload. We run multiple single-threaded Dbench instances in parallel.

Due to the overhead, SpanFS with 16 domains is 55% slower than Ext4 at one single core. However, as shown in Figure 7(d), due to the reduced contention and better parallelism SpanFS with 16 domains outperforms Ext4 by 16% at 32 cores.

4.2.4 Comparison with other file systems

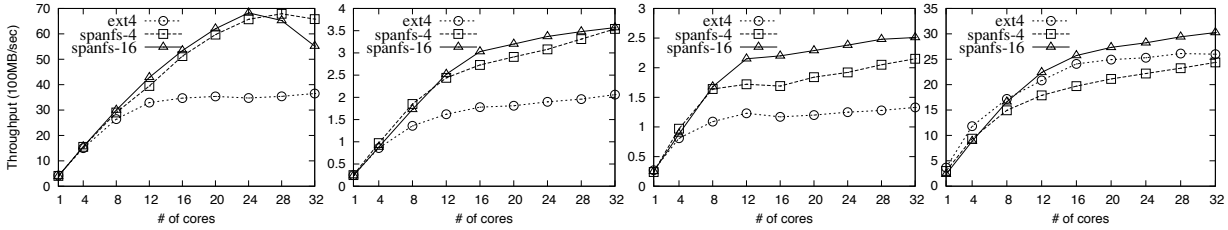
We make a comparison of SpanFS with other file systems on scalability using the Fileserver workload. Figure 8 (a) shows that SpanFS with 16 domains achieves much better scalability than XFS, Btrfs and ZFS.

We also make a comparison with MultiLanes+ [26], an extended version of our previous work [27]. As MultiLanes+ stacks a unified namespace on top of multiple virtual block devices, it comes at the cost of namespace unification. We evaluate SpanFS with 32 domains against MultiLanes+ with 32 disk partitions using the *create*



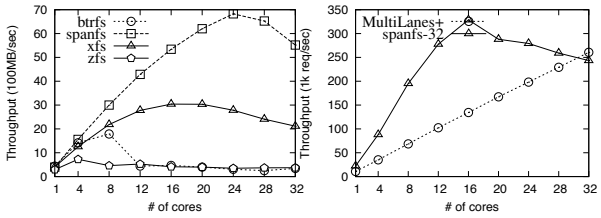
(a) sequential buffered write (b) random buffered write (c) synchronous write (d) random synchronous write

Figure 6: **Sysbench.** *This figure depicts the total throughput of sysbench on SpanFS against Ext4 under sequential buffered writes, random buffered writes, sequential synchronous writes and random synchronous writes, respectively. The first two buffered I/O patterns do not issue any* fsync() *while the synchronous I/O patterns issue a* fsync() *after each write.*



(a) fileserver (b) varmail (c) varmail (*data journal*) (d) dbench

Figure 7: **Filebench and Dbench.** *This figure depicts the total throughput of Filebench (Fileserver and Varmail) and Dbench on SpanFS against Ext4, respectively.*



(a) fileserver (b) create

Figure 8: **Comparison with other major file systems.** *The throughput of Filebench Fileserver on SpanFS against other three major file systems: XFS, Btrfs and ZFS and the throughput of the* create *benchmark on SpanFS against MultiLanes+.*

benchmark. The journaling size of each domain/partition is set to 128 MB. Figure 8 (b) shows the lines generated by SpanFS against MultiLanes+. As the *create* operation need to perform namespace unification which is expensive in MultiLanes+, SpanFS performs much better than MultiLanes+ from 1 to 28 cores. Especially, SpanFS is faster than MultiLanes+ between 72% and 185% from 1 core to 20 cores. Due to the increased contention, the performance improvement shrinks from 24 cores to 32 cores.

4.2.5 Garbage Collection Performance

We evaluate the time the GC takes to scan different numbers of files. We use the *create* benchmark to prepare a set of files in parallel using 32 threads under 32 directories and then remount the file system with the GC thread running at the background. As the GC thread only needs to run when SpanFS finds out that it has just gone through a crash we manually enable the GC thread. Table 4 shows that the GC thread only needs 2.4 seconds to scan and validate all the remote objects when there are 320000 files in the file system, and the time only increases to 20 seconds when there exist 3.2 millions of files. The cost the GC thread incurred is relatively small thanks to the high performance provided by the SSD.

# of files	32000	320000	3200000
# of remote dentries	30032	300030	3000030
Time	1071 ms	2403 ms	20725 ms

Table 4: **Garbage collection performance.** *The time taken to scan the span directories to perform garbage collection. As there exist normal objects, the number of remote dentries represents the actual number of dentries that the GC thread has scanned and validated.*

Then we measure the overhead that the background GC activities contribute to the foreground I/O workloads. Specifically, we prepare 3.2 millions of files as the above does, remount the file system with the GC thread running and then immediately run 32 Varmail instances in parallel. The Varmail workload runs for 60 s.

We measure the aggregative throughput of the 32 Varmail instances. Compared with the normal case without the GC thread running, the total throughput of the Varmail workload has been degraded by 12% (357 MB vs 313 MB), and the GC thread has taken 21950 ms to validate 3024296 remote objects. The number of the validated remote objects is higher than the number in Table 4 as the GC has scanned the remote objects created by the running Varmail workload. Meanwhile, during the above process, the GC thread has found four false invalid remote objects. These false invalid objects are created by

the Varmail workload and are deleted before the GC integrity validation phase. This test also demonstrates that SpanFS can correctly deal with the conflicts between the GC thread and the normal I/O operations.

	Ext4	SpanFS-16
Open without VFS cache	13.9 µs	24.7 µs
Open with VFS cache	3.4 µs	3.5 µs
Rename (1 core)	24 µs	609 µs
Rename (32 cores)	65 µs	2591 µs
unmount	4.323 s	5.272 s
mount	0.021 s	0.086 s

Table 5: **The operation latency.**

4.2.6 Overhead Analysis

We use the average operation latency reported by the above Dbench running in Section 4.2.3 to show the rename overhead. As shown in Table 5, SpanFS with 16 domains is 24X and 39X slower than Ext4 at one core and at 32 cores for the rename operation in Dbench, respectively. We also create a micro-benchmark to evaluate the rename overhead. The rename benchmark renames 10000 files to new empty locations and renames 10000 files to overwrite 10000 existing files. The result shows that SpanFS is 25X and 84X slower than Ext4 for the rename and overwritten rename.

We construct a benchmark *open* to evaluate the overhead of validating the distributed object's integrity during *lookup()* in SpanFS. The benchmark creates 10000 files, remounts the file system to clean the cache and then opens the 10000 files successively. We measure the average latency of each operation. As shown in Table 5, the average latency in Ext4 is around 13.9 µs. In contrast, the average latency is around 24.7 µs in SpanFS with 16 domains. We then open the 10000 files again without remounting the file system. The results show that with the VFS cache SpanFS exhibits almost the same performance with Ext4.

We create a benchmark *mount* to evaluate the performance of the *mount* and *unmount* operation in SpanFS. The *mount* benchmark untars the compressed Linux 3.18.0 kernel source, then unmounts the file system and mounts it again. Table 5 shows the time taken for SpanFS and Ext4 to unmount and mount the file system. As SpanFS builds the domains in sequence, SpanFS with 16 domains performs significantly worse than Ext4 for the mount operation. Nevertheless, the time taken to mount SpanFS only costs 86 ms.

5 Related Work

Scalable I/O stacks. Zheng et al. [43] mainly focus on addressing the scalability issues within the page cache layer, and try to sidestep the kernel file system bottlenecks by creating one I/O service thread for each SSD. However, their approach comes at the cost of communication between application threads and the I/O threads.

Some work that tries to scale in-memory file systems has emerged. ScaleFS [21] uses per core operation logs to achieve scalability of in-memory file system operations that can commute [19]. Hare [25, 24] tries to build a scalable in-memory file system for multi-kernel OS. However, these work does not focus on the scalability issues of the on-disk file systems that need to provide durability and crash consistency.

Wang et al. [40] leverage emerging non-volatile memories (NVMs) to build scalable logging for databases, which uses a global sequence number (GSN) for dependency tracking between update records and transactions across multiple logs. However, due to the need of complex dependency tracking, applying their approach to the file systems needs to copy the updates to the journaling layer, which will introduce copying overhead that has almost been eliminated in the file system journaling [39]. Meanwhile, their work needs the support of emerging NVMs.

Isolated I/O stacks. Some work that tries to build isolated I/O stacks shares some similarities with the domain abstraction in our work in functionality. MultiLanes [27] builds an isolated I/O stack for each OS-level container to eliminate contention. Vanguard [36] and its relative Jericho [32] build isolated I/O stacks called slices and place independent workloads among the slices to eliminate performance interference by assigning the top-level directories under the root directory to the slices in a round-robin manner. IceFS [31] partitions the on-disk resources among containers called cubes to provide isolated I/O stacks mainly for fault tolerant and provides a dedicated running transaction for each container to enable parallel transaction commits. However, these work cannot reduce the contention within each single workload that runs multiple threads/processes as it is hosted inside one single isolated I/O stack.

In contrast with the above work, our work distributes all files and directories among the domains to achieve scalability and proposes a set of techniques to build a global namespace and to provide crash consistency.

Although the domain abstraction in our work shares some similarities with the cube abstraction of IceFS [31], they differ in the following aspects. First, the cubes of IceFS still share the same journaling instance, which can cause contention when multiple cubes allocate log space for new transactions simultaneously. Meanwhile, their approach may still need to serialize parallel checkpoints to make free space due to the single log shared by multiple cubes. In contrast, each domain in SpanFS has its own journaling instance. Second, IceFS does not focus on the lock contention within the block buffer cache layer while SpanFS provides a dedicated buffer cache address space for each domain to avoid such contention. Third,

although IceFS supports dynamic block group allocation, their paper does not describe how to provide crash consistency during allocation. In contrast, our work provides a detailed design of the block group reallocation mechanism as well as how to maintain crash consistency during reallocation.

The online adjusting of each domain's size in the unit of block groups shares some similarities with the Ext2/3 online resizing [20]. However, the Ext2/3 online resizing only focuses on adjusting one file system's size. Our work provides a design on online adjusting of the storage space among multiple domains on demand and maintaining crash consistency during reallocation.

RadixVM [18] implements a scalable virtual memory address space for non-overlapping operations in their research OS. However, applying their approach to the buffer cache address space needs to modify the Linux kernel. In contrast, we leverage the Linux OS block architecture to provide a dedicated buffer cache address space for each domain to avoid the lock contention.

Scalable kernels. Disco [12] and Cerberus [37] run multiple operating systems through virtualization to provide scalability. Cerberus [37] provides a consistent clustered file system view on top of the virtual machines (VMs). However, their approach comes with the cost of inter-VM communication. Moreover, their paper does not explicitly discuss how to maintain consistency of the clustered file system in case of system crashes. Hive [14] and Barrelfish [8] achieve scalability on many-core through the multikernel model. Some work proposes new OS structures to achieve scalability on many-core, such as Corey [10], K42 [28] and Tornado [22]. SpanFS is influenced and inspired by these work but focuses on scaling file systems on fast storage as well as providing crash consistency.

File system consistency check. NoFS [17] stores the backpointers in data blocks, files and directories to verify the file system inconsistencies online, avoiding the journaling overhead. However, as NoFS cannot verify the inconsistencies of allocation structures such as inode bitmap, it needs to scan all the blocks and inodes to build the allocation information at mount time [17]. In contrast, SpanFS only needs to perform GC when it has gone through a crash and only needs to scan the remote dentries under the span directories rather than the whole device in case of a system crash.

Distributed file system. Some distributed file systems, such as Ceph [41] and IndexFS [34], partition the global namespace across computer nodes to provide parallel metadata service. These work relies on the interconnected network in a cluster to maintain a consistent view across machines. In contrast, SpanFS relies on the CPU cache coherence prototype to maintain consistency on data structures within a single many-core machine.

6 Conclusion and Future Work

In this paper, we first make an exhaustive analysis of the scalability bottlenecks of existing file systems, and attribute the scalability issues to their centralized design, especially the contention on shared in-memory data structures and the serialization of internal actions on devices. Then we propose a novel file system SpanFS to achieve scalability on many cores. Experiments show that SpanFS scales much better than Ext4, bringing significant performance improvements.

In our future work, we will implement the online adjusting of each domain's size, explore the adjusting policies and evaluate their performance. In our current prototype, the number of domains is fixed. We will explore the dynamic domain creation strategy.

7 Acknowledgements

We would like to thank our shepherd Chia-Lin Yang and the anonymous reviewers for their valuable suggestions that help improve this paper significantly. Junbin Kang and Benlong Zhang have contributed equally to this work. Junbin Kang and Tianyu Wo are the corresponding authors of this paper. This work is supported in part by China 973 Program (2011CB302602), National Natural Science Foundation of China (91118008, 61322207), China 863 program (2013AA01A213), HGJ Program (2013ZX01039002-001) and Beijing Higher Education Young Elite Teacher Project.

References

[1] SysBench, https://github.com/akopytov/sysbench.

[2] OpenZFS, http://open-zfs.org/wiki/Main_Page.

[3] IOzone Benchmark, http://www.iozone.org/.

[4] Dbench, https://dbench.samba.org/.

[5] Ext4 Disk Layout. https://ext4.wiki.kernel.org /index.php/Ext4_Disk_Layout. Accessed May 2015.

[6] Filebench. http://filebench.sourceforge.net/wiki /index.php/Main_Page.

[7] lockstat. https://www.kernel.org/doc/Documentation/ locking/lockstat.txt.

[8] BAUMANN, A., BARHAM, P., DAGAND, P., HARRIS, T. L., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (2009).

[9] BJØRLING, M., AXBOE, J., NELLANS, D. W., AND BONNET, P. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *SYSTOR* (2013).

[10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, M. F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *OSDI* (2008).

[11] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *OSDI* (2010).

[12] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. DISCO: running commodity operating systems on scalable multiprocessors. In *SOSP* (1997).

[13] CAO, M., BHATTACHARYA, S., AND TS'O, T. Ext4: The next generation of ext2/3 filesystem. In *2007 Linux Storage & Filesystem Workshop, LSF 2007* (2007).

[14] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *SOSP* (1995).

[15] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA* (2011).

[16] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *SOSP* (2013).

[17] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *FAST* (2012).

[18] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys* (2013).

[19] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP* (2013).

[20] DILGER, A. E. Online ext2 and ext3 filesystem resizing. In *Ottawa Linux Symposium 2002* (2002).

[21] EQBAL, R. ScaleFS: A multicore-scalable file system. Master's thesis, Massachusetts Institute of Technology, Aug. 2014.

[22] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI* (1999).

[23] HAGMANN, R. B. Reimplementing the cedar file system using logging and group commit. In *SOSP* (1987).

[24] III, C. G. *Providing a Shared File System in the Hare POSIX Multikernel*. PhD thesis, Massachusetts Institute of Technology, June 2014.

[25] III, C. G., SIRONI, F., KAASHOEK, M. F., AND ZELDOVICH, N. Hare: a file system for non-cache-coherent multicores. In *EuroSys* (2015).

[26] KANG, J., HU, C., WO, T., ZHAI, Y., ZHANG, B., AND HUAI, J. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. An extended verison of [27] submitted to a journal.

[27] KANG, J., ZHANG, B., WO, T., HU, C., AND HUAI, J. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. In *FAST* (2014).

[28] KRIEGER, O., AUSLANDER, M. A., ROSENBURG, B. S., WISNIEWSKI, R. W., XENIDIS, J., SILVA, D. D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M. A., MERGEN, M. F., WATERLAND, A., AND UHLIG, V. K42: building a complete operating system. In *EuroSys* (2006).

[29] LEE, S., MOON, B., AND PARK, C. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD* (2009).

[30] LIU, R., ZHANG, H., AND CHEN, H. Scalable read-mostly synchronization using passive reader-writer locks. In *USENIX ATC* (2014).

[31] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical disentanglement in a container-based file system. In *OSDI* (2014).

[32] MAVRIDIS, S., SFAKIANAKIS, Y., PAPAGIANNIS, A., MARAZAKIS, M., AND BILAS, A. Jericho: Achieving scalability through optimal data placement on multicore systems. In *IEEE MSST* (2014).

[33] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. In *Ottawa Linux Symposium* (2001).

[34] REN, K., ZHENG, Q., PATIL, S., AND GIBSON, G. A. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC* (2014).

[35] RODEH, O., BACIK, J., AND MASON, C. BTRFS: the Linux B-Tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 9:1–9:32.

[36] SFAKIANAKIS, Y., MAVRIDIS, S., PAPAGIANNIS, A., PAPAGEORGIOU, S., FOUNTOULAKIS, M., MARAZAKIS, M., AND BILAS, A. Vanguard: Increasing server efficiency via workload isolation in the storage I/O path. In *Proceedings of the ACM Symposium on Cloud Computing* (2014).

[37] SONG, X., CHEN, H., CHEN, R., WANG, Y., AND ZANG, B. A case for scaling applications to many-core with OS clustering. In *EuroSys* (2011).

[38] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *USENIX ATC* (1996).

[39] TWEEDIE, S. C. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[40] WANG, T., AND JOHNSON, R. Scalable logging through emerging non-volatile memory. *PVLDB 7*, 10 (2014), 865–876.

[41] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI* (2006).

[42] WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage (TOS) 2*, 1 (2006), 74–105.

[43] ZHENG, D., BURNS, R. C., AND SZALAY, A. S. Toward millions of file system IOPS on low-cost, commodity hardware. In *SC* (2013).

Shoal: smart allocation and replication of memory for parallel programs

Stefan Kaestle, Reto Achermann, Timothy Roscoe, Tim Harris*
Systems Group, Dept. of Computer Science, ETH Zurich **Oracle Labs, Cambridge, UK*

Abstract

Modern NUMA multi-core machines exhibit complex latency and throughput characteristics, making it hard to allocate memory optimally for a given program's access patterns. However, sub-optimal allocation can significantly impact performance of parallel programs.

We present an array abstraction that allows data placement to be automatically inferred from program analysis, and implement the abstraction in Shoal, a runtime library for parallel programs on NUMA machines. In Shoal, arrays can be automatically replicated, distributed, or partitioned across NUMA domains based on annotating memory allocation statements to indicate access patterns. We further show how such annotations can be automatically provided by compilers for high-level domain-specific languages (for example, the Green-Marl graph language). Finally, we show how Shoal can exploit additional hardware such as programmable DMA copy engines to further improve parallel program performance.

We demonstrate significant performance benefits from automatically selecting a good array implementation based on memory access patterns and machine characteristics. We present two case-studies: (*i*) Green-Marl, a graph analytics workload using automatically annotated code based on information extracted from the high-level program and (*ii*) a manually-annotated version of the PARSEC Streamcluster benchmark.

1 Introduction

Memory allocation in NUMA multi-core machines is increasingly complex. Good placement of and access to program data is crucial for application performance, and, if not carefully done, can significantly impact scalability [3, 13]. Although there is research (e.g. [7, 3]) in adapting to the concrete characteristics of such machines, many programmers struggle to develop software applying these techniques. We show an example in Section 5.1.

The problem is that it is unclear which NUMA optimization to apply in which situation and, with rapidly evolving and diversifying hardware, programmers must repeatedly make manual changes to their software to keep up with new hardware performance properties.

One solution to achieve better data placement and faster data access is to rely on automatic online monitoring of program performance to decide how to migrate data [13]. However, monitoring may be expensive due to missing hardware support (if pages must be unmapped to trigger a fault when data is accessed) or insufficiently precise (if based on sampling using performance counters). Both approaches are limited to a relatively small number of optimizations (e.g. it is hard to incrementally activate large pages or switch to using DMA hardware for data copies based on monitoring or event counters)

We present Shoal, a system that abstracts memory access and provides a rich programming interface that accepts hints on memory access patterns at the runtime. These hints can either be manually written or automatically derived from high-level descriptions of parallel programs such as domain specific languages. Shoal includes a machine-aware runtime that selects optimal implementations for this memory abstraction dynamically during buffer allocation based on the hints and a concrete combination of machine and workload. If available, Shoal is able to exploit not only NUMA properties but also hardware features such as large pages and DMA copy engines. Our contributions are:

- a memory abstraction based on arrays that decouples data access from the rest of the program,
- an interface for programs to specify memory access patterns when allocating memory,
- a runtime that selects from several highly tuned array implementations based on access patterns and machine characteristics and can exploit machine specific hardware, features
- modifications to Green-Marl [20], a graph analytics language, to show how Shoal can extract access patterns automatically from high-level descriptions.

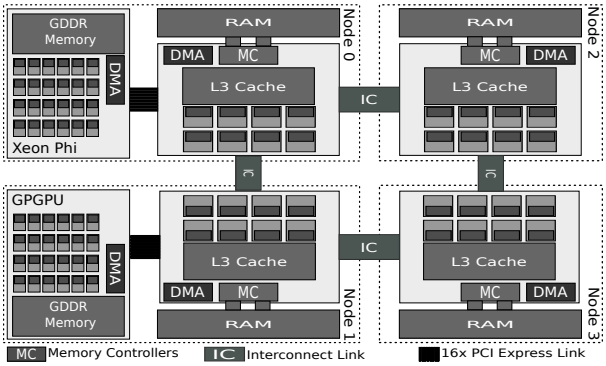


Figure 1: Architecture of a modern multi-core machine.

2 Motivation

Modern multi-core machines have complex memory hierarchies consisting of several memory controllers placed across the machine – for example, Figure 1 shows such a machine with four host main memory controllers (one per processor socket). Memory latency and bandwidth depend on which core accesses which memory location [8, 10]. The interconnect may suffer congestion when access to memory controllers is unbalanced [13].

Future machines will be more complex: they may not provide global cache coherence [21, 25], or even shared global physical addresses [1]. Even today, accelerators like Intel's Xeon Phi, GPGPUs, and FPGAs have higher memory access costs for parts of the physical memory space [1]. Such hardware demands even more care in application data placement.

This poses challenges to programmers when allocating and accessing memory. First, detailed knowledge of hardware characteristics and a good understanding of their implications for algorithm performance is needed for efficient scalable programs. Care must be taken when choosing a memory controller to allocate memory from, and how to subsequently access that memory.

Second, hardware changes quickly meaning that design choices must be constantly re-evaluated to ensure good performance on current hardware. This imposes high engineering and maintenance costs. This is worthwhile in high-performance computing or niche markets, but general purpose machines have too broad a hardware range for this to be practical for many domains. The result is poor performance on most platforms.

These problems can be seen in much code today. Programmers take little care of where memory is allocated and how it is accessed. In cases like the popular Streamcluster benchmark (evaluated in Section 5.1) and applications from the NAS benchmark suite [13], memory is allocated using a low-level malloc call which provides no guarantees about where memory is allocated or other details such as the page size to use.

For example, Linux currently employs a first-touch memory allocation strategy. Memory is not allocated directly when calling `malloc`, but mapped only when the corresponding memory is first accessed by a thread. This resulting page fault will cause Linux to back the faulting page from the NUMA node of the faulting core.

A surprising consequence of this choice is that on Linux the implementation of the initialization phase of a program is often critical to its memory performance, even through programmers rarely consider initialization as a candidate for heavy optimization, since it almost never dominates the total *execution time* of the program. To see why, consider that `memset` is the most widely used approach for initializing the elements of an array. Most programmers will spend little time evaluating alternatives, since the time spent in the initialization phase is usually negligible. An example is as follows:

```
// ---- Initialization (sequential) -----------
void *ptr = malloc(ARRSIZE);
memset(ptr, 0, ARRSIZE);
// ---- Work (parallel, highly optimized) -----
execute_work_in_parallel();
```

The scalability of a program written this way can be limited. `memset` executes on a single core and so all memory is allocated on the NUMA node of that core. For memory-bound parallel programs, one memory controller will be saturated quickly while others remain idle since all threads (up to 64 on the machines we evaluate) request memory from the same controller. Furthermore, the interconnect close to this memory controller will be more susceptible to congestion.

There are two problems here: (*i*) memory is not allocated (or mapped) *when* the interface suggests (memory is not allocated inside `malloc` itself but later in the execution) and (*ii*) the choice of where to allocate memory is made in a subsystem (the OS kernel) that has no knowledge of the intended access patterns of this memory.

This can be addressed by tuning algorithms to specific operating systems. For example, we could initialize memory using a parallel for loop:

```
// ---- Initialization (parallel) -------------
void *ptr = malloc(ARRSIZE);
#pragma omp parallel for
for (int i=0; i<ARRSIZE; i++)
  init(i);
// ---- Work (parallel, highly optimized) -----
execute_work_in_parallel();
```

This will be faster and retain scalability in current versions of Linux. The first-touch strategy will equally spread out memory across all memory controllers, which balances the load on them and reduces contention on individual interconnect links.

One drawback of this strategy is the loss of portability and scalability when the OS kernel's internal memory

allocation policies change. Furthermore, it also requires correct setup of OpenMP's CPU affinity to ensure that all cores participate in this parallel initialization phase in order to spread memory equally on all memory controllers. Finally, we might do better: allocate memory close to the cores that access it the most.

Beyond simple placement of data, ideas and techniques from traditional distributed systems like replication and partitioning can help to improve memory management [31]. Replication localizes data access by storing several copies of the same data, distributing load and reducing communication costs. Replication carries the cost of maintaining consistency when updating data, as well as increasing the program's memory footprint. Partitioning chunks data and places these blocks onto different nodes. This balances load, and, if work is scheduled close to data it is accessing, also localizes array accesses. The key challenge in applying these techniques is that the choice of a good distribution strategy depends critically on concrete combinations of machine and workload.

Our work starts with the observation that memory access patterns of applications are often encoded in high-level languages or known by programmers. We show how this information can be used to tune memory placement and access without programmers having to understand the characteristics of the machine at hand.

Automatic annotations from high-level DSLs. A trend we exploit is the emergence of high-level domain specific languages (DSLs) [20, 37, 41]. These languages are known for the ease of programming since their semantics closely match a specific application domain. DSLs typically compile to a low-level language (such as C), possibly with several backends depending on the target machine to execute the program. DSLs can provide us with memory access patterns directly from the input program with relatively simple modifications to high-level compilers. Listing 1 shows an example program for the Green-Marl graph analytics DSL.

Memory access patterns from two of Green-Marl's high-level constructs in the PageRank example can be determined as follows: (*i*) `Foreach (T: G.Nodes)` means the nodes-array will be accessed sequentially, read-only, and with an index, and (*ii*) `Sum(w: t.InNbrs)` implies read-only, indexed accesses on in-neighbors array.

```
Procedure pagerank(/* arguments */) {
    // .. initialization here
    Do {
        diff = 0.0; cnt++;
        Foreach (t: G.Nodes) {
            Double val = (1-d) / N + d*
                Sum(w: t.InNbrs) {
                   w.pg_rank / w.OutDegree()} ;
            diff += | val - t.pg_rank |;
            t.pg_rank <= val @ t;
        }
    } While ((diff > e) && (cnt < max));
}
```

Listing 1: Excerpt from Green-Marl's PageRank

We argue that memory access patterns encoded in DSLs present a significant performance opportunity, and should be passed to the runtime to enable automatic tuning of memory allocation and access. Since low-level code is generated by the DSL compiler, it is also relatively easy to change the programming abstractions used by the generated code for accessing memory. Only the compiler (rather than the input program) must be changed in such a case.

Manual annotations. Even without a DSL, programmers often know data access patterns when writing a program. They understand the semantics of their programs and, hence, how memory is accessed, but have no way of passing this knowledge to the runtime to guide data placement and access. Existing interfaces intended to enable this coarse-grained and inflexible. One example is libnuma's [34] NUMA-aware memory allocation, which allows a client to specify which node memory should be allocated from, but does not allow combining this with other allocation options (such as large pages), and requires a programmer to manually integrate this with parallel task scheduling.

In Shoal, we automatically tune data placement and access based on memory access patterns and hints provided by a high-level compiler or by programmers. We introduce (*i*) a new interface for memory allocation, including machine-aware `malloc` call that accepts hints to guide placement and (*ii*) an abstraction for data access based on arrays. For these arrays, we provide several implementations including data distribution, replication and partitioning. All implementations can be interchanged transparently without the need to change programs. Our abstraction also admits implementations that are tuned to hardware features (such as DMA engines) or accelerators (Xeon Phi). The Shoal library automatically selects array implementations based on array access patterns and machine specifications. We currently support adaptions based on the NUMA hierarchy, DMA engines, and large MMU pages.

The result is that Shoal allows programmers to write programs that achieve good performance without having (*i*) to understand machine characteristics and (*ii*) needing to constantly rewrite applications in order to keep up with hardware changes. We demonstrate Shoal using the Green-Marl graph DSL, and the Streamcluster low-level C program from the PARSEC benchmark suite.

3 Shoal's array abstraction

Shoal's memory abstraction is based on arrays. We found this sufficient for the workloads we have been looking at in the context of this research, but we expect to add more data types in the future. We provide several array implementations, but all of them implement the same interface. This allows Shoal to select an implementation transparently to the programmer. The optimal choice depends on machine characteristics and memory access patterns.

```
// allocate an array
template<class T>
shl_array<T>* shl__malloc_array(size_t size,
                        bool ro, bool indexed,
                        bool used);
// get element at position i
T get(size_t i);
// set element at position i to v
void set(size_t i, T v);
// number of elements
size_t get_size(void);
// copy from another Shoal array
int copy_from_array(shl_array<T> *src);
// initialize every element with value
int init_from_value(T value);
// copy from a non-Shoal array
void copy_from(T* src);
// calculate the CRC checksum
unsigned long get_crc(void);
// initialize the current thread
void shl__thread_init(void);
// synchronize replicas
void shl__repl_sync(void* src, void **dest,
                size_t num_dest, size_t size);
```

Listing 2: Interface of Shoal

3.1 Interface and programming model

Listing 2 illustrates Shoal's programming interface, which decouples computation and memory access allowing transparent selection of different array implementations. Besides the usual `set()` and `get()` operators, we provide a collection of high-level functions to initialize memory and copy between Shoal arrays.

Thread initialization. In OpenMP, Shoal uses builtin functions to determine the thread ID and the corresponding replica to be used. Per-thread array pointers can otherwise be setup by manually calling `shl__thread_init()` on each thread.

Array allocation. `shl__malloc_array` allocates Shoal arrays and selects the best implementation for the machine it is running on based on memory access pattern hints given as arguments. Shoal always maps all pages of an array to guarantee memory allocation and avoid non-determinism.

Data operations. Reads and writes to arrays are performed with `get()` and `set()`, but we also provide optimized high-level array operations for initializing and copying arrays. These provide relaxed consistency guarantees: the order in which elements are initialized or copied is not specified, allowing these operations to be parallelized and offloaded to DMA engines in an asynchronous fashion. Writes to replicas can be realized by writing to the master copy and propagating the changes to all replicas using `shl__repl_sync()`. This allows to re-initialize replicated arrays, for example to reuse otherwise read-only buffers in streaming applications.

3.2 Array types

We currently provide four array implementations.

Single-node allocation. Allocates the entire array on the local node. While limited in scalability, performance is independent of the OS since memory is guaranteed to be mapped in the allocation phase. Single-node arrays are rarely used for parallel programs.

Distribution. Distributed arrays allocate data equally across NUMA nodes. The precise distribution is not specified and depends on the implementation. This reduces pressure on memory controllers, but can lead to high latency or congestion if many accesses are remote. The performance of distributed arrays can be non-deterministic, as data is scattered semi-randomly and might vary between program executions.

Replication. Several copies of the array are allocated. We currently always place one replica on each memory controller. All data is then accessed locally. In addition to distributing load across the system, this reduces pressure on the interconnect at the cost of increased memory footprint.

Partitioning. Partitioning is a form of distribution where data is spread out in the machine such that work units can be executed local to where their data is allocated. If done carefully, array accesses are local as with replication, but without the increased memory footprint. This implies a scheduling challenge, since the working set for each thread must be known and the jobs scheduled accordingly.

3.3 Selection of arrays

In selecting an array implementation, we try to: (*i*) maximize local access to minimize interconnect traffic, (*ii*) load-balance memory on all available controllers to avoid points of contention, and (*iii*) transparently use hardware features when available (e.g. DMA and large pages).

We show our policy for selecting array implementations in Figure 2: 1. we use partitioning if the array is only accessed via an index, 2. we enable replication if the array is read-only and fits into every NUMA node of the host machine, and 3. we otherwise use a uniform distribution among all available memory controllers.

We only replicate read-only arrays, as we found that the cost for maintaining consistency dominates the performance benefits in current NUMA machines (Section 5.4) – however, we plan to revisit this for more complex NUMA hierarchies. In case of limited RAM, where the increase in working set size with replication is not tolerable, we can selectively activate replication based on the cost function of memory accesses extracted from the high-level program.

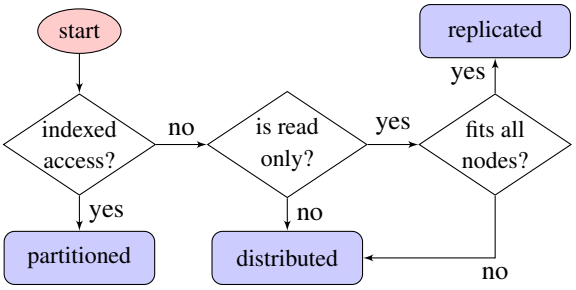


Figure 2: Array Selection

Large pages. If available, we use *large pages*. This is not always optimal, and the impact of using large pages is hard to anticipate, but on average enabling large pages improves performance (in the future, we plan to enable large pages per-array). Most current multi-core machines have independent TLBs for different page sizes, suggesting it might be useful to retain some arrays on normal pages. Also, the TLBs coverage for randomly accessed large arrays is still not sufficient to prevent TLB misses. One approach would use large pages for mostly-sequential array access, and 4k pages for randomly accessed data. We also plan to use *huge* pages (typically 1GB), which may keep most of the working set covered by the TLB even for big workloads.

Overall, we have found this policy to be simple, but effective.

4 Implementation

The Shoal runtime library is structured in two parts: (*i*) a high-level array representation as defined in Section 3 based on C++ templates and (*ii*) a low-level, OS-specific backend. We now describe the work-flow invoked when Shoal is used together with a high-level DSL, and then describe our low-level backends.

Figure 3 shows how Shoal is used with high-level, parallel languages.

High-level program. The input program is written in a high-level parallel language, which in this paper is Green-Marl, a DSL for graph analysis. Many other high-level languages such as SQL [18] and OptiML [37] provide similar resource usage information to the kind we extract from Green-Marl; we show an example of a Green-Marl expression of PageRank in Listing 1.

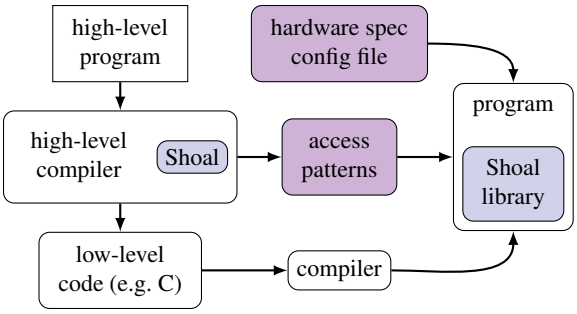


Figure 3: Shoal system overview

High-level compiler. High-level DSLs often encode access patterns in an intuitive way: for instance, the Green-Marl DSL features language constructs to access all nodes in a graph (see `Foreach (t: G.Nodes)` in Listing 1). From the language specification, we know that this represents a read-only and sequential data access to the nodes array. The high-level compiler translates the input into low-level code while such access information is lost. Our modifications to the Green-Marl compiler extract this knowledge about array access patterns from the input source code and makes it accessible to the generated code which uses Shoal's array abstraction.

Low-level code with array abstractions. The generated code – here, C++ – uses Shoal's abstraction to allocate and access memory. At compile time the concrete choice of array implementation is *not* made; this happens later at runtime based on hardware specifications.

Access patterns. In high-level languages, memory access are usually implicit and translated into simple load and store instructions. In Shoal, however, we use the compiler to generate important information about load/-store patterns which is then used by the runtime library.

Firstly, we capture the *read/write-ratio*. The number of reads and writes by a program is workload specific, but Shoal can still in many cases extract a formula estimating the number of reads and writes for a given input. For example, the number of reads in Green-Marl's PageRank rank array is: $kE * kN$, where E is the number of edges and N is the number of nodes. Currently, Shoal derives these formulas but only uses them to determine if the array is read-only; we expect more sophisticated uses of this information in the future. Secondly, we infer if all accesses to an array are based solely on the loop variable of a parallel loop. `tmp` indicates the array is used for temporary values, `ro` that it is read-only, etc.

An example of the automatically extracted information for Green-Marl's PageRank can be seen on Table 1.

Hardware specification. The Shoal runtime takes the hardware configuration of the system into account when selecting array implementations. Currently, we consider the following hardware features: (*i*) *NUMA topology*: Shoal has a NUMA-aware array allocation function that

array	N	E	ro	std	tmp	indexed
`begin`	y		y	y		
`r_begin`	y		y	y		
`r_node_idx`		y	y	y		
`pg_rank`	y					
`pg_rank_nxt`	y				y	y

Table 1: Shoal's extracted array properties for PageRank

attempts to distribute load on all memory controllers and localize access to reduce pressure on interconnects. (*ii*) *RAM size*: available memory can limit which arrays can be replicated. (*iii*) *Page size*: Modern systems offer various page sizes and often provide a dedicated TLB for each size. The use of large and huge pages is useful in reducing TLB misses [17] in large working sets. (*iv*) *DMA engines*: Some CPUs have integrated DMA engines [22]. We make use of these for copy and initialization.

Shoal program. The Shoal library takes care of selecting array implementations based on the extracted access patterns, and hardware specification of the machine. The executable generated by the compiler is a program binary which links against the Shoal library.

OS-specific backends. To improve portability, we separate high-level array implementations from low-level, OS-dependent functions which mediate access to the memory allocation facilities or DMA devices. Currently, we run on the Linux and Barrelfish [6] OSes to demonstrate portability.

Topology information. Shoal needs to obtain information about the system architecture including number of NUMA nodes, their sizes and corresponding CPU affinities. On Linux, this information can be obtained using *libnuma* [34]. In Barrelfish, hardware information is stored in the system knowledge base [33].

Scheduling. For replication and partitioning, Shoal must map threads to cores. On Linux we pin threads by setting the affinities, whereas on Barrelfish we directly create threads on specific cores. Given a concrete data distribution, scheduling can be optimized to execute work units close to where data is accessed. To date, Shoal is not fully integrated with the OpenMP runtime and we use a static OpenMP schedule for partitioning to ensure that work units are executed close to the partitions they are working on. This works well for balanced workloads, but can lead to significant slowdown compared to dynamic schedules if the cost of executing work units is non-uniform. In the future, we plan to design and integrate our own OpenMP runtime to provide us fine-grained control of scheduling without losing performance for unbalanced workloads. An alternative approach would schedule work units on partitions using OpenMP 4.0's `team`-statement.

Memory allocation. We want to provide strong guarantees on where memory is allocated, but allocation policies are not consistent across OSes – indeed, they even change between different version of the same OS. Linux, for instance, implements a first touch allocation policy, which causes confusion about where and when memory will actually be allocated. Libraries such as libnuma provide an interface which gives more control, but this lacks support for large and huge pages. Barrelfish [7] gives the user the ability to manage its own address space via self-paging [19]: an application requests memory explicitly from a specific NUMA node and maps it as it wishes.

These systems provide different trade-offs between complexity, portability, and maintainability of application code and efficient use of the memory system: an explicit, flexible interface imposes an additional burden on the client. We believe that programmers should not have to deal with this complexity and want to avoid manual tuning to adapt programs to new machines.

5 Evaluation

Our goal in this section: we show that programs scale and perform significantly better with Shoal than with a regular memory runtime. We also show a comparison of our array implementations and analyze Shoal's initialization cost, and finally we investigate the benefits of using a DMA engine for array copy.

Table 2 shows the machines used for our evaluation. Our results on both, 8x8 AMD Opteron and 4x8x2 Intel Xeon, are similar. For brevity, we focus on results from our 8x8 AMD Opteron unless stated otherwise. We use two workloads: Green-Marl and PARSEC Streamcluster.

Green-Marl. The Green-Marl compiler comes with a variety of programs of which we have selected three graph algorithms to demonstrate the performance characteristics of Shoal: (*i*) *PageRank* [30] iteratively calculates the importance of each node in the graph as a sum of the rank of all incoming neighbors divided by the number of outgoing edges they have, (*ii*) *hop-distance* calculates the distance of every node from the root using Bellman-Ford, and (*iii*) *triangle-counting* that counts the number of triangles in the input graph. This is implemented as a triple loop: for all nodes in the graph, it looks at all combinations of nodes reachable from it and checks if there is an edge connecting them.

For our evaluation we used two graphs: (*i*) the Twitter graph [23] having 41M nodes and 1468M edges. The total working set size is 2.459 GB with Green-Marl configured to 64 bit node and edge types (excluding unused arrays). We were not able to run triangle-counting on the Twitter graph on our system, hence we were falling back on the LiveJournal graph [5] for that workload. LiveJournal has 4M nodes and 69M edges with a total working set of 392 MB in Green-Marl.

machine	8x8 AMD Opteron	4x8x2 Intel Xeon	2x10 Intel Xeon
CPU	AMD Opteron 6378	Intel Xeon E5-4640	Intel Xeon E5-2670 v2
micro architecture	Piledriver	Sandy Bridge	Ivy Bridge
#nodes/#sockets	8/4 @ 2.4 GHz	4/4 @ 2.4 GHz	2/2 @ 2.5 GHz
L1 data size	16K /thread	32K /core	32K /core
L2/L3 size	2048K / 6144K	256K / 20480K	256K / 25600K
memory	512 GB (64 GB per node)	512 GB (128 GB per node)	256 GB (128 GB per node)

Table 2: Machines used for evaluation (L2 shared by core, L3 shared by socket)

PARSEC – Streamcluster. Streamcluster [9] solves the online clustering problem. Input data is given as an array of multi-dimensional points. We manually modified it to use Shoal for memory allocation and accesses.

5.1 Scalability

In highly parallel workloads, scalability is one of the key concerns. In this section we show the benefits of using Shoal over unmodified versions of the workloads and that allocating memory based on access patterns, if available, is favorable over online methods.

Green-Marl. We evaluated scalability of three Green-Marl workloads on an 8x8 AMD Opteron comparing Shoal (—■—) against the original Green-Marl implementation (—●—) and Carrefour [13] (—⋆—). Figure 4 shows that Shoal clearly outruns the original implementation by almost 2x and also performs better than the online method as a result of an optimized memory placement. Furthermore, our results show that an online method can harm the performance in case pages are getting migrated back and forth (hop-distance). Overall, except for triangle-counting, all implementations scale well. Note that we do not include the graph loading time and Shoal initialization.

We also executed the same measurements on Barrelfish (—♦—) to show Shoal's portability. Our intention is not to show that either operating system is faster than the other, but rather their comparability. On Barrelfish, only static OpenMP schedules are supported due to implementation limitations. This negatively impacts the performance for triangle-counting. However, Shoal still performs better than the original implementation, which uses dynamic OpenMP schedules.

PARSEC – Streamcluster. In contrast to Green-Marl, Streamcluster is implemented in C and hence there is no automatic method of extracting access patterns. We modified Streamcluster to use Shoal's array abstraction to demonstrate that using Shoal directly by programmers can improve scalability with little efforts for manual annotation. To make Streamcluster work with Shoal, we had to (*i*) abstract access to arrays using Shoal's get and set methods, (*ii*) initialize each thread using `shl__thread_init()` and change the array allocation to use `shl__malloc_array()` instead of `malloc()`.

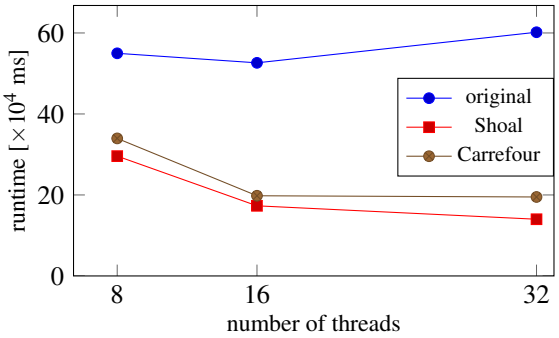


Figure 5: Scalability of PARSEC streamcluster on 8x8 AMD Opteron

Since Streamcluster is a streaming application, arrays for input coordinates are reused for each chunk of new streaming data, but are otherwise read-only. We use (*iii*) `shl__repl_sync()` to synchronize the master copy of the array to it's replicas once after a new chunk has been read. We compare the original Streamcluster implementation with Carrefour and Shoal (Figure 5). Our results confirm the bad scalability of Streamcluster due to the use of `memset()` after allocating arrays [13, 17], which causes all memory to be allocated on a single NUMA node. This leads to congestion of the interconnect and memory controllers of that node. Shoal achieves an 4x improvement over the original implementation. Shoal's annotated access allocation function outperforms Carrefour's online method. We want to emphasize here, that we replaced only one of the used arrays with a Shoal array (large pages and replication) and did not apply further optimizations.

5.2 Comparison of array implementations

We conducted a detailed analysis of Shoal's different array implementations using all physical cores of our machines. In this section we show, that Shoal achieves better performance than the original Green-Marl implementation regardless of which array configuration we use. Figure 6 shows our results normalized to the original Green-Marl implementation and Figure 8 shows the breakdown into initialization and computation times. The measurements were executed on the 8x8 AMD Opteron using all 32 physical cores. Following, we give

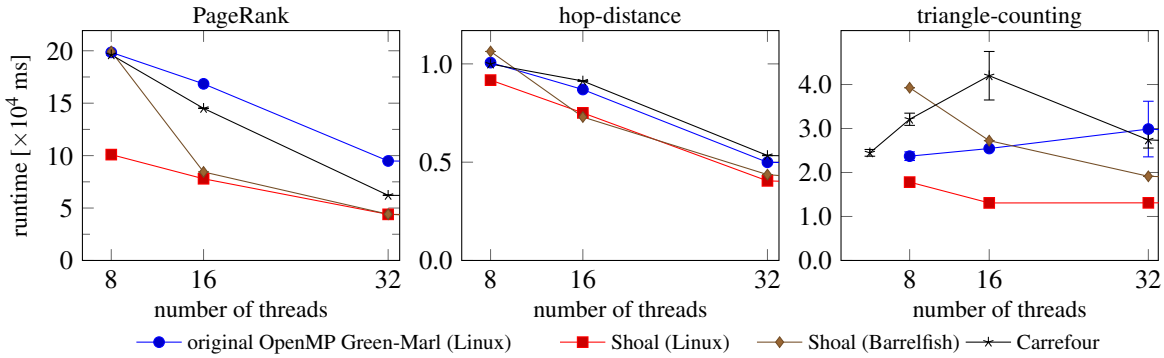


Figure 4: Scalability on 8x8 AMD Opteron. Workload: Twitter (LiveJournal for triangle-counting). For Shoal, we chose the best configuration each. We omit results for 64 threads: using SMT threads has similar performance to using only the 32 physical cores.

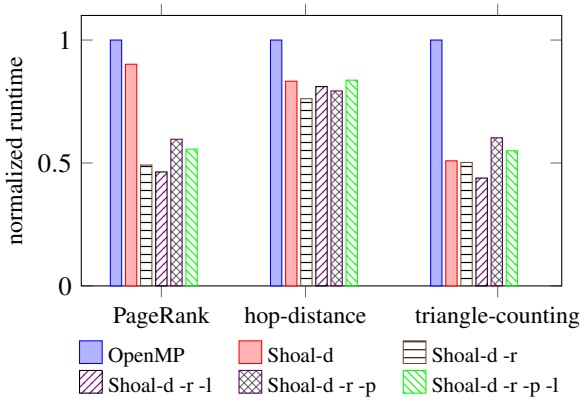


Figure 6: Comparison of various combinations of array implementations on 8x8 AMD Opteron: distribution (-d), replication (-r), partitioning (-p), large page (-l) (runtime normalized to stock-Green-Marl)

explanations for each configuration and relate them to our performance counter observations (Figure 7).

Distribution ( ). The original Green-Marl implementation already initializes memory for storing the graphs with a OpenMP loop to distribute memory in the machine. However, this is not done for dynamically allocated arrays (e.g. the `rank_next` in PageRank). With Shoal, *all* arrays are ensured to be distributed among the nodes, resulting in a more even distribution of memory and a better performance across all workloads. Initialization of distributed arrays relies on an OpenMP loop to allocate memory evenly across all NUMA nodes. Initializing memory is hence executed in parallel, which results in small initialization cost compared to other array types. We show this in Figure 8.

Our claims are supported by measurements of each memory controller's read- and write throughput. (Figure 7): compared to the original implementation (*i*) where all reads and writes are executed on socket 0, enabling distribution (*ii*) results in an evenly distributed load on all memory controllers. However, in both cases, the memory controllers are not saturated. Memory throughput suffers from the lower bandwidth of the interconnect links, i.e. 9.6GB/s for QPI. With randomly distribution of memory, only 1/4 of all memory accesses are expected to be local.

Distribution + replication ( ). In contrast to distribution, replication is applied only to read-only data. In our workloads, the graph itself is not altered by the program and hence replicated among the nodes. This results in a increased fraction of locally served memory accesses and lower interconnect traffic: memory accesses are evenly distributed among all memory controllers as shown in (*iv*) of Figure 7. Note, enabling replication without distribution allocates non-read-only arrays into single-node arrays resulting in an unbalanced memory access for that part of the working set, see (*iii*) of Figure 7. Initialization cost for replicated arrays are higher than distributed arrays because more memory needs to be allocated. We force correct allocation by touching each replica on its designated node. Finally, copying the master array to the other replicas causes some additional overhead when initializing such an array (Figure 8).

Partitioning (and). Replication of data increases the memory footprint of the application. Partitioning tries to preserve the locality of replication without increasing the memory footprint. Our current implementation requires a static OpenMP schedule for partitioning to ensure scheduling of work units to the right partitions. However, static schedules potentially lead to imbalance of work among the execution units as workloads may be skewed (e.g. in triangle-counting). Eventhough the same amount of memory has to be allocated as with distributed arrays, its initialization is more complex: using Linux' first touch policy, Shoal ensures memory is touched on the correct node by migrating a thread to where memory should be allocated and touching each page from there. This results in similar initialization time as with replication, but slightly less time to copy the data.

Large Pages (and). Modern CPUs support various page sizes and have a distinct TLB for each page size. A miss in the TLB enforces the CPU to do a full page table walk which drastically increases the access time. Shoal supports large pages for its arrays. Enabling large pages for PageRank and triangle-counting results in a slightly better performance, while hop-distance runtime increases slightly. Gaud *et al.* [17] concluded similar findings in their experiments with large pages. Enabling large pages reduces the total number of pages used and therefore the number of required first touches in the allocation process. This results in a decrease of the allocation time (Figure 8).

We conclude that despite the additional overhead of allocation and initialization, the total runtime with Shoal is still reduced. However, we want to emphasize here, that we do not consider initialization time as a main target of optimization as typically time spent for computation dominates the program execution. Nevertheless, allocation could be improved by (*i*) maintaining a cache of pre-allocated pages on each node, (*ii*) applying a smarter page mapping strategy or (*iii*) by initializing Shoal while input data (e.g. the graph) is loaded.

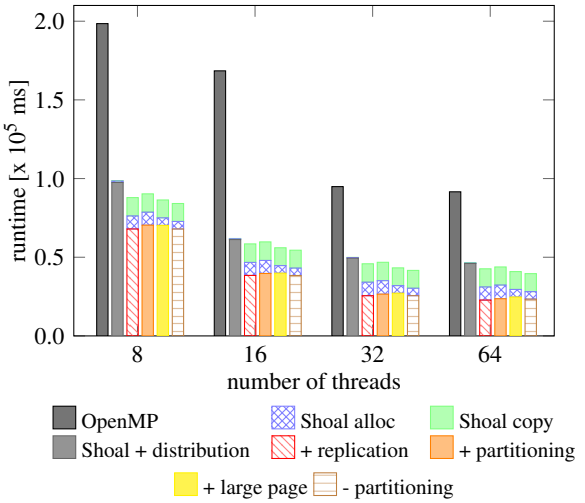


Figure 8: Shoal initialization and runtime on 8x8 AMD Opteron for various array configurations using PageRank with Twitter workload

5.3 Use of DMA engines

Modern CPUs have integrated DMA engines, which provide a rich set of memory operations. For instance, recent Intel server CPUs provide integrated CrystalBeach 3 DMA engines [22]. We evaluate the use of DMA engines for initialization and copy operations on a 2x10 Intel Xeon (our 8x8 AMD Opteron and 4x8x2 Intel Xeon do not have DMA engines). We run these experiments on Barrelfish, as user-level support for DMA engines is already integrated and requires no additional setup.

We now compare the raw copy performance of DMA controllers to CPU `memcpy()` and further evaluate how DMA engines can be used in Shoal. Asynchronous memory operations offered by DMA controllers can free up the CPU of the burden of copying data around and provide cycles to do actual work. Shoal offers an interface to start an asynchronous memory copy and to check for completion of the operation.

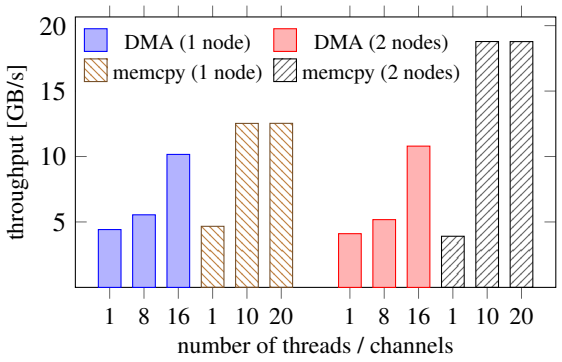


Figure 9: Comparison of parallel OpenMP copy and DMA copy on 2x10 Intel Xeon for large buffers ($\gg$ cache size)

Comparing raw memory throughput. Our results of a raw throughput evaluation (Figure 9) show, that the use of DMA engines does not necessarily improve the sequential performance, especially for blocking copy operations as used by PageRank. However, to outperform the DMA controller, all threads of the CPU have to be used for memory copying and hence no other computational task can be executed in the meantime.

We now show the DMA engines are useful if only a few threads are available for synchronously copying arrays or asynchronous copies. For example, we are planing to evaluate the use of DMA engines to propagate writes to replicas in the background.

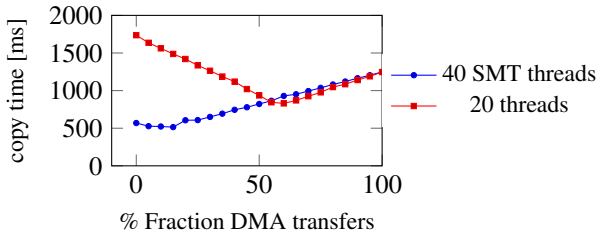


Figure 10: Initialization cost for copying data into Shoal arrays using the Twitter working set with replication on Barrelfish

DMA engines for initialization. We benchmark the initialization phase of PageRank where data is copied from the graph's memory into the Shoal arrays. We copy a certain ratio of the array using DMA engines asynchronously while using parallel OpenMP loops to copy the remaining elements. Figure 10 shows how varying

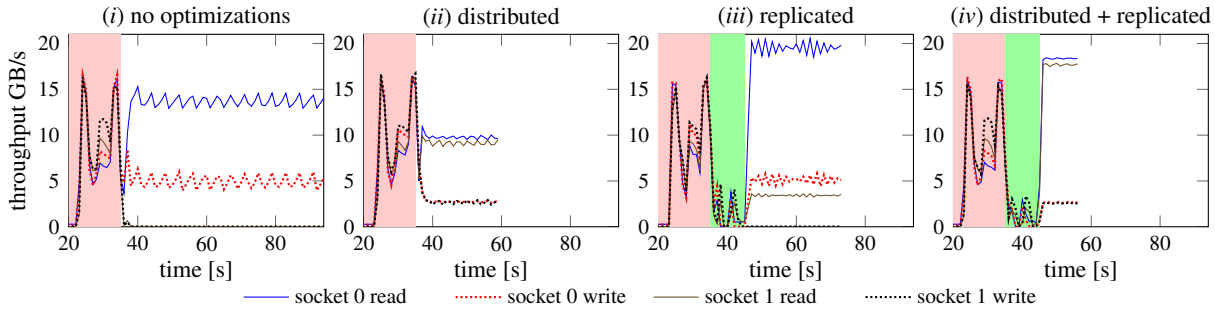


Figure 7: Memory throughput for sockets 0 and 1 on 4x8x2 Intel Xeon. In the first 35 seconds, the graph is loaded from disk. For replication, we show the replica initialization cost in green. Note: Sockets 2 and 3 are equal to socket 1 and left out for readability.

the ratio of how much of the array is copied using DMA engines vs. parallel OpenMP loops affects performance. For the parallel copy, we show the result for using all 20 physical threads and 40 SMT-threads respectively. First, we see a big difference if we enable SMT (—●—): as expected, memory access latency is hidden and the use of a DMA engine improves performance only slightly (about 10%). This is presumably because these 40 hyper-threads are sufficient to saturate all memory controllers on that machine. With SMT disabled (—■—), the memory latency cannot be hidden. Our results show clearly, that using DMA engines and CPU copy simultaneously reduces the time for copying arrays by 2x.

DMA engines for array copy. In our PageRank workload, the ranks are copied between two arrays in every iteration. With our implementation, we can use DMA engines to improve the copy time in that case too. However, our measurements show, that depending on the array configuration only 1-5% of the entire runtime is spent copying and hence optimize that part does not have a notable effect on PageRank's total runtime, hop-distance behaves similarly. However, workloads allowing asynchronously copy of data would be a more obvious candidate for optimizations based on DMA engines.

To sum up, the effect of using DMA controllers for memory operations highly depends on whether the program has to share the resources with other workloads or not. If all resources are available, DMA engines provide about 10% improvement. On the otherhand if resources are shared with other users, DMA engines provide up to 2x improvement in our case.

5.4 Writeable replication

Finally, we look at the issue of write-shared arrays.

Efficiently maintaining consistency of replicated data is difficult; updates must be propagated to all replicas. This can be achieved by issuing writes to all replicas or by applying techniques such as double-buffering and asynchronous copies. Both relax consistency guarantees, but are strong enough for use with OpenMP loops, where concurrent writes and reads in the same loop iteration would cause non-determinism.

In this section, we show that replicating non-read-only data does not deliver much benefit on current NUMA machines for already otherwise optimized workloads: the additional cost of house-keeping (e.g. maintaining write-sets) and propagating updates to all replicas outweighs the potential performance gain of replication.

We compare writeable replicas with single-node allocation and distribution (Table 3). Our results show that the cost of maintaining consistency grows with the number of replicas. Furthermore, replication not necessarily achieves better performance compared to distributed arrays as the load on the interconnect in the latter case is already relatively low.

We believe that writeable replication will be useful (and needed) in heterogeneous systems, where memory non-uniformity is more drastic (e.g. more NUMA nodes, slower links). In that case, replication of data in local memory is crucial for performance even in the presence of updates. Writeable replication could also have an application for more complex workloads (e.g. a smaller fraction of read-only data), where the simple mechanisms we presented in this paper cannot be applied.

dist configuration	cost	stderr	notes
single-node	214.0	11.0	
distributed	203.0	0.9	
wr-rep, 2 reps	248.8	7.0	nodes: 0,n-1
wr-rep, 4 reps	333.6	5.9	nodes: 0,n-1
wr-rep w/o copy op	202.9	7.2	

Table 3: Writeable replicas on 8x8 AMD Opteron. Workload: hop-distance with -d -r -h configuration

6 Related work

Our work was originally inspired by recent research in domain specific languages. Such languages are based on the observation that it is hard to write efficient code for a wide-range of different systems, as algorithms need to be tuned to a concrete machine in order to achieve good performance. DLSs express algorithms in a rich

and intuitive way. The Green-Marl [20] graph analytics DSL, OptiML [37], a machine learning DSL , as well as SPL [41], a signal processing language, all provide a rich set of powerful high-level operators. They use a compiler that generates code that is highly tuned to the target machine and makes heavy use of data parallelism. While all of these languages encode memory access patterns in their high-level languages, none uses them to adapt memory allocation at runtime.

Modern machines are becoming inherently complex: Baumann *et al.* argued that computers are already a distributed system in their own right [8]. They proposed a multikernel approach [7] which avoids sharing of state among OS nodes by replication and applying techniques from distributed systems. Similarly, Wentzlaff *et al.* [40] apply partitioning to OS services. Techniques from distributed systems are beneficial not only on an OS level, but also for applications. Multimed [31] replicates database instances within a single multi-core machine. Salomie *et al.* showed that congestion of memory controllers and interconnects impact the overall performance. Carrefour [13] attempts to reduce the contention on interconnect and memory controllers by online monitoring of memory accesses and auto-tuning NUMA-aware memory allocation. This approach can be applied to any application without modifications to the program code, but is less fine-grained. While Shoal derives a program's semantics from annotations or DSL compiler analysis, the former these approaches need to guess programmers' intentions in retrospect.

Systems such as SGI's Origin 2000 [35] use page-level migration and replication of data. Hardware monitors detect the access patterns to pages (e.g., which processors tend to access the page, and whether these are reads or writes). Based on the gathered data, pages are replicated or migrated towards a frequently accessing CPU.

With highly parallel workloads, efficient synchronization is crucial for application performance [14]. Lock cohorting [15] implements NUMA-aware locks by taking cache hierarchy and NUMA-topology into account. Shoal's treatment of memory is analogous to these systems' treatment of locks.

Access to large and huge MMU pages is provided by services and libraries like `libhugetlbfs` [4]. The latter, however, requires static setup of a large page pool, among other issues.

Cache coherence protocols like MOESI [2] allow cache-lines to be in a shared state which is a form of hardware-level replication. This is only effective with workloads having good locality and small working set, which is not the case for our graph workloads. Research systems, such as Stanford FLASH, have provided software control over this form of replication [36].

The Solaris operating system provides a `madvise` [28] operation to let an application give hints on future accesses to a memory region which results in a distributed or local allocation to the calling thread. Shoal extends this approach with a wider range of possible usage patterns, and infers appropriate settings to use.

Li [24] attempts to find the best algorithm for a specific task depending on machine characteristics and workload based on empirical search. In contrast, we decide a priori based on additional information extracted from high-level languages or given by manual annotations. Franchetti *et al.* [16] automatically tune FFT programs to multi-core machines. They argue that programming such machines is increasingly complicated, which increases the burden for programmers and makes a case for automatic tuning. Atune-IL [32] auto-tunes applications, including the number of threads etc. It explores all possible parameters, but tries to reduce the search space.

However, tuning data placement and parallelism individually is not optimal, because data and threads may not end up on the same node. Hence, affinity of threads and data need to be enforced in order to improve the performance of OpenMP programs [38].

Finally, PGAS languages such as UPC [39], co-array Fortran [27], X10 [12], Chapel [11], and Fortress [29] provide an abstraction of shared arrays which can be implemented across a distributed system. Code iterating over an array can execute on the node holding the portion of the array being accessed.

In high-performance computing, array abstractions [26] have been used to simplify programming while still providing good performance and scalability. They support high-level operations on arrays e.g. matrix multiplications or atomic operators.

7 Conclusion

In this paper, we presented Shoal, a library that provides an array abstraction and rich memory allocation functions that allow automatic tuning of data placement and access depending on workload and machine characteristics. Tuning is based on memory access patterns. These are either (*i*) given by manual annotation, or, ideally, (*ii*) by modifying compilers of high-level languages to extract that information automatically. We have shown that we can use this additional information to automatically choose array implementations that increase performance on today's NUMA systems. We report an up 2x improvement for Green-Marl, a high-level graph analytics workload, without changing the Green-Marl input program. We found our memory abstraction as well as the simple policy for selecting the array implementation sufficient for current workloads and machines, but believe that future machines can benefit from a more fine-grained selection of array implementations.

References

[1] ACHERMANN, R. *Message Passing and Bulk Transport on Heterogenous Multiprocessors*. ETH Zurich, 2014. Master's Thesis, http://dx.doi.org/10.3929/ethz-a-010262232.

[2] ADVANCED MICRO DEVICES. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Online, 2013. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24593_APM_v21.pdf. Publication Number 24593. Revision 3.23.

[3] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience Distributing Objects in an SMMP OS. *ACM Transactions on Computer Systems 25*, 3 (Aug. 2007).

[4] ARAVAMUDAN, N., LITKE, A., AND MUNSON, E. libhugetlbfs. Online, 2015. http://libhugetlbfs.sourceforge.net/.

[5] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 44–54.

[6] BARRELFISH PROJECT. The Barrelfish Operating System. Online, 2015. www.barrelfish.org.

[7] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 29–44.

[8] BAUMANN, A., PETER, S., SCHÜPBACH, A., SINGHANIA, A., ROSCOE, T., BARHAM, P., AND ISAACS, R. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2009), HotOS'09, USENIX Association, pp. 12–12.

[9] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 72–81.

[10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 43–57.

[11] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (Aug. 2007), 291–312.

[12] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 519–538.

[13] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 381–394.

[14] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 33–48.

[15] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 247–256.

[16] FRANCHETTI, F., VORONENKO, Y., AND PÜSCHEL, M. FFT Program Generation for Shared Memory: SMP and Multicore. In *Proceedings of the 2006 ACM/IEEE Conference on*

Supercomputing (New York, NY, USA, 2006), SC '06, ACM.

[17] GAUD, F., LEPERS, B., DECOUCHANT, J., FUNSTON, J., FEDOROVA, A., AND QUEMA, V. Large Pages May Be Harmful on NUMA Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), USENIX Association.

[18] GICEVA, J., SALOMIE, T.-I., SCHÜPBACH, A., ALONSO, G., AND ROSCOE, T. COD: Database / Operating System Co-Design. In *CIDR* (2013), www.cidrdb.org.

[19] HAND, S. M. Self-paging in the nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (1999), pp. 73 – 86.

[20] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 349–362.

[21] HOWARD, J., DIGHE, S., VANGAL, S. R., RUHL, G., BORKAR, N., JAIN, S., ERRAGUNTLA, V., KONOW, M., RIEPEN, M., GRIES, M., ET AL. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *IEEE Journal of Solid-State Circuits 46*, 1 (2011), 173–183.

[22] INTEL CORPORATION. Intel Xeon Processor E5-1600/E5-2600/E5-4600 v2 Product Families, Datasheet - Volume One of Two. Online, 2014. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-1.pdf`, Document Number: 329187-003.

[23] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a Social Network or a News Media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[24] LI, X., GARZARÁN, M. J., AND PADUA, D. A Dynamically Tuned Sorting Library. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, p. 111.

[25] MATTSON, T. G., VAN DER WIJNGAART, R., AND FRUMKIN, M. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 38:1–38:11.

[26] NIEPLOCHA, J., HARRISON, R. J., AND LITTLEFIELD, R. J. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing 10*, 2 (June 1996), 169–189.

[27] NUMRICH, R. W., AND REID, J. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum 17*, 2 (Aug. 1998), 1–31.

[28] ORACLE CORPORATION. `madvise()` in Solaris 10. Online, 2015. `http://docs.oracle.com/cd/E19253-01/817-0547/whatsnew-updates-72/index.html`.

[29] ORACLE LABS. Fortress. Online, 2015. `https://projectfortress.java.net`.

[30] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[31] SALOMIE, T.-I., SUBASU, I. E., GICEVA, J., AND ALONSO, G. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *Proceedings of the 6th Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 17–30.

[32] SCHAEFER, C. A., PANKRATIUS, V., AND TICHY, W. F. Atune-IL: An Instrumentation Language for Auto-Tuning Parallel applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing* (Berlin, Heidelberg, 2009), Euro-Par '09, Springer-Verlag, pp. 9–20.

[33] SCHÜPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems* (2008).

[34] SILICON GRAPHICS INTERNATIONAL CORPORATION. libnuma. Online, 2015. `http://oss.sgi.com/projects/libnuma/`.

[35] SILICON GRAPHICS INTERNATIONAL CORPORATION. Origin and Onyx2 Theory of Operations Manual. Online, 2015. `http://techpubs.sgi.com/library/tpl/`

cgi-bin/getdoc.cgi?coll=0650&db=bks&
srch=&fname=/SGI_Developer/OrOn2_
Theops/sgi_html/ch02.html.

[36] SOUNDARARAJAN, V., HEINRICH, M., VERGHESE, B., GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 1998), ISCA '98, IEEE Computer Society, pp. 342–355.

[37] SUJEETH, A., LEE, H., BROWN, K., ROMPF, T., CHAFI, H., WU, M., ATREYA, A., ODERSKY, M., AND OLUKOTUN, K. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (2011), pp. 609–616.

[38] TERBOVEN, C., AN MEY, D., SCHMIDL, D., JIN, H., AND REICHSTEIN, T. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?* (New York, NY, USA, 2008), MAW '08, ACM, pp. 377–384.

[39] UPC CONSORTIUM. *UPC Language and Library Specifications*, November 2013. Version 1.3. Online. http://upc.lbl.gov/publications/upc-spec-1.3.pdf.

[40] WENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Operating Systems Review 43*, 2 (Apr. 2009), 76–85.

[41] XIONG, J., JOHNSON, J., JOHNSON, R., AND PADUA, D. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 298–308.

Thread and Memory Placement on NUMA Systems: Asymmetry Matters

Baptiste Lepers
Simon Fraser University

Vivien Quéma
Grenoble INP

Alexandra Fedorova
Simon Fraser University

Abstract

It is well known that the placement of threads and memory plays a crucial role for performance on NUMA (Non-Uniform Memory-Access) systems. The conventional wisdom is to place threads close to their memory, to collocate on the same node threads that share data, and to segregate on different nodes threads that compete for memory bandwidth or cache resources. While many studies addressed thread and data placement, none of them considered a crucial property of modern NUMA systems that is likely to prevail in the future: *asymmetric interconnect*. When the nodes are connected by links of different bandwidth, we must consider not only whether the threads and data are placed on the same or different nodes, but how these nodes are connected.

We study the effects of asymmetry on a widely available x86 system and find that performance can vary by more than 2× under the same distribution of thread and data across the nodes but different inter-node connectivity. The key new insight is that the best-performing connectivity is the one with the greatest total bandwidth as opposed to the smallest number of hops. Based on our findings we designed and implemented a dynamic thread and memory placement algorithm in Linux that delivers similar or better performance than the best static placement and up to 218% better performance than when the placement is chosen randomly.

1 Introduction

Typical modern CPU systems are structured as several CPU/memory nodes connected via an interconnect. These architectures are usually characterized by non-uniform memory access times (NUMA), meaning that the latency of data access depends on *where* (which CPU-cache or memory node) the data is located. For this reason, the placement of threads and memory plays a crucial role in performance. This property inspired many NUMA-aware algorithms for operating systems. Their insight is to place threads close to their memory [19, 12, 9], spread the memory pages across the system to avoid the overload on memory controllers and interconnect links [12], to collocate data-sharing threads on the same node [30, 31] while avoiding memory controller contention [7, 31, 10], and to segregate threads competing for cache and memory bandwidth on different nodes [34].

Further, modern operating systems aim to reduce the number of hops used for thread-to-thread and thread-to-memory communication. When balancing the load across CPUs, Linux first uses CPUs on the same node, then those one hop apart and lastly two or more hops apart. These techniques assume that the interconnect between nodes is symmetric: given any pair of nodes connected via a direct link, the links have the same bandwidth and the same latency. *On modern NUMA systems this is not the case*.

Figure 1 depicts an AMD Bulldozer NUMA machine with eight nodes (each hosting eight cores). Interconnect links exhibit many disparities: (i) Links have different bandwidths: some are 16-bit wide, some are 8-bit wide; (ii) Some links can send data faster in one direction than in the other (i.e., one side sends data at 3/4 the speed of a 16-bit link, while the other side can only send data at the speed of an 8-bit link). We call these links *16/8-bit links*; (iii) Links are shared differently. For instance the link between nodes 4 and 3 is only used by these two nodes, while the link between nodes 2 and 3 is shared by nodes 0, 1, 2, 3, 6 and 7; (iv) Some links are unidirectional. For instance node 7 sends requests directly to node 3, but node 3 routes its answers via node 2. This creates an asymmetry in read/write bandwidth: node 7 can write at 4GB/s to node 3, but can only read at 2GB/s.

The asymmetry of interconnect links has dramatic and at times surprising effects on performance. Figure 2 shows the performance of 20 different applications on

Machine A (Figure 1)[1]. Each application runs with 24 threads and so it needs three nodes to run on. We vary *which* three nodes are assigned to the application and hence the *connectivity* between the nodes. The relative placement of threads and memory on those nodes is *identical* in all configurations. The only difference is *how the chosen nodes are connected*. The figure shows the performance on the best-performing and the worst-performing subset of nodes for that application compared to the average (obtained by measuring the performance on all 336 unique subsets of nodes and computing the mean). We make several observations. First, the performance on the best subset is up to 88% faster than the average, and the performance on the worst subset is up to 44% slower. Second, the maximum performance difference between the best and the worst subsets is 237% (for facerec). Finally, the mean difference between the best and worst subsets is 40% and the median 14%. In the following section we demonstrate that these performance differences are caused by the asymmetry of the interconnect between the nodes.

This work makes the following contributions:

- We quantify and characterize the effects of asymmetric interconnect on a commercial x86 NUMA system. The key insight is that the best-performing connectivity is the one with the greatest total bandwidth as opposed to the smallest number of hops.
- We design, implement and evaluate a new algorithm that dynamically picks the best subset of nodes for applications requiring more than one node. This algorithm places the clusters of threads and their memory to ensure that the most intensive CPU-to-CPU or CPU-to-memory communication occurs between the best-connected nodes. Our evaluation shows that this algorithm performs as well as or better than the best set of nodes chosen statically.
- Our implementation revealed a limitation in hardware counters, which prevented us from having certain flexibilities in the algorithm. We discuss them and make suggestions for improvements.

The paper is structured as follows. Section 2 studies the impact of interconnect asymmetry and discusses challenges in catering to this phenomenon in an operating system. Section 3 discusses current architectural trends and shows that machines are becoming increasingly asymmetric. Section 4 presents our algorithm, and Section 5 reports on the evaluation. Section 6 discusses related work, and Section 7 provides a summary.

[1] Additional details about the applications and the machine are provided in Section 5.

2 The Impact of Interconnect Asymmetry on Performance

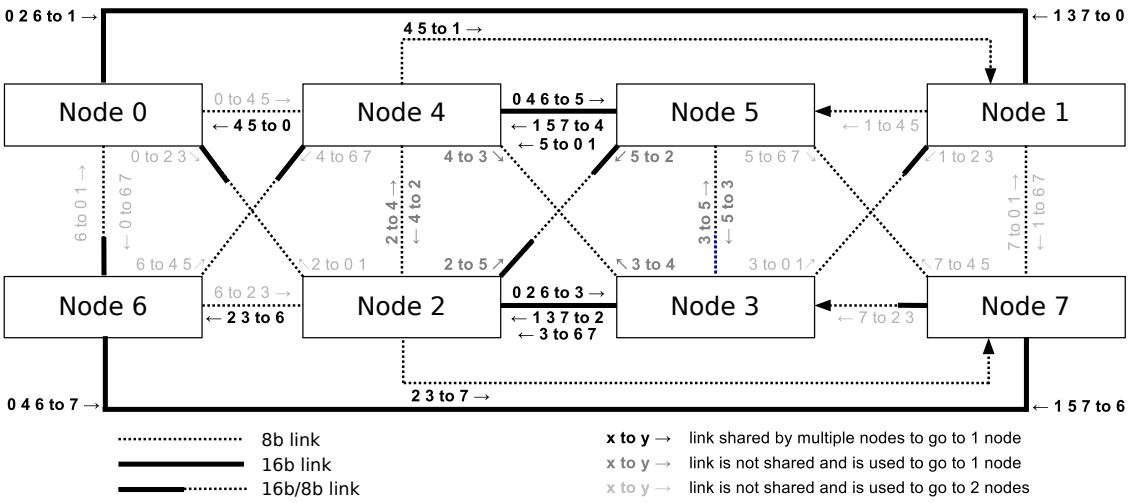
To explain the reasons behind the performance reported in Figure 2, Figure 3 shows the memory latency measured when the application runs on the best and worst node subsets relative to the latency averaged across all 336 possible subsets. We can see that memory accesses performed by facerec are approximately 600 cycles faster when running on the best subset of nodes relative to the average, and 1400 cycles faster relative to the worst. We can see that the latency differences are tightly correlated with the performance difference between configurations. The applications that are the most affected by the choice of nodes on which to run are also those with the highest difference in the memory latencies.

To further understand the cause of very high latencies on "bad" configurations we analyzed streamcluster – an application from the Parsec [26] benchmark suite, which was among the most affected by the placement of its threads and memory. In the following experiment we run streamcluster with 16 threads on two nodes. Table 1 presents the salient metrics for each possible two-node subset. Depending on which two nodes we chose, we observe large (up to 133%) disparities in performance. The data in Table 1 leads to several crucial observations:

- As shown earlier, performance is correlated with the latency of memory accesses.
- Surprisingly, the latency of memory accesses is not correlated with the number of hops between the nodes: some two-hop configurations (shown in bold) are faster than one-hop configurations.
- The latency of memory accesses is actually correlated with the *bandwidth* between the nodes. Note that this makes sense: the difference between one-hop vs. two-hop latency is only 80 cycles when the interconnect is nearly idle. So a higher number of hops alone cannot explain the latency differences of thousands of cycles.

Bandwidth between the nodes matters more than the distance between them.

So the problem of choosing a "good" subset of nodes is essentially the problem of ***the placement of threads and memory pages on a well-connected subset of nodes***. When an application executes on only two nodes on a machine similar to the one used in the aforementioned experiments, the placement on the nodes connected with the widest (16-bit) link is always the best because it maximizes the bandwidth and minimizes the latency between the nodes. However, when an application needs more than two nodes to run, no configuration exists with 16-bit links between every pair of nodes, so we must decide



(Machines A and B)

Figure 1: Modern NUMA systems, with eight nodes. The width of links varies, some paths are unidirectional (e.g., between 7 and 3) and links may be shared by multiple nodes. Machine A has 64 cores (8 cores per node - not represented in the picture) and machine B has 48 cores (6 cores per node). Not shown in the picture: the links between nodes 4 and 1 and between nodes 2 and 7 are bidirectional on machine B. This changes the routing of requests from node 7 to 2 and node 1 to 4.

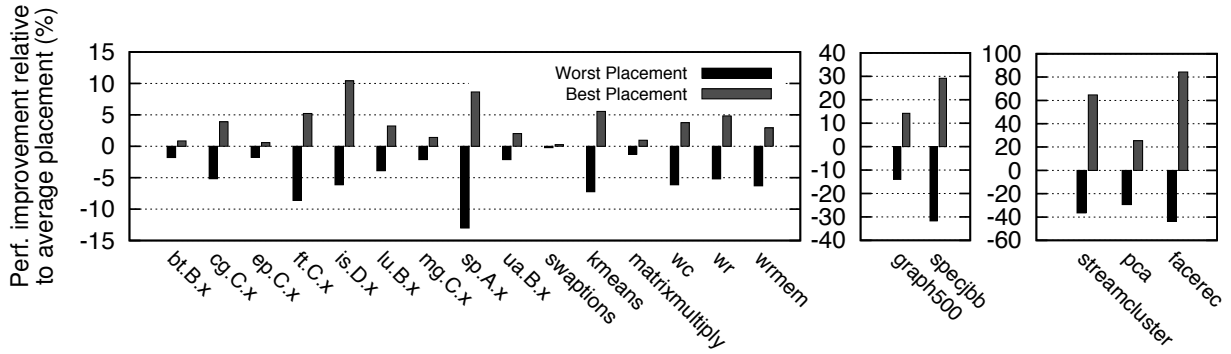


Figure 2: Performance difference between the best, and worst thread placement with respect to the average thread placement on Machine A. Applications run with 24 threads on three nodes. Graph500, specjbb, streamcluster, pca and facerec are highly affected by the choice of nodes and are shown separately with a different y-axis range.

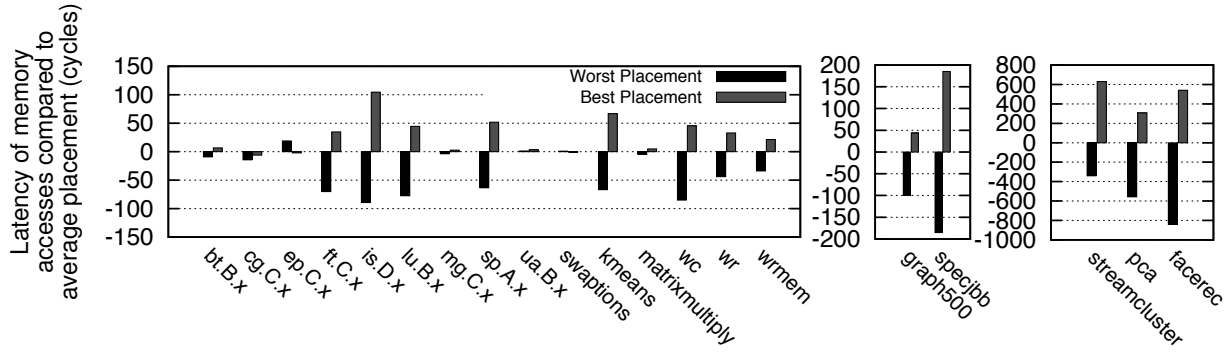
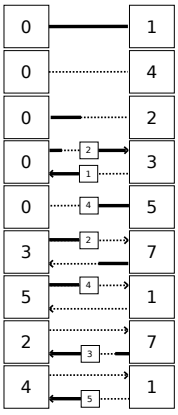


Figure 3: Difference in latency of memory accesses between the best, and worst thread placement with respect to the average node configuration on Machine A. Positive numbers mean that memory accesses are faster than the average.



Master thread node	Execution Time (s)	Diff with 0-1 (%)	Latency of memory accesses (cycles) (compared to 0-1(%))		% accesses via 2-hop links	Bandwidth to the "master" node (MB/s)
-	148	0%	750		0	5598
-	228	56%	1169	(56%)	0	2999
0	228	56%	1179	(57%)	0	2973
2	168	15%	855	(14%)	0	4329
0	340	133%	1527	(104%)	98	1915
3	**185**	27%	1040	(39%)	98	3741
0	340	133%	1601	(113%)	98	1903
5	**228**	56%	1206	(61%)	98	2884
3	185	27%	1020	(36%)	0	3748
7	338	132%	1614	(115%)	98	1928
1	338	132%	1612	(115%)	98	1891
5	230	58%	1200	(60%)	0	2880
2	**167**	15%	867	(16%)	98	3748
7	225	54%	1220	(63%)	0	3014
4	230	58%	1205	(60%)	0	2959
1	**226**	55%	1203	(60%)	98	2880

Table 1: Performance of streamcluster executing with 16 threads on 2 nodes on machine A. The performance depends on the connectivity between the nodes on which streamcluster is executing and on the node on which the master thread is executing. Numbers in bold indicate 2-hops configurations that are as fast or faster than some 1-hop configurations.

which nodes to pick. When there is more than one application running, we need to decide how to allocate the nodes among multiple applications.

Nodes	% perf. relative to best subset	
	streamcluster	SPECjbb
0, 1, 3, 4, and 7	-64%	0% (best)
2, 3, 4, 5, and 6	0% (best)	-9.4%

Table 2: Performance of streamcluster and SPECjbb on two different set of nodes on machine A, relative to the best set of nodes for the respective application.

In this paper, we present a new thread and memory placement algorithm. Designing such an algorithm for asymmetrically connected NUMA systems is challenging for the following reasons:

Efficient online measurement of communication patterns is challenging: The algorithm must measure the volume of CPU-to-CPU and CPU-to-memory communication for different threads in order to determine the best placement when we cannot run the entire application on the best connected nodes. This measurement process must be very efficient, because it must be done continuously in order to adapt to phase changes.

Changing the placement of threads and memory may incur high overhead: Frequent migration of threads may be costly, because of the associated CPU overhead, but most importantly because cache affinity is not preserved. Moreover, when threads are migrated to "better" nodes, it might be necessary to migrate their memory in order to avoid the overhead of remote accesses and overloaded memory controllers. Migrating large amounts of memory can be extremely costly. Thus, thread migration must be done in a way that minimizes memory migration.

Accomodating multiple applications simultaneously is challenging: Applications have different communication patterns and are thus differently impacted by the connectivity between the nodes they run on. As an illustration, Table 2 presents the performance of streamcluster and SPECjbb executing on two different sets of five nodes (the best set of nodes for the two applications, respectively). The two applications behave differently on these two sets of nodes: streamcluster is 64% slower on the best set of nodes for SPECjbb than on its own best set. The algorithm must, therefore, determine the best set of nodes for every application. Furthermore, it cannot always place each application on its best set of nodes, because applications may have conflicting preferences.

Selecting the best placement is combinatorially difficult: The number of possible application placements on an eight-node machine is very large (e.g., 5040 possible configurations for four applications executing on two nodes). So, (i) it is not possible to try all configurations *online* by migrating threads and then choosing the best configurations, and (ii) doing even the simplest computation involving "all possible placements" can still add a significant overhead to a placement algorithm.

Before describing how we addressed these challenges, we briefly discuss architectural trends and the increasing impact of interconnect asymmetry.

3 Architectural trends

Asymmetric interconnect is not a new phenomenon. Nevertheless, we show in this section that its effects on

performance are increasing as machines are built with more nodes and cores. For that purpose, we measured the performance of streamcluster on four different asymmetric machines: two recent machines with 64 and 48 cores respectively, and 8 nodes (Machines A and B, Figure 1), and two older machines with 24 and 16 cores respectively, and 4 nodes (Machines C and D, not depicted). Machines A and B have highly asymmetric interconnect; Table 1 lists all possible interconnect configurations between 2 nodes. Machines C and D have a less pronounced asymmetry. Machine C has full connectivity, but two of the links are slower than the rest. Machine D has links with equal bandwidth, but two nodes do not have a link between them.

Table 3 shows the performance of streamcluster with 16 threads on the best-performing and the worst-performing set of nodes on each machine. The performance difference between the best and worst configurations increases with the number of cores in the machine: from 3% for the 16-core machine to 133% for the 64-core machine. We explain this as follows: (i) On the 16-core Machine D, the only difference between configurations is the longer wire delay between the nodes that are not connected via a direct link. This delay is not significant compared to the extra latency induced by bandwidth contention on the interconnect. (ii) The CPUs on 24-core Machine C have a low frequency compared to the other machines. As a result, the impact of longer memory latency is not as pronounced. More importantly, the network on this machine is still a fully connected mesh, so there is less asymmetry than on Machines A and B. (iii) The 48- and 64-core Machines B and A offer a wider range of bandwidth configurations, which increases the difference between the best and the worst placements. The 64-core machine is more affected than the 48-core machine because it has more cores per node, which increases the effects of bandwidth contention.

If this trend holds across different machines and architectures, then it is clear that the effects of asymmetry can no longer be ignored.

Machine	Best time	Worst time	Difference
A (64 cores)	148s	340s	133%
B (48 cores)	149s	277s	85%
C (24 cores)	171s	229s	33%
D (16 cores)	255s	262s	3%

Table 3: Performance of streamcluster executing on 2 nodes on machine A, B, C, and D. The performance of streamcluster depends on the placement of its threads. The impact of thread placement is more important on recent machines (A and B) than on older ones (C and D).

4 Solution

4.1 Overview

We designed *AsymSched*, a thread and memory placement algorithm that takes into account the bandwidth asymmetry of asymmetric NUMA systems. *AsymSched*'s goal is to maximize the bandwidth for *CPU-to-CPU communication*, which occurs between threads that exchange data, and *CPU-to-memory communication*, which occurs between a CPU and a memory node upon a cache miss. To that end, *AsymSched* places threads that perform extensive communication on relatively well-connected nodes and places the frequently accessed memory pages such that the data requests are either local or travel across high-bandwidth paths.

AsymSched is implemented as a user level process and interacts with the kernel and the hardware using system calls and `/proc` file system, but could also be easily integrated with the kernel scheduler if needed.

AsymSched continuously monitors hardware counters to detect opportunities for better thread placements. The thread placement decision occurs every second, and *AsymSched* only migrates threads when the benefits of migration is expected to exceed its overhead. The placement of memory pages follows the placement of threads.

AsymSched relies on three main techniques to manage threads and memory: (i) **Thread migration**: changing the node where a thread is running. (ii) **Full memory migration**: migrating all pages of an application from one node to another. Full memory migration is performed using a new system call that we present in Section 4.3. (iii) **Dynamic memory migration**: migrating only the pages that an application actively accesses. Dynamic memory migration uses *Instruction-Based Sampling* (IBS), a profiling mechanism available in AMD processors[2], to sample memory accesses and to identify the most frequently accessed pages. Then, the pages that are not shared are migrated to the node that accesses them. Shared pages are spread across multiple nodes. We use the same algorithm and techniques described in [12]; we therefore omit further details on dynamic memory migration.

4.2 Algorithm

AsymSched relies on 3 components. The *measurement* component continuously computes salient metrics. The *decision* component uses these metrics to periodically compute the best thread placements. The *migration* component migrates threads and memory. Table 4 presents the definitions relevant to *AsymSched*. Algorithm 1 summarizes the algorithm.

[2]Intel processors have a similar mechanism called *Precise Event-Based Sampling* (PEBS).

Per cluster (C) statistics	
C_{rbw}	Remote bandwidth: the number of memory accesses performed by threads in the cluster to another node, i.e., *remote accesses*.
C_{weight}	"Weight" of the cluster. Clusters with the highest weights are scheduled on the nodes with the highest interconnect bandwidth. By default $C_{weight} = log(C_{rbw})$.
$C_{bw}(P)$	Maximum bandwidth of C threads on placement P.
Per placement (P) statistics	
P_{wbw}	Weighted total bandwidth of P. Is equal to the sum of the $C_{bw}(P) * C_{weight}$ for every placed cluster C.
P_{mm}	Amount of memory that has to be migrated to use this placement.
Per application (A) statistics	
A_{tm}	Time already spent migrating memory.
A_{tt}	Dynamic running time of the application.
$A_{mm}[node]$	Resident set size of the application, per node.
A_{oldA}	Percentage of memory accesses performed on nodes on which the application *was* scheduled but is no longer scheduled on.

Table 4: Definitions relevant to *AsymSched*.

Measurement. *AsymSched* continuously gathers the metrics characterizing the volume of CPU-to-CPU and CPU-to-memory communication. On our experimental system there is a single counter that captures both: it measures the number of *data accesses performed by a CPU to a given* node and includes both the accesses to cached data (CPU-to-CPU communication) and to the data located in RAM (CPU-to-memory communication). Ideally, we would like to measure the communication volume from every CPU to every other CPU, however the counters available on AMD systems do not offer this opportunity. One alternative is to use AMD's Instruction Based Sampling (IBS)[3]. Unfortunately, to accurately track CPU-to-CPU communication, IBS requires a high sampling rate, and that introduces too much overhead. Lightweight Profiling (LWP), a new profiling facility of AMD processors, has a smaller overhead, but is only partially implemented in current processors. Despite these limitations, we believe that it is only a matter of time until they are addressed in the mainstream hardware, so for the time being we use the following work-around.

The algorithm described below relies on detecting which threads share data. Since we can only practically measure the communication between a CPU and a remote node, but not CPU-to-CPU communication (either across or within nodes), we make the following simplifying assumptions: *(a)* a thread may share data with any other thread running on the same node, *(b)* if there is a high volume of communication between a CPU and a node, a thread running on that CPU may share data with any thread of the same application on that node. To reduce the occurrence of situations where we assume data sharing while in reality there is none, we initially collocate threads from the same application on the same node, to the extent possible. Data sharing is far more common between threads from the same application than between threads from different applications.

The downside of this simplifying assumption is that we may unnecessarily keep a group of threads collocated on the same node even if they do not share data. But there is also an important benefit: characterizing the communication in terms of CPU-to-node keeps the number of sharing relationships to consider small and reduces the complexity of the algorithm.

Decision. The following description relies on definitions in Table 4. **Step 1**: *AsymSched* groups threads of the same application that share data in virtual *clusters*. A cluster is simply a list of threads that share data. It then assigns a weight C_{weight} to each cluster; clusters with the highest weights will be scheduled on the nodes with the best connectivity. By default clusters are weighed by the logarithm of the number of remote memory accesses performed by their threads ($C_{weight} = log(C_{rbw})$). The logarithm deemphasizes small differences in C_{rbw} between the clusters, while preserving large differences. This makes it much easier for the algorithm to pick out the clusters with a relatively high C_{rbw} and place them on well-connected nodes.

Step 2: *AsymSched* computes possible *placements* for all the clusters. A placement is an array mapping clusters to nodes. It works at the node granularity, so the number of possible placements is equal to the number of node permutations (i.e., migrating all threads of node X to node Y and vice versa). As this number can be very large, it is important that *AsymSched* not test all possible placements. Section 4.3 details how *AsymSched* avoids testing all possible placements. For each placement P, *AsymSched* computes the maximum bandwidth $C_{bw}(P)$ that each cluster C would receive if it were put in this placement. Each placement is assigned a performance metric, P_{wbw}, the *weighted bandwidth* of P, defined as $P_{wbw} = \sum_{C \in clusters} C_{bw}(P) * C_{weight}$. The higher P_{wbw}, the higher the bandwidth available to clusters that perform a lot of remote communications. The definition of C_{weight} implies that our algorithm aims to optimize the overall communication bandwidth across all applications. The

[3]PEBS, Precise Event-Based Sampling, is a similar feature on Intel systems.

algorithm can be easily changed to optimize a different metric, e.g., one that takes into account application priorities, by changing the definition of C_{weight}.

Step 3: *AsymSched* filters placements to keep only those that have a *weighted bandwidth* value at least equal to 90% of the maximum weighted bandwidth. Among these remaining placements *AsymSched* chooses those that will minimize the number of page migrations.

Step 4: For each application, *AsymSched* estimates the overhead of memory migration assuming the cost of 0.3s per GB, which was derived on our system using simple experiments. If the overhead is deemed too high, the new placement will not be applied. Another goal here is to avoid migrating the applications back and forth because of recurring changes in communication patterns and accumulating a high overhead. To that end, *AsymSched* keeps track of the total time already spent doing memory migration for the application: A_{tm}. If that time plus the estimated cost of additional migration ($\sum_{n \in migrated_nodes} A_{mm}[n] * 0.3$) exceeds 5% of the running time of the application (A_{tt}), then *AsymSched* does not apply the new thread placement. We chose 5% as a reasonable maximum overhead value. In practice, the highest overhead we observed was around 3%.

Migration. Step 1: *AsymSched* migrates threads using system calls that are available in the Linux kernel.

Step 2: *AsymSched* relies on *dynamic migration* to migrate the subset of pages that the application uses. If, after two seconds, the application still performs more than 90% of its memory accesses on the nodes where it was previously running ($A_{oldA} > 90\%$), then *AsymSched* concludes that *dynamic migration* was not able to migrate the working set of the application and performs a *full memory migration*.

The Measurement, Decision and Migration phases described above are performed continuosly to account for phase changes in applications and other dynamics.

Algorithm 1 *AsymSched* algorithm

```
1: if Threads of nodes N1 and N2 access a common memory controller and threads of N1 and N2 have the same pid then
2:     Put all threads running on N1 and N2 in a cluster C and Increase C_rbw
3: end if
4: Compute relevant cluster placements
5: Max_wbw = 0
6: for all P ∈ computed placements do
7:     P_wbw = Σ_{C∈clusters} C_bw(P) * C_weight
8:     Max_wbw = max(Max_wbw, P_wbw)
9: end for
10: for all P ∈ computed placements do
11:     Skip if P_wbw <90%*Max_wbw
12:     Compute P_mm
13: end for
14: Choose the placement with the lowest P_mm
15: for all A ∈ migrated applications do
16:     if A_tm + Σ_{n∈migrated_nodes} A_mm[n] * 0.3 > 0.05*A_tt then
17:         Do not change thread placement
18:     end if
19: end for
20: Migrate threads
21: Use dynamic memory migration
22: After 2 seconds:
23: for all A ∈ migrated applications do
24:     if A_oldA > 90% then
25:         Fully migrate memory of A
26:     end if
27: end for
```

4.3 Optimizations and tricks

We integrated several optimizations within *AsymSched* to ensure that it runs accurately and with low overhead.

Fast memory migration. When *AsymSched* performs full memory migration, all the pages located on one node are migrated to another node. The applications we tested have large working sets (up to 15GB per node), and migrating pages is costly. We measured that migrating 10GB of data using the standard `migrate_pages` system call takes 51 seconds on average, making migration of large applications impractical.

Therefore, we designed a new system call for memory migration. This system call performs memory migration without locks in most cases, and exploits the parallelism available on multicore machines. Using our system call, migrating memory between two nodes is on average 17× faster than using the default Linux system call and is only limited by the bandwidth available on interconnect links. Unlike the Linux system call, our system call can migrate memory from multiple nodes simultaneously. So if we are migrating the memory simultaneously between two pairs of nodes that do not use the same interconnect path, our system call will run about 34 times faster.

Fast migration works as follows. (i) First, we "freeze" the application by sending SIGSTOP to all its threads. Freezing the application is done to ensure that the application does not allocate or free pages during migration. This allows removing many locks taken by the Linux memory migration mechanism, and since up to 80% of migration time can be wasted waiting on locks, the resulting performance improvements are significant. (ii) Second, we parse the memory map of the application and store all pages in an array. We then launch worker

threads on the node(s) on which the application is scheduled. Worker threads process pages stored in the array in chunks of 30 thousand. The old page is unmapped, data is copied to a new page, the new page is remapped, and the old page is freed. Shared pages or pages that are currently swapped are ignored.

Avoiding evaluation of all possible placements. The number of all possible thread placements on a machine can be very large. We use two techniques to avoid computing all thread placements: (i) A lot of thread placement configurations are "obviously" bad. For instance, when a communication-intensive application uses two nodes, we only consider configurations with nodes connected with a 16-bit link. (ii) Several configurations are equivalent (e.g., the bandwidth between nodes 0 and 1 and between nodes 2 and 3 is the same). To avoid estimating the bandwidth of *all* placements, we create a hash for each placement. The hash is computed so that equivalent configurations have the same hash. Using simple dynamic programming techniques, we only perform computations on non-equivalent configurations.

These two techniques allow skipping between 67% and 99% of computations in all tested configurations with clusters of 2, 3 or 5 nodes (e.g., with 4 clusters of 2 nodes, we only evaluate 20 configurations out of 5040).

5 Evaluation

Our goal is to evaluate the impact of asymmetry-aware thread placement *in isolation* from other effects, such as those stemming purely from collocating threads that share data on the same node. Performance benefits of sharing-aware thread clustering are well known [30]. *AsymSched* clusters threads that share data as described in the Section 4; the Linux thread scheduler, however, does not. We experimentally observed that Linux performed worse than clustered configurations. E.g., when graph500 and specjbb are scheduled simultaneously, both run 23% slower on Linux than on an average clustered placement. Since comparing Linux to *AsymSched* would not be meaningful because of that, we instead compare *AsymSched*[4] to the best and the worst static placements of data-sharing thread clusters. We also compare the average performance achieved under all static placements that are unique in terms of connectivity. We obtain all unique static placements with respect to connectivity by examining the topology of the machine. There are 336 placements for single-application scenarios and 560 placements for multi-application scenarios.

Further, we want to isolate the effects of thread placement with *AsymSched* from the effects of dynamic memory migration. To that end, we compare *AsymSched* to the subset of our algorithm that performs the dynamic placement of memory only, turning off the parts performing thread placement.

[4]When running *AsymSched*, thread clusters are initially placed on a randomly chosen set of nodes.

5.1 Experimental platform

We evaluate *AsymSched* on machine A. It is equipped with four AMD Opteron 6272 processors, each with two NUMA nodes and 8 cores per node (64 cores in total). The machine has 256GB of RAM and uses HyperTransport 3.0. It runs Linux 3.9.

We used several benchmark suites: the NAS Parallel Benchmarks suite [6] which is composed of numeric kernels, MapReduce benchmarks from Metis [25], parallel applications from Parsec [26], Graph500 [1], a graph processing application with a problem size of 21, FaceRec from the ALPBench benchmark suite [11], and SPECjbb [2] running on OpenJDK7. From the NAS and Parsec benchmark suites we picked the benchmarks that run for at least 15 seconds, and that can be executed with arbitrary numbers of threads. The memory usage of the benchmarks ranges from 518MB for EP from the NAS suite to 34,291MB for IS from NAS. Except for SPECjbb, we use the execution time of applications as performance indicator. SPECjbb runs during a fixed amount of time; we use the throughput (measured in SPECjbb bops) as performance indicator.

5.2 Single application workloads

The results are presented in Figure 4. *AsymSched* always performs close to the best static thread placement. In a few cases where it does not, the difference is not statistically significant. For applications that produce the highest degree of contention on the interconnect links (streamcluster, pca, and facerec), *AsymSched* achieves much better performance than the best thread placement, because the dynamic memory migration component balances memory accesses across nodes, thus reducing contention on interconnect links and memory controllers.

We also observe that dynamic memory migration without the migration of threads is not sufficient to achieve the best performance. More precisely, dynamic memory migration alone often achieves performance close to average. Moreover, it produces a high standard deviation for many benchmarks: the minimum and maximum performance often being the same as that of the best and worst static thread placement. For instance, on SPECjbb, the difference between the minimum and maximum performance with dynamic memory migration alone is 91%.

In contrast, *AsymSched* produces a very low standard deviation for most benchmarks. Two exceptions are is.D

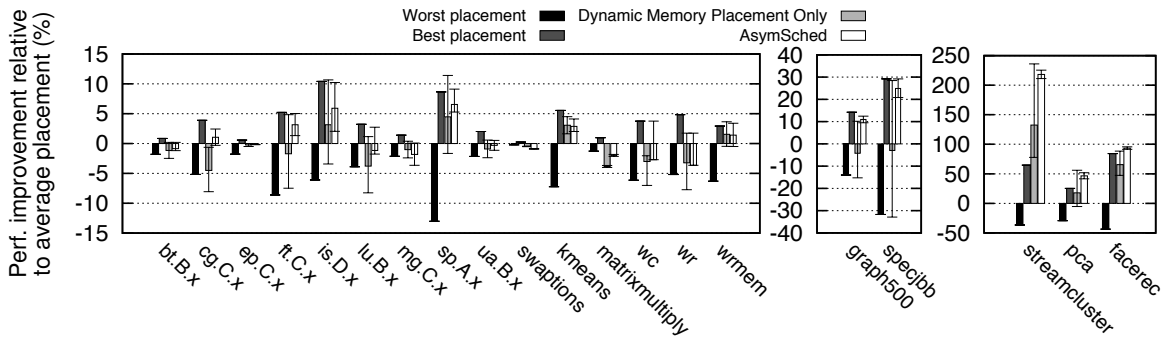


Figure 4: Performance difference between the best and worst static thread placement, dynamic memory placement, *AsymSched* and the average thread placement on machine A. Applications run with 24 threads on 3 nodes.

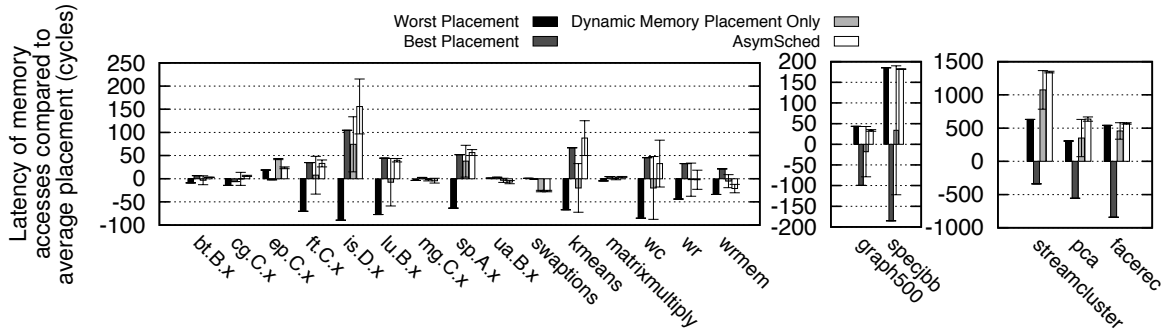


Figure 5: Memory latency under the best and worst static thread placement, dynamic memory placement, *AsymSched* and the average thread placement on machine A. Applications run with 24 threads on 3 nodes.

and SPECjbb. This is because in both cases, *AsymSched* migrates a large amount of memory. Both applications become memory intensive after an initialization phase, and *AsymSched* starts migrating memory only after the entire working set has been allocated. For instance, in the case of is.D, *AsymSched* migrates between 0GB and 20GB, depending on the initial placement of threads.

Figure 5 shows the latency of memory accesses compared to the average. For most applications, the dynamics of latency closely matches that of the performance. A few exceptions are is.D, lu.B and kmeans. For is.D, the latency is drastically improved by *AsymSched* but the impact on performance is not visible because of the time lost performing memory migrations. Lu.B is extremely memory intensive during its first seconds of execution, but performs very few memory accesses thereafter; *AsymSched* improves this initial phase but has no impact on the rest of the running time. Kmeans is very bursty; placing its threads has a huge impact on the latency of memory accesses performed during bursts of memory accesses but not on the rest of the execution.

5.3 Multi application workloads

We evaluate several multi-application workloads using the applications studied in section 5.2. We chose four applications that benefit to various degrees from *AsymSched*: streamcluster (benefits to a high degree), SPECjbb (benefits to a moderate degree), graph500 (benefits to a small degree), and matrixmultiply (does not benefit). Some of these applications have different phases during their execution; for instance, streamcluster processes its input set in five distinct rounds, and SPECjbb spends significant amount of time initializing data before emulating a three-tier client/server system.

Figure 6 presents the performance on multi application workloads. We chose two different clustering configurations: (i) Three applications executing on three, three and two nodes, respectively; (ii) Two applications executing on five and three nodes respectively.

In all workloads, *AsymSched* achieves performance that is close or better than the best static thread placement on Linux. Furthermore, it produces a very low standard deviation. In constrast, dynamic memory migration alone exhibits high standard deviation and, like with single application workloads, is unable to improve performance for Graph500 and SPECjbb.

AsymSched significantly improves the latency of applications that benefit from thread and memory migration (Figure 7), in particular for streamcluster. This is because *AsymSched* chooses configurations in which the links used by streamcluster are not shared with any other application.

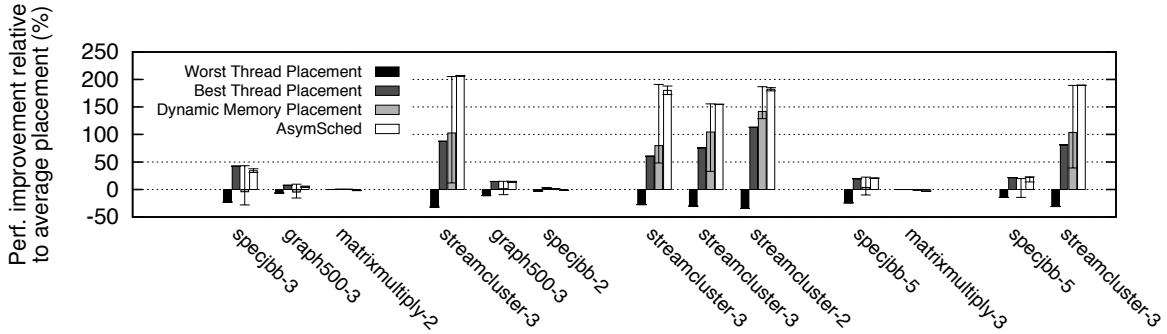


Figure 6: Performance difference between the best thread placement, the worst thread placement, dynamic memory placement, *AsymSched* and the average thread placement of applications on machine A. The numbers appended to the name of applications specify the number of nodes on which the application runs.

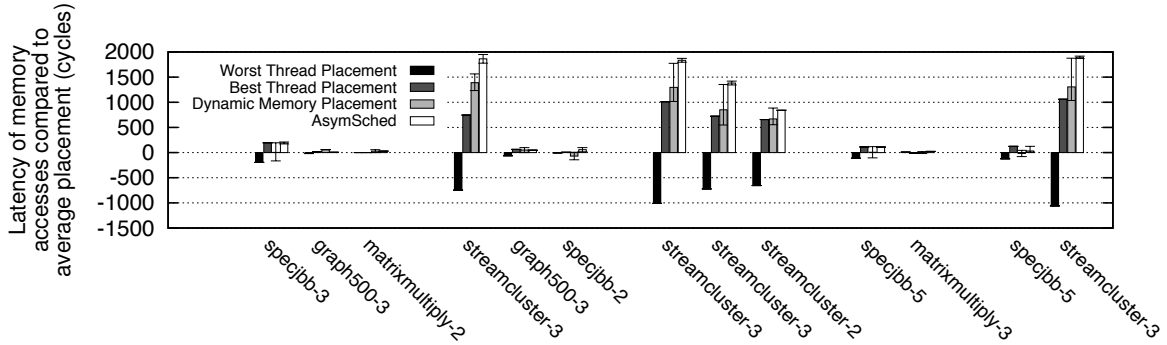


Figure 7: Performance difference between the best thread placement, the worst thread placement, dynamic memory placement, *AsymSched* and the average thread placement of applications on machine A.

	cg.B	ft.C	is.D	sp.A	streamcluster	graph500	specJBB
Migrated memory (GB)	0.17	2.5	20	0.1	0.15	0.3	10
Average time - Linux syscall (ms)	860	12700	101000	490	750	1500	50500
Average time - fast migration (ms)	51	380	3050	30	45	90	1500

Table 5: Average amount of migrated memory for various applications running on 3 nodes and required time to perform the migration using the standard Linux system call and using fast memory migration.

5.4 Overhead

The main overhead of *AsymSched* is due to memory migration. This explains why we implemented a custom system call (see Section 4.3). Table 5 compares the migration time when running the standard Linux system call and when running our custom system call. For instance, for is.D, migration takes 101 seconds using the Linux system call (50% overhead), but only 3 seconds using our custom system call (1.5% overhead). To keep the overhead low, *AsymSched* performs migrations only if the predicted overhead is below 5%. In practice, the maximum migration overhead we observed was 3%.

The cost of collecting metrics and computing cluster placement is below 0.5% on all studied applications. Moreover, *AsymSched* requires less than 2MB of RAM.

The overhead of thread migration is negligible and we did not observe any noticeable effect of thread migrations on cache misses.

Finally, when dynamic memory placement is used, IBS sampling incurs a light overhead (within 2% in our experiments) and statistics on memory accesses are stored in about 20MB of RAM.

5.5 Discussion - Applicability on future NUMA machines

We believe that the findings and the solution presented in this paper are likely to be applicable on future NUMA systems. First, we believe that the clustering and placement techniques used in *AsymSched* can scale on machines with a much larger number of nodes. With very simple heuristics we were able to avoid computing up to 99% of the possible thread placements. Such optimizations will still likely be possible on future machines, as machines are usually made of multiple identical cores/sockets (e.g., our 64-core machine has 4 identical sockets). On machines that offer a wider diversity

of thread placements, a possibility is to use statistical approaches, such as that of Radojkovic et al. [27] to find good thread placements with a bounded overhead.

Furthermore, *AsymSched* can easily be adapted to different optimization goals. On current NUMA machines, maximizing the bandwidth between threads was the key to achieving good performance, but our solution could be easily adapted to take other metrics into account.

6 Related Work

NUMA optimizations and contention management: Optimizing thread and memory placement on NUMA systems has been extensively studied [8, 9, 20, 33, 19, 12, 9, 7, 31, 5, 23, 22, 10]. However, as shown in [19, 12, 5, 23], contention on interconnect links and memory controllers remains a major source of inefficiencies on modern NUMA machines. Our work complements these previous studies by minimizing contention on interconnect links on asymmetrically connected NUMA systems. We adopt a dynamic memory management algorithm presented in [12] for memory placement, but the key contribution of our work is the algorithm that efficiently computes the placement of threads on nodes to maximize the bandwidth between communicating threads.

Several extensions to Linux improve data-access locality on NUMA systems, but do not improve the bandwidth for communicating threads and do not address interconnect asymmetry. For example, Sched/NUMA [32] adds the notion of a "home node": when scheduling threads, Linux will try to collocate threads and data on the "home node" of the corresponding process. While this may improve communication bandwidth for applications with the number of threads not exceeding the number of cores in a node, it does not address applications spanning several nodes. Another extension, called AutoNUMA [4], implements locality optimizations by migrating pages on the nodes from which they are accessed. *AsymSched* uses a similar dynamic algorithm to migrate memory, but unlike AutoNUMA it also places threads so as to optimize communication bandwidth.

Several studies addressed contention for the memory hierarchy of UMA systems [16, 24, 34] by segregating competing threads on different nodes. None of these systems, however, addressed contention on the interconnect.

Scheduling on asymmetric architectures: Several thread schedulers catered to the asymmetry of CPUs [29, 15, 21, 17]. They optimize thread placement on processors with asymmetric characteristics (e.g., different frequencies, or different hardware features). The techniques used to address processor asymmetry are fundamentally different than those needed to address interconnect asymmetry, so there is no overlap with our study.

Thread clustering: Pusukuri et al. [18] cluster threads based on lock contention and memory access latencies. Kamali [14] and Tam [30] proposed algorithms that cluster threads that share data on the same shared cache. *AsymSched* uses a similar high-level idea: it samples hardware counters to detect communicating threads and place them onto a well-connected nodes. However, the problem addressed in *AsymSched* (asymmetric interconnect) and the specific algorithm proposed is quite different from those in the aforementioned studies.

Radojkovic et al. [28] present a scheduler that takes into account resource sharing inside a processor. They model the benefits and drawbacks of data and instruction cache sharing between threads, and they schedule threads on the the set of cores that will maximize performance. Their solution explores all possible thread placements. Their follow-up work [27] refines the solution to use a statistical approach to find the optimal placement. Our solution could benefit from a similar technique on machines with a much larger number of dissimilar nodes.

Network optimizations: Network traffic optimization is a well studied problem. Machines are often interconnected with asymmetric Ethernet links; optimizing the bandwidth on asymmetric NUMA systems shares a lot of similarities with optimizations problems found in these systems. For instance, Volley [3] proposed an algorithm to place data used by Cloud services. As *AsymSched*, this algorithm takes into account the available bandwidth between nodes (that are geographically distributed computers in their case) in order to optimize performance.

Hermenier et al. [13] present a consolidation manager for distributed systems. The goal of their system is to minimize energy consumption in a cluster. To this end, they place virtual machines on the smallest possible number of physical machines while meeting certain performance constraints. They model the problem as the multiple knapsack problem and use a constraint-satisfaction solver to find good placements. *AsymSched* could use a similar technique, but we found that on existing systems a much simpler solution was sufficient.

7 Conclusion

We showed that the asymmetry of the interconnect in modern NUMA systems drastically impacts performance. We found that the performance is more affected by the bandwidth between nodes than by the distance between them. We developed *AsymSched*, a new thread and memory placement algorithm that maximizes the bandwidth for communicating threads.

As the number of nodes in NUMA systems increases, the interconnect is less likely to remain symmetric. *AsymSched* design principles will, therefore, be of growing importance in the future.

References

[1] Graph500 reference implementation. `http://www.graph500.org/referencecode`.

[2] specjbb2005 - industry-standard server-side java benchmark (j2se 5.0). `http://www.spec.org/jbb2005/`.

[3] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[4] AutonNUMA: the other approach to NUMA scheduling. LWN.net, March 2012. `http://lwn.net/Articles/488709/`.

[5] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT*, 2010.

[6] D.H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R.A Fatoohi, P. O. Frederickson, T. A Lasinski, R. S. Schreiber, H.D. Simon, V. VenkataKrishnan, and S.K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165, Nov 1991.

[7] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[8] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Management. In *SOSP*, 1989.

[9] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX SEDMS*, 1993.

[10] J. Mark Bull and Chris Johnson. Data distribution, migration and replication on a ccnuma architecture. In *In Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[11] CSU Face Identification Evaluation System. `http://www.cs.colostate.edu/evalfacerec/index10.php`.

[12] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 381–394. ACM, 2013.

[13] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[14] Ali Kamali. Sharing aware scheduling on multicore systems. In *MSc Thesis, Simon Fraser Univ.*, 2010.

[15] Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. Aash: An asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 85–96, New York, NY, USA, 2010. ACM.

[16] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):pp. 54–66, 2008.

[17] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[18] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[19] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX ATC*, 2012.

[20] Richard P. LaRowe, Jr., Carla Schlatter Ellis, and Mark A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE TPDS*, 3:686–701, 1991.

[21] Tong Li, Dan Baumberger, David A Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Supercomputing, 2007. SC '07. Proceedings of*

the 2007 ACM/IEEE Conference on, pages 1–11, Nov 2007.

[22] Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *ISMM*, 2011.

[23] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *SYSTOR*, 2011.

[24] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *EuroSys*, 2010.

[25] Metis MapReduce Library. `http://pdos.csail.mit.edu/metis/`.

[26] PARSEC Benchmark Suite. `http://parsec.cs.princeton.edu/`.

[27] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Optimal task assignment in multithreaded processors: A statistical approach. *SIGARCH Comput. Archit. News*, 40(1):235–248, March 2012.

[28] Petar Radojković, Vladimir Čakarević, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Thread to strand binding of parallel network applications in massive multi-threaded systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 191–202, New York, NY, USA, 2010. ACM.

[29] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 139–152, New York, NY, USA, 2010. ACM.

[30] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys*, 2007.

[31] Lingjia Tang, J. Mars, Xiao Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing google's warehouse scale computers: The numa experience. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 188–197, Feb 2013.

[32] Toward better NUMA scheduling. Linux Weekly News, March 2012. `http://lwn.net/Articles/486858/`.

[33] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS*, 1996.

[34] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.

Latency-Tolerant Software Distributed Shared Memory

Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin

University of Washington
Department of Computer Science and Engineering

Abstract

We present Grappa, a modern take on software distributed shared memory (DSM) for in-memory data-intensive applications. Grappa enables users to program a cluster as if it were a single, large, non-uniform memory access (NUMA) machine. Performance scales up even for applications that have poor locality and input-dependent load distribution. Grappa addresses deficiencies of previous DSM systems by exploiting application parallelism, trading off latency for throughput. We evaluate Grappa with an in-memory MapReduce framework ($10\times$ faster than Spark [74]); a vertex-centric framework inspired by GraphLab ($1.33\times$ faster than native GraphLab [48]); and a relational query execution engine ($12.5\times$ faster than Shark [31]). All these frameworks required only 60-690 lines of Grappa code.

1 Introduction

Data-intensive applications (e.g., ad placement, social network analysis, PageRank, etc.) make up an important class of large-scale computations. Typical hardware computing infrastructures for these applications are a collection of multicore nodes connected via a high-bandwidth commodity network (a.k.a. a cluster). Scaling up performance requires careful partitioning of data and computation; i.e., programmers have to reason about data placement and parallelism explicitly, and for some applications, such as graph analytics, partitioning is difficult. This has led to a diverse ecosystem of frameworks—MapReduce [26], Dryad [43], and Spark [74] for data-parallel applications, GraphLab [48] for certain graph-based applications, Shark [31] for relational queries, etc. They ease development by specializing to algorithmic structure and dynamic behavior; however, applications that do not fit well into one particular model suffer in performance.

Software distributed shared memory (DSM) systems provide shared memory abstractions for clusters. Historically, these systems [15, 19, 45, 47] performed poorly, largely due to limited inter-node bandwidth, high inter-node latency, and the design decision of piggybacking on the virtual memory system for seamless global memory accesses. Past software DSM systems were largely inspired by symmetric multiprocessors (SMPs), attempting to scale that programming mindset to a cluster. However, applications were only suitable for them if they exhibited significant locality, limited sharing and coarse-grain synchronization—a poor fit for many modern data-analytics applications. Recently there has been a renewed interest in DSM research [27, 51], sparked by the widespread availability of high-bandwidth low-latency networks with remote memory access (RDMA) capability.

In this paper we describe Grappa, a software DSM system for commodity clusters designed for data-intensive applications. Grappa is inspired by the Tera MTA [10, 11], a custom hardware-based system. Like the MTA, instead of relying on *locality* to reduce the cost of memory accesses, Grappa depends on *parallelism* to keep processor resources busy and hide the high cost of inter-node communication. Grappa also adopts the shared-memory, fine-grained parallel programming mindset from the MTA. To support fine-grained messaging like the MTA, Grappa includes an overlay network that combines small messages together into larger physical network packets, thereby maximizing the available bisection bandwidth of commodity networks. This communication layer is built in user-space, utilizing modern programming language features to provide the global address space abstraction. Efficiencies come from supporting sharing at a finer granularity than a page, avoiding the page-fault trap overhead, and enabling compiler optimizations on global memory accesses.

The runtime system is implemented in C++ for a cluster of x86 machines with an InfiniBand interconnect, and consists of three main components: a global address space (§3.1), lightweight user-level tasking (§3.2), and an aggregating communication layer (§3.3). We demonstrate the generality and performance of Grappa as a common runtime by implementing three domain-specific platforms on top of it: a simple in-memory MapReduce framework; a vertex-centric API (i.e. like GraphLab); and a relational query processing engine. Comparing against GraphLab itself, we find that a simple, randomly partitioned graph representation on Grappa performs $2.5\times$ better than GraphLab's random partitioning and $1.33\times$ better than their best partitioning strategy, and scales comparably out to 128 cluster nodes. The query engine built on Grappa, on the other hand, performs $12.5\times$ faster than Shark on a standard benchmark suite. The flexibility and efficiency of the Grappa shared-memory programming model allows these frameworks to co-exist in the same application and to exploit application-specific optimizations that do not fit within any existing model.

The next section provides an overview of how data-intensive application frameworks can easily and effi-

ciently map to a shared-memory programming model. §3 describes the Grappa system. §4 presents a quantitive evaluation of the Grappa runtime. §5 describes related work, and §6 concludes.

2 Data-Intensive Application Frameworks

Analytics frameworks—such as MapReduce, graph processing and relational query execution—are typically implemented for distributed private memory systems (clusters) to achieve scale-out performance. While implementing these frameworks in a shared-memory system would be straightforward, this has generally been avoided because of scalability concerns. We argue that modern data-intensive applications have properties that can be exploited to make these frameworks run efficiently and scale well on *distributed* shared memory systems.

Figure 1 shows a minimal example of implementing a "word count"-like application in actual Grappa DSM code. The input array, `chars`, and output hash table, `cells`, are distributed over multiple nodes. A parallel loop over the input array runs on all nodes, hashing each key to its cell and incrementing the corresponding count atomically. The syntax and details will be discussed in later sections, but the important thing to note is that it looks similar to plain shared-memory code, yet spans multiple nodes and, as we will demonstrate in later sections, scales efficiently.

Here we describe how three data-intensive computing frameworks map to a DSM, followed by a discussion of the challenges and opportunities they provide for an efficient implementation:

MapReduce. Data parallel operations like map and reduce are simple to think of in terms of shared memory. Map is simply a parallel loop over the input (an array or other distributed data structure). It produces intermediate results into a hash table similar to that in Figure 1. Reduce is a parallel loop over all the keys in the hash table.

Vertex-centric. GraphLab/PowerGraph is an example of a vertex-centric execution model, designed for implementing machine-learning and graph-based applications [35, 48]. Its three-phase gather-apply-scatter (GAS) API for vertex programs enables several optimizations pertinent to natural graphs. Such graphs are difficult to partition well, so algorithms traversing them exhibit poor locality. Each phase can be implemented as a parallel loop over vertices, but fetching each vertex's neighbors results in many fine-grained data requests.

Relational query execution. Decision support, often in the form of relational queries, is an important domain of data-intensive workloads. All data is kept in hash tables stored in a DSM. Communication is a function of inserting into and looking up in hash tables. One parallel loop builds a hash table, followed by a second parallel loop that filters and probes the hash table, producing the results. These steps rely heavily on consistent, fine-grained updates to hash tables.

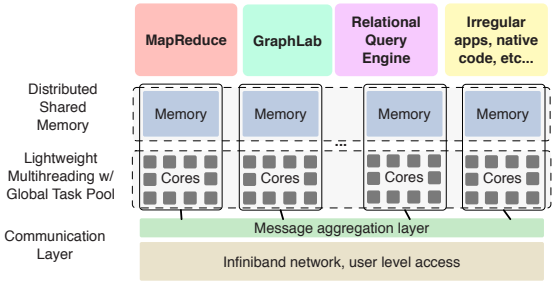


Figure 2: Grappa design overview

The key challenges in implementing these frameworks on a DSM are:

Small messages. Programs written to a shared memory model tend to access small pieces of data, which when executing on a DSM system lead to small inter-node messages. What were load or store operations become complex transactions involving small messages over the network. Conversely, programs written using a message passing library, such as MPI, expose this complexity to programmers, and hence encourage them to optimize it.

Poor locality. As previously mentioned, data-intensive applications often exhibit poor locality. For example, how much communication GraphLab's gather and scatter operations conduct is a function of the graph partition. Complex graphs frustrate even the most advanced partitioning schemes [35]. This leads to poor spatial locality. Moreover, which vertices are accessed varies from iteration to iteration. This leads to poor temporal locality.

Need for fine-grain synchronization. Typical data-parallel applications offer coarse-grained concurrency with infrequent synchronization—e.g., between phases of processing a large chunk of data. Conversely, graph-parallel applications exhibit fine-grain concurrency with frequent synchronization—e.g., when done processing work associated with a single vertex. Therefore, for a DSM solution to be general, it needs to support fine-grain synchronization efficiently.

Fortunately, data-intensive applications have properties that can be exploited to make DSMs efficient: their abundant data parallelism enables high degrees of concurrency; and their performance depends not on the *latency* of execution of any specific parallel task/thread, as it would in for example a web server, but rather on the aggregate execution time (i.e., *throughput*) of *all* tasks/threads. In the next section we explore how these application properties can be exploited to implement an efficient DSM.

3 Grappa Design

Figure 2 shows an overview of Grappa's DSM system. Before describing the Grappa system in detail, we describe its three main components:

```
// distributed input array
GlobalAddress<char> chars = load_input();

// distributed hash table:
using Cell = std::map<char,int>;
GlobalAddress<Cell> cells = global_alloc<Cell>(ncells);

forall(chars, nchars, [=](char& c) {
  // hash the char to determine destination
  size_t idx = hash(c) % ncells;
  delegate(&cells[idx], [=](Cell& cell)
  { // runs atomically
    if (cell.count(c) == 0) cell[c] = 1;
    else cell[c] += 1;
  });
});
```

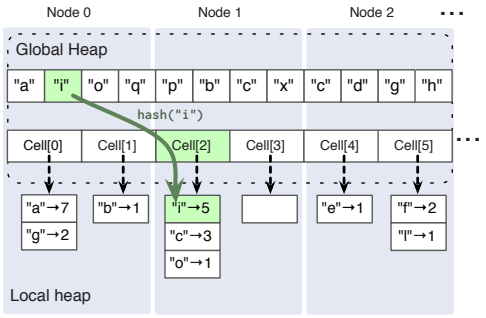


Figure 1: "Character count" with a simple hash table implemented using Grappa's distributed shared memory.

Distributed shared memory. The DSM system provides fine-grain access to data anywhere in the system. Every piece of global memory is owned by a particular core in the system. Access to data on remote nodes is provided by *delegate* operations that run on the owning core. Delegate operations may include normal memory operations such as *read* and *write* as well as synchronizing operations such as *fetch-and-add* [36]. Due to delegation, the memory model offered is similar to what underpins C/C++ [17, 44], so it is familiar to programmers.

Tasking system. The tasking system supports lightweight multithreading and global distributed work-stealing—tasks can be stolen from any node in the system, which provides automated load balancing. Concurrency is expressed through cooperatively-scheduled user-level threads. Threads that perform long-latency operations (i.e., remote memory access) automatically suspend while the operation is executing and wake up when the operation completes.

Communication layer. The main goal of our communication layer is to aggregate small messages into large ones. This process is invisible to the application programmer. Its interface is based on active messages [69]. Since aggregation and deaggregation of messages needs to be very efficient, we perform the process in parallel and carefully use lock-free synchronization operations. For portability, we use MPI [50] as the underlying messaging library as well as for process setup and tear down.

3.1 Distributed Shared Memory

Below we describe how Grappa implements a shared global address space and the consistency model it offers.

3.1.1 Addressing Modes

Local memory addressing. Applications written for Grappa may address memory in two ways: locally and globally. Local memory is local to a single core within a node in the system. Accesses occur through conventional pointers. Applications use local accesses for a number of things in Grappa: the stack associated with a task, accesses to global memory from the memory's home core, and accesses to debugging infrastructure local to each system node. Local pointers cannot access memory on other cores, and are valid only on their home core.

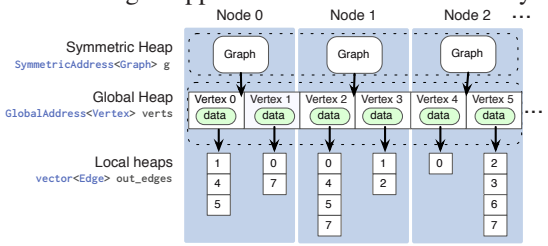


Figure 3: Using global addressing for graph layout.

Global memory addressing. Grappa allows any local data on a core's stacks or heap to be exported to the global address space to be made accessible to other cores across the system. This uses a traditional PGAS (partitioned global address space [30]) addressing model, where each address is a tuple of a rank in the job (or global process ID) and an address in that process.

Grappa also supports *symmetric* allocations, which allocates space for a copy (or proxy) of an object on every core in the system. The behavior is identical to performing a local allocation on all cores, but the local addresses of all the allocations are guaranteed to be identical. Symmetric objects are often treated as a proxy to a global object, holding local copies of constant data, or allowing operations to be transparently buffered. A separate publication [41] describes how this was used to implement Grappa's synchronized global data structures, including vector and hash map.

Putting it all together. Figure 3 shows an example of how global, local and symmetric heaps can all be used together for a simple graph data structure. In this example, vertices are allocated from the global heap, automatically distributing them across nodes. Symmetric pointers are used to access local objects which hold information about the graph, such as the base pointer to the vertices, from any core without communication. Finally, each vertex holds a vector of edges allocated from their core's local heap, which other cores can access by going through the vertex.

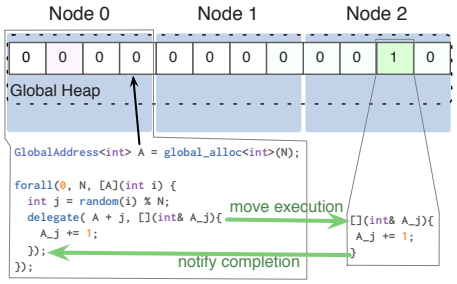


Figure 4: Grappa delegate example.

3.1.2 Delegate Operations

Access to Grappa's distributed shared memory is provided through *delegate* operations, which are short operations performed at the memory location's home node. When the data access pattern has low locality, it is more efficient to modify the data on its home core rather than bringing a copy to the requesting core and returning a modified version. Delegate operations [49, 53] provide this capability. While delegates can trivially implement *read*/*write* operations to global memory, they can also implement more complex *read-modify-write* and synchronization operations (e.g., *fetch-and-add*, mutex acquire, queue insert). Figure 4 shows an example.

Delegate operations must be expressed explicitly to the Grappa runtime, a change from the traditional DSM model. In practice, even programmers using implicit DSMs had to work to express and exploit locality to obtain performance. In other work we have developed a compiler [40] that automatically identifies and extracts productive delegate operations from ordinary code.

A delegate operation can execute arbitrary code provided it does not lead to a context switch. This guarantees atomicity for all delegate operations. To avoid context switches, a delegate must only touch memory owned by a single core. A delegate is *always* executed at the home core of the data addresses it touches. Given these restrictions, we can ensure that delegate operations for the same address from multiple requesters are always serialized through a single core in the system, providing atomicity with strong isolation. A side benefit is that atomic operations on data that are highly contended are faster. When programmers want to operate on data structures spread across multiple nodes, accesses must be expressed as multiple delegate operations along with appropriate synchronization operations.

3.1.3 Memory Consistency Model

Accessing global memory though delegate operations allows us to provide a familiar memory model. All synchronization is done via delegate operations. Since delegate operations execute on the home core of their operand in some serial order and only touch data owned by that single core, they are guaranteed to be globally linearizable [38], with their updates visible to all cores across the system in the same order. In addition, only one synchronous delegate will be in flight at a time from a particular task, i.e., synchronization operations from a particular task are not subject to reordering. Moreover, once one core is able to see an update from a synchronous delegate, all other cores are too. Consequently, all synchronization operations execute in program order and are made visible in the same order to all cores in the system. These properties are sufficient to guarantee a memory model that offers sequential consistency for data-race-free programs [5], which is what underpins C/C++ [17, 44]. The synchronous property of delegates provides a clean model but is restrictive: we discuss asynchronous operations within the next section.

3.2 Tasking System

Each hardware core has a single operating system thread pinned to it; all Grappa code runs in these threads. The basic unit of execution in Grappa is a *task*. When a task is ready to execute, it is mapped to a user-level *worker* thread that is scheduled within an operating system thread; we refer to these as workers to avoid confusion. Scheduling between tasks is carried out entirely in user-mode without operating system intervention.

Tasks. Tasks are specified by a closure (also referred to as a "functor" or "function object" in C++) that holds both code to execute and initial state. The closure can be specified with a function pointer and explicit arguments, a C++ struct that overloads the parentheses operator, or a C++11 lambda construct. These objects, typically small ($\sim$ 32 bytes), hold read-only values such as an iteration index and pointers to common data or synchronization objects. Task closures can be serialized and transported around the system, and are eventually executed by a worker.

Workers. Workers execute application and system (e.g., communication) tasks. A worker is simply a collection of status bits and a stack, allocated at a particular core. When a task is ready to execute it is assigned to a worker, that executes the task closure on its own stack. Once a task is mapped to a worker it stays with that worker until it finishes.

Scheduling. During execution, a worker yields control of its core whenever performing a long-latency operation, allowing the processor to remain busy while waiting for the operation to complete. In addition, a programmer can direct scheduling explicitly. To minimize context-switch overhead, the Grappa scheduler operates entirely in user-space and does little more than store state of one worker and load that of another. When a task encounters a long-latency operation, its worker is suspended and subsequently woken when the operation completes.

Each core in a Grappa system has its own independent scheduler. The scheduler has a collection of active workers ready to execute called the *ready worker queue*. Each

scheduler also has three queues of tasks waiting to be assigned a worker. The first two run user tasks: a public queue of tasks that are not bound to a core yet, and a private queue of tasks already bound to the core where the data they touch is located. The third is a priority queue scheduled according to task-specific deadline constraints; this queue manages high priority system tasks, such as periodically servicing communication requests.

Context switching. Grappa context switches between workers non-preemptively. As with other cooperative multithreading systems, we treat context switches as function calls, saving and restoring only the callee-saved state as specified in the x86-64 ABI [12] rather than the full register set required for a preemptive context switch. This requires 62 bytes of storage.

Grappa's scheduler is designed to support a very large number of concurrently-active workers—so large, in fact, that their combined context data will not fit in cache. In order to minimize unnecessary cache misses on context data, the scheduler explicitly manages the movement of context data into the cache. To accomplish this, we establish a pipeline of ready worker references in the scheduler. This pipeline consists of *ready-unscheduled*, *ready-scheduled*, and *ready-resident* stages. When context prefetching is on, the scheduler is only ever allowed to run workers that are *ready-resident*; all other workers are assumed to be out-of-cache. The examined part of the ready queue itself must also be in cache. In a FIFO schedule, the head of the queue will always be in cache due to its spatial locality. Other schedules are possible as long as the amount of data they need to examine to make a decision is independent of the total number of workers.

When a worker is signaled, its reference is marked *ready-unscheduled*. Every time the scheduler runs, one of its responsibilities is to pick a *ready-unscheduled* worker to transition to *ready-scheduled*: it issues a software prefetch to start moving the task toward L1. A worker needs its metadata (one cache line) and its private working set. Determining the exact working set might be difficult, but we find that approximating the working set with the top 2-3 cache lines of the stack is the best naive heuristic. The worker data is *ready-resident* when it arrives in cache. Since the arrival of a prefetched cache line is generally not part of the architecture, we must determine the latency from profiling.

At our standard operating point on our cluster ($\approx$1,000 workers), context switch time is on the order of 50 ns. As we add workers, the time increases slowly, but levels off: with 500,000 workers context switch time is around 75 ns. Without prefetching, context switching is limited by memory access latency—approximately 120 ns for 1,000 workers. Conversely, with prefetching on, context switching rate is limited by memory bandwidth—we determine this by calculating total data movement based on switch rate and cache lines per switch in a microbenchmark. As a reference point, for the same yield test using kernel-level Pthreads on a single core, the switch time is 450ns for a few threads and 800ns for 1000–32000 threads.

Expressing parallelism. The Grappa API supports spawning individual tasks, with optional data locality constraints. These tasks may run as full-fledged workers with a stack and the ability to block, or they may be *asynchronous delegates*, which like delegate operations execute non-blocking regions of code atomically on a single core's memory. Asynchronous delegates are treated as task spawns in the memory model.

For better programmability, tasks are automatically generated from parallel loop constructs, as in Figure 1. Grappa's parallel loops spawn tasks using a *recursive decomposition* of iterations, similar to Cilk's cilk_for construct [16], and TBB's `parallel_for` [59]. This generates a logarithmically-deep tree of tasks, stopping to execute the loop body when the number of iterations is below a user-definable threshold.

Grappa loops can iterate over an index space or over a region of shared memory. In the former case, tasks are spawned with no locality constraints, and may be stolen by any core in the system. In the latter case, tasks are bound to the home core of the piece of memory on which they are operating so that the loop body may optimize for this locality, if available. The local region of memory is still recursively decomposed so that if a particular loop iteration's task blocks, other iterations may run concurrently on the core.

3.3 Communication Support

Grappa's communication layer has two components: a user-level messaging interface based on active messages, and a network-level transport layer that supports request aggregation for better communication bandwidth.

Active message interface. At the upper (user-level) layer, Grappa implements asynchronous active messages [69]. Our active messages are simply a C++11 lambda or other closure. We take advantage of the fact that our homogeneous cluster hardware runs the same binary in every process: each message consists of a template-generated deserializer pointer, a byte-for-byte copy of the closure, and an optional data payload.

Message aggregation. Since communication is very frequent in Grappa, aggregating and sending messages efficiently is very important. To achieve that, Grappa makes careful use of caches, prefetching, and lock-free synchronization operations.

Figure 5 shows the aggregation process. Cores keep their own outgoing message lists, with as many entries as the number of system cores in a Grappa system. These lists are accessible to all cores in a Grappa node to allow cores to peek at each other's message lists. When a task

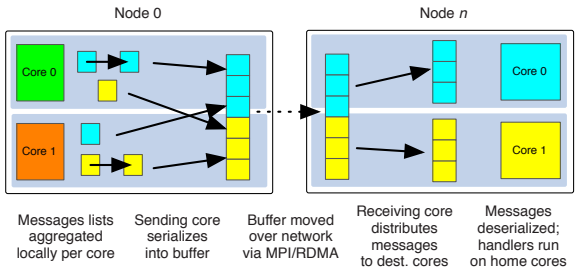


Figure 5: Message aggregation process.

sends a message, it allocates a buffer from a pool, determines the destination system node, writes the message contents into the buffer, and links the buffer into the corresponding outgoing list. These buffers are referenced only twice for each message sent: once when the message is created, and (much later) when the message is serialized for transmission. The pool allocator prefetches the buffers with the non-temporal flag to minimize cache pollution.

Each processing core in a given system node is responsible for aggregating and sending the resulting messages from all cores on that node to a set of destination nodes. Cores periodically execute a system task that examines the outgoing message lists for each destination node for which the core is responsible; if the list is long enough or a message has waited past a time-out period, all messages to a given destination system node from that source system node are sent by copying them to a buffer visible to the network card. Actual message transmission can be done purely in user-mode using MPI, which in turn uses RDMA.

The final message assembly process involves manipulating several shared data-structures (the message lists), so it uses CAS (compare-and-swap) operations to avoid high synchronization costs. This traversal requires careful prefetching because most of the outbound messages are *not* in the processor cache at this time (recall that a core can be aggregating messages originating from other cores in the same node). Note that we use a per-core array of message lists that is only periodically modified across processor cores, having experimentally determined that this approach is faster (sometimes significantly) than a global per-system node array of message lists.

Once the remote system node has received the message buffer, a management task is spawned to manage the unpacking process. The management task spawns a task on each core at the receiving system to simultaneously unpack messages destined for that core. Upon completion, these unpacking tasks synchronize with the management task. Once all cores have processed the message buffer, the management task sends a reply to the sending system node indicating the successful delivery of the messages.

3.3.1 Why not just use native RDMA support?

Given the increasing availability and decreasing cost of RDMA-enabled network hardware, it would seem logical to use this hardware to implement Grappa's DSM. Figure 6 shows the performance difference between native RDMA atomic increments and Grappa atomic increments using the GUPS cluster-wide random access benchmark using the cluster described in §4. The cluster has Mellanox ConnectX-2 40Gb InfiniBand cards connected through a QLogic switch with no oversubscription. The RDMA setting of the experiment used the network card's native atomic fetch-and-increment operation, and issued increments to the card in batches of 512. The Grappa setting issued delegate increments in a parallel for loop. Both settings perform increments to random locations in a 32 GB array of 64-bit integers distributed across the cluster. Figure 6(left) shows how aggregation allows Grappa to exceed the performance of the card by $25\times$ at 128 nodes. We measured the effective bisection bandwidth of the cluster as described in [39]: for GUPS, performance is limited by memory bandwidth during aggregation, and uses $\sim 40\%$ of available bisection bandwidth.

Figure 6(right) illustrates why using RDMA directly is not sufficient. The data also shows that MPI over InfiniBand has negligible overhead. Our cluster's cards are unable to push small messages at line rate into the network: we measured the peak RDMA performance of our cluster's cards to be 3.2 million 8-byte writes per second, when the wire-rate limit is over 76 million [42]. We believe this limitation is primarily due to the latency of the multiple PCI Express round trips necessary to issue one operation; a similar problem was studied in [34]. Furthermore, RDMA network cards have severely limited support for synchronization with the CPU [27, 51]. Finally, framing overheads can be large: InfiniBand 8-byte RDMA writes moves 50 bytes on the wire; Ethernet-based RDMA using RoCE moves 98 bytes. Work is ongoing to improve network card small message performance [1, 4, 28, 34, 55, 57, 61, 68]: even if native small message performance improves in future hardware, our aggregation support will still be useful to minimize cache line movement, PCI Express round trips, and other memory hierarchy limitations.

3.4 Fault tolerance discussion

A number of recent "big data" workload studies [22, 60, 62] suggest that over 90 percent of current analytics jobs require less than one terabyte of input data and run for less than one hour. We designed Grappa to support this size of workload on medium-scale clusters, with tens to hundreds of nodes and a few terabytes of main memory. At this scale, the extreme fault tolerance found in systems like Hadoop is largely wasted — e.g., assuming a per-machine MTBF of 1 year, we would estimate the MTBF of our 128-node cluster to be 2.85 days.

We could add checkpoint/restart functionality to Grappa, either natively or using a standard HPC library

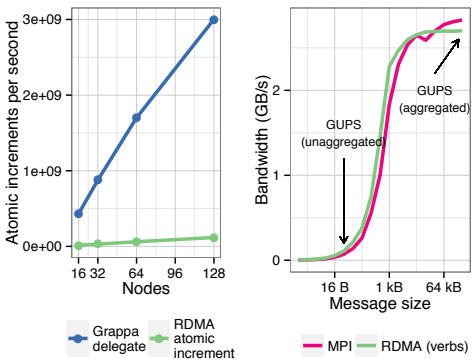


Figure 6: On the left, random updates to a billion-integer distributed array with the GUPS benchmark. On the right, ping-pong bandwidth measured between two nodes.

[29]. Writing a checkpoint would take on the order of minutes; for instance, our cluster can write all 8 TB of main memory to its parallel filesystem in approximately 10 minutes. It is important to balance the cost of taking a checkpoint with the work lost since the last checkpoint in the event of a failure. We can approximate the optimum checkpoint interval using [73]; assuming a checkpoint time of 10 minutes and a per-machine MTBF of 1 year, we should take checkpoints every 4.7 hours. These estimates are similar to what Low et al. measured for Graphlab's checkpoint mechanism in [48]. In this regime, it is likely cheaper to restart a failed job than it is to pay the overhead of taking checkpoints and recovering from a failure.

Given these estimates, we chose not to implement fault tolerance in this work. Adding more sophisticated fault tolerance to Grappa for clusters with thousands of nodes is an interesting area of future work.

4 Evaluation

We implemented Grappa in C++ for the Linux operating system. The core runtime system system is 17K lines of code. We ran experiments on a cluster of AMD Interlagos processors with 128 nodes. Nodes have 32 cores operating at 2.1GHz, spread across two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards. Nodes are connected via a QLogic InfiniBand switch with no oversubscription. We used a stock OS kernel and device drivers. The experiments were run in a machine without administrator access or special privileges. GraphLab and Spark communicated using IP-over-InfiniBand in Connected mode.

4.1 Vertex-centric Programs on Grappa

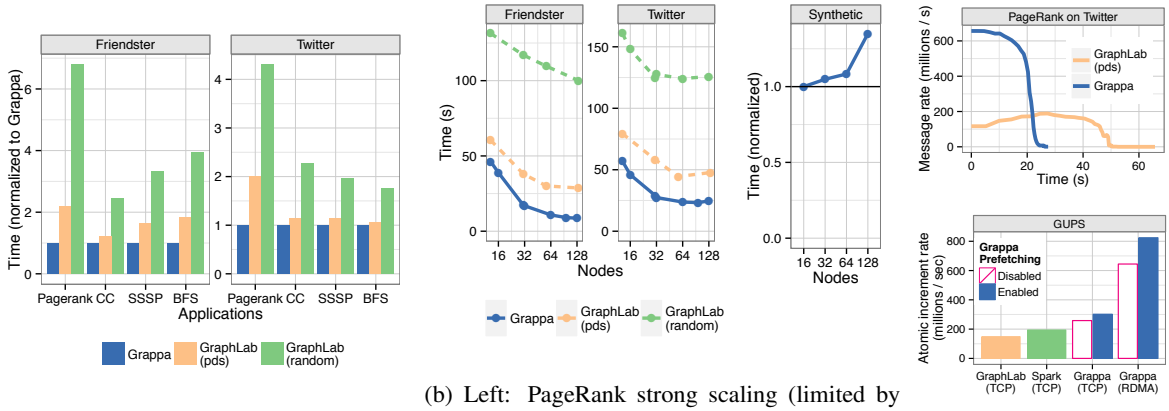
We implemented a vertex-centric programming framework in Grappa with most of the same core functionality as GraphLab [35, 48] using the graph data structure provided by the Grappa library (Figure 3). Unlike GraphLab we do not focus on intelligent partitioning, instead choosing a simple random placement of vertices to cores. Edges are stored co-located on the same core with vertex data. Using this graph representation, we implement a subset of GraphLab's synchronous engine, including the delta caching optimization, in ∼60 lines of Grappa code. Parallel iterators are defined over the vertex array and over each vertex's outgoing edge list. Given our graph structure, we can efficiently support gather on incoming edges and scatter on outgoing edges. Users of our Vertex-centric Grappa framework specify the gather, apply, and scatter operations in a "vertex program" structure. Vertex program state is represented as additional data attached to each vertex. The synchronous engine consists of several parallel `forall` loops executing the gather, apply, and scatter phases within an outer "superstep" loop until all vertices are inactive.

We implemented three graph analytics applications from GraphBench [3] using vertex program definitions equivalent to GraphLab's: PageRank, Single Source Shortest Path (SSSP), and Connected Components (CC). In addition, we implemented a simple Breadth-first search (BFS) application in the spirit of the Graph500 benchmark [37], which finds a "parent" for each vertex with a given source. The implementation in the GraphLab API is similar to the SSSP vertex program.

4.1.1 Performance

To evaluate Grappa's Vertex-centric framework implementation, we ran each application on the Twitter follower graph [46] (41 M vertices, 1 B directed edges) and the Friendster social network [72] (65 M vertices, 1.8 B undirected edges). For each we run to convergence—for PageRank we use GraphLab's default threshold criteria—resulting in the same number of iterations for each. Additionally, for PageRank we ran with delta caching enabled, as it proved to perform better. For Grappa we use the no-replication graph structure with random vertex placement; for GraphLab, we show results for random partitioning and the current best partitioning strategy: "PDS" which computes the "perfect difference set", but can only be run with $p^2 + p + 1$ (where p is prime) nodes. Most of the comparisons are done at 31 nodes for this reason.

Figure 7a depicts performance results at 31 nodes, normalized to Grappa's execution time. We can see that Grappa is faster than random partitioning on all the benchmarks (on average 2.57×), and 1.33× faster than the best partitioning, despite not replicating the graph at all. Both implementations of PageRank issue application-level requests on the order of 32 bytes (mostly communicating updated rank values). However, since these would perform terribly on the network, both systems aggregate updates into larger wire-level messages. Grappa's performance exceeds that of GraphLab primarily because it does this faster.



(a) Application performance (31 nodes)

(b) Left: PageRank strong scaling (limited by dataset size)
Right: weak scaling on synthetic graphs.

(c) Message rates (31 nodes).

Figure 7: *Performance characterization of Grappa's Vertex-centric framework* (a) shows time to converge (same number of iterations) normalized to Grappa, on the Twitter and Friendster datasets. (b) shows scaling results for PageRank out to 128 nodes—Friendster and Twitter measure *strong* scaling, and *weak* scaling is measured on synthetic power-law graphs scaled proportionally with nodes. (c) On top, cluster-wide message rates (average per iteration) while computing PageRank. On the bottom, GUPS message rates for GraphLab, Spark, and Grappa on 31 nodes. Grappa is shown using both TCP-based and RDMA-based configuration, with message prefetching on and off.

Figure 7c(bottom) explores this difference using the GUPS benchmark from §3.3.1. All systems send 32-byte updates to random nodes which then update a 64-bit word in memory: this experiment models only the communication of PageRank and not the activation of vertices, etc. For GraphLab and Spark, the messaging uses TCP-over-IPoIB and the aggregators make 64KB batches (GraphLab also uses MPI, but for job startup only). At 31 nodes, GraphLab's aggregator achieves 0.14 GUPS, while Grappa achieves 0.82 GUPS. Grappa's use of RDMA accounts for about half of that difference; when Grappa uses MPI-over-TCP-over-IPoIB it achieves 0.30 GUPS. The other half comes from Grappa's prefetching, more efficient serialization, and other messaging design decisions. The Spark result is an upper bound obtained by writing directly to Spark's `java.nio`-based messaging API rather than Spark's user-level API.

During the PageRank computation, Grappa's unsophisticated graph representation sends 2× as many messages as GraphLab's replicated representation. However, as can be seen in Figure 7c(top), Grappa sends these messages at up to 4× the rate of GraphLab over the bulk of its execution. At the end of the execution when the number of active vertices is low, both systems' message rates drop, but Grappa's simpler graph representation allows it to execute these iterations faster as well. Overall, this leads to a 2× speedup.

Figure 8 demonstrates the connection between concurrency and aggregation over time while executing PageRank. We see that at each iteration, the number of concurrent tasks spikes as *scatter* delegates are performed on outgoing edges, which leads to a corresponding spike in bandwidth due to aggregating the many concurrent messages. At these points, Grappa achieves roughly 1.1 GB/s per node, which is 47% of peak bisection bandwidth for large packets discussed in §3.3.1, or 61% of the bandwidth for 80 kB messages, the average aggregated size. This discrepancy is due to not being able to aggregate packets as fast as the network can send them, but is still significantly better than unaggregated bandwidth.

Figure 7b(left) shows strong scaling results on both datasets. As we can see, scaling is poor beyond 32 nodes for both platforms, due to the relatively small size of the graphs—there is not enough parallelism for either system to scale on this hardware. To explore how Grappa fares on larger graphs, we show results of a weak scaling experiment in Figure 7b(right). This experiment runs PageRank on synthetic graphs generated using Graph500's Kronecker generator, scaling the graph size with the number of nodes, from 200M vertices, 4B edges, up to 2.1B vertices, 34B edges. Runtime is normalized to show distance from ideal scaling (horizontal line), showing that scaling deteriorates less than 30% at 128 nodes.

4.2 Relational queries on Grappa

We used Grappa to build a distributed backend to Raco, a relational algebra compiler and optimization framework [58]. Raco supports a variety of relational query language frontends, including SQL, Datalog, and an imperative language, MyriaL. It includes an extensible relational algebra optimizer and various intermediate query plan representations.

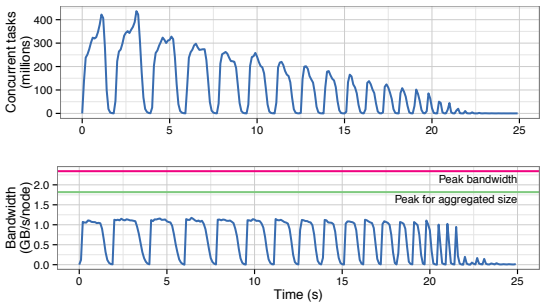


Figure 8: Grappa PageRank execution over time on 32 nodes. The top shows the total number of concurrent tasks (including delegate operations), over the 85 iterations, peaks diminishing as fewer vertices are being updated. The bottom shows message bandwidth per node, which correlates directly with the concurrency at each time step, compared against the peak bandwidth, and the bandwidth for the given message size.

We compare performance of our system to that of Shark, a fast implementation of Hive (SQL-like), built upon Spark. We chose this comparison point because Shark is optimized for in-memory execution and performs competitively with parallel databases [71].

Our particular approach for the Grappa backend to Raco is source-to-source translation. We generate `forall`s for each pipeline in the physical query plan. We extend the code generation approach for *serial* code in [54] to generating parallel shared memory code. The generated code is sent through a normal C++11 compiler.

All data structures used in query execution (e.g. hash tables for joins) are globally distributed and shared. While this a departure from the shared-nothing architecture of nearly all parallel databases, the locality-oriented execution model of Grappa makes the execution of the query virtually identical to that of traditional designs. We expect (and later demonstrate) that Grappa will excel at hash joins, given that it achieves high throughput on random access.

Implementing the parallel Grappa code generation was a relatively simple extension of the generator for serial C++ code that we use for testing Raco. It required less than 90 lines of template C++/Grappa code and 600 lines of support and data structure C++/Grappa code to implement conjunctive queries, including two join implementations.

4.2.1 Performance

We focus on workloads that can be processed in memory, since storage is out of scope for this work. For Grappa, we scan all tables into distributed arrays of rows in memory, then time the query processing. To ensure all timed processing in Shark is done in memory, we use the methodology that Shark's developers use for benchmarking [2]. In particular, all input tables are cached in memory and the output is materialized to an in-memory table. The number of reducer tasks for shuffles was set to 3 per Spark worker, which balances overhead and load balance. Each worker JVM was assigned 52GB of memory.

We ran conjunctive queries from SP^{2}Bench [63]. The queries in this benchmark involve several joins, which makes it interesting for evaluating parallel in-memory systems. We show results on 16 nodes (we found Shark failed to scale beyond 16 nodes on this data set) in Figure 9a. Grappa has a geometric mean speedup of $12.5\times$ over Shark. The benchmarks vary in performance due to differences in magnitude of communication and output size.

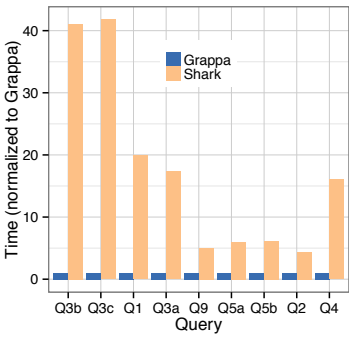
There are many differences between the two runtime systems (e.g. messaging layers, JVM and native) and the query processing approach (e.g. iterators vs compiled code), making it challenging to clearly understand the source of the performance difference between the two systems. To do so, we computed a detailed breakdown (Figure 9b) of the execution of Q2. We took sample-based profiles of both systems and categorized CPU time into five components: *network* (low-level networking overheads, such as MPI and TCP/IP messaging), *serialization* (aggregation in Grappa, Java object serialization in Shark), *iteration* (loop decomposition and iterator overheads), *application* (actual user-level query directives), and *other* (remaining runtime overheads for each system).

Overall, we find that the systems spend nearly the same amount of CPU time in application computation, and that more than half of Grappa's performance advantage comes from efficient message aggregation and a more efficient network stack. An additional benefit comes from iterating via Grappa's compiled parallel for-loops compared to Shark's dynamic iterators. Finally, both systems have other, unique overheads: Grappa's scheduling time is higher than Shark due to frequent context switches, whereas Shark spends time dynamically checking the types of data values.

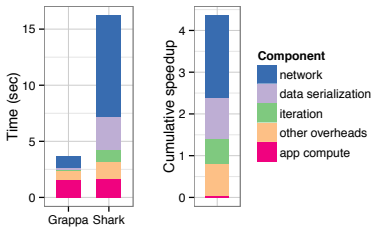
Shark's execution of these queries appears to place bursty demands on the network, and is sensitive to network bandwidth. On query Q2, Shark achieves the same peak bandwidth as GUPS (Figure 7c) sustains (200MB/s/node), but its sustained bandwidth is just over half this amount (116 MB/s/node).

4.3 Iterative MapReduce on Grappa

We experiment with data parallel workloads by implementing an in-memory MapReduce API in 152 lines of Grappa code. The implementation involves a `forall` over inputs followed by a `forall` over key groups. In the all-to-all communication, mappers push to reducers. As with other MapReduce implementations, a combiner function can be specified to reduce communication. In this case, the mappers materialize results into a local hash table, us-



(a) The SP^{2}Bench benchmark on 16 nodes. Query Q4 is a large workload so it was run with 64 nodes.



(b) Performance breakdown of speedup of Grappa over Shark on Q2.

Figure 9: Relational query performance.

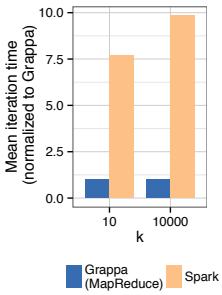


Figure 10: Data parallel experiments using k-means on a 8.9GB Seaflow dataset with 64 nodes.

ing Grappa's partition-awareness. The global-view model of Grappa allows iterations to be implemented by the application programmer with a `while` loop.

4.3.1 Performance

We pick k-means clustering as a test workload; it exercises all-to-all communication and iteration. To provide a reference point, we compare the performance to the `SparkKMeans` implementation for Spark. Both versions use the same algorithm: map the points, reduce the cluster means, and broadcast local means. The Spark code caches the input points in memory and does not persist partitions. Currently, our implementation of MapReduce is not fault-tolerant. To ensure the comparison is fair, we made sure Spark did not use fault-tolerance features: we used `MEMORY_ONLY` storage level for RDDs, which does not replicate an RDD or persist it to disk and verified during the runs that no partitions were recomputed due to failures. We run k-means on a dataset from Seaflow [66], where each instance is a flow cytometry sample of seawater containing characteristics of phytoplankton cells. The dataset is 8.9GB and contains 123M instances. The clustering task is to identify species of phytoplankton so the populations may be counted.

The results are shown in Figure 10 for $K = 10$ and $K = 10000$. We find Grappa-MapReduce to be nearly an order of magnitude faster than the comparable Spark implementation. Absolute runtime for Grappa-MapReduce is 0.13s per iteration for $K = 10$ and 17.3s per iteration for $K = 10000$, compared to 1s and 170s respectively for Spark.

We examined profiles to understand this difference. We see similar results as with Shark: the bulk of the difference comes from the networking layer and from data serialization. As K grows, this problem *should be* compute-bound: most execution time is spent assigning points to clusters in the map step. At large K, Grappa-MapReduce is clearly compute-bound, but Spark spends only 50% of its time on compute; the rest is in network code in the reduce step. Grappa's efficient small message support and support for overlapping communication and computation help it perform well here.

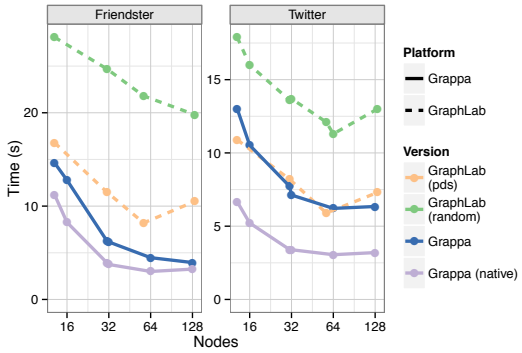


Figure 11: Scaling BFS out to 128 nodes. In addition to Grappa's GraphLab engine, we also show a custom algorithm for BFS implemented natively which employs Beamer's bottom-up optimization to achieve even better performance.

4.4 Writing directly to Grappa

Not all problems fit perfectly into current restricted programming models—for many, a better solution can be found by breaking these restrictions. An advantage of building specialized systems on top of a flexible, high-performance platform is that it makes it easier to implement new optimizations into domain-specific models, or implement a new algorithm from scratch natively. For example, for BFS, Beamer's direction-optimizing algorithm has been shown to greatly improve performance on the Graph500 benchmark by traversing the graph "bottom-up" in order to visit a subset of the edges [13]. This cannot be written in a pure Vertex-centric framework like GraphLab. We implemented the Beamer's BFS algorithm directly on the existing graph data structure in 70 lines of code. Performance results in Figure 11 show that this algorithm's

performance is nearly a factor of 2 better than the pure Vertex-centric abstraction can achieve.

5 Related Work

Multithreading. Hardware-based implementations of multithreading to tolerate latency include the Denelcor HEP [65], Tera MTA [11], Cray XMT [33], Simultaneous multithreading [67], MIT Alewife [6], Cyclops [9], and GPUs [32]. Hardware multithreading often pays with lower single-threaded performance that may limit appeal in the mainstream market. As a software implementation of multithreading for mainstream general-purpose processors, Grappa provides the benefits of latency tolerance only when warranted, leaving single-threaded performance intact.

Grappa's closest software-based multithreading ancestor is the Threaded Abstract Machine (TAM) [24]. TAM is a software runtime system designed for prototyping dataflow execution models on distributed memory supercomputers. Like Grappa, TAM supports inter-node communication, management of the memory hierarchy, and lightweight asynchronous scheduling of tasks to processors, all in support of computational throughput despite the high latency of communications. A notable conclusion [25] was that threading for latency tolerance was fundamentally limited because the latency of the top-level store (e.g. L1 cache) is in direct competition with the number of contexts that can fit in it. However, we find prefetching is effective at hiding DRAM latency in context switching. Indeed, a key difference between Grappa's support for lightweight threads and that of other user level threading packages, such as QThreads [70], TBB [59], Cilk [16] and Capriccio [14] is Grappa's context prefetching. Grappa's prefetching could likely improve from compiler analyses inspired by those of Capriccio for reducing memory usage.

Software distributed shared memory. Much of the innovation in DSM over the past 30 years has focused on reducing the synchronization costs of updates. The first DSM systems, including IVY [47], used frequent invalidations to provide sequential consistency, inducing high communication costs for write-heavy workloads. Later systems relaxed the consistency model to reduce communication demands; some systems further mitigated performance degradation due to false sharing by adopting multiple writer protocols that delay integration of concurrent writes made to the same page. The Munin [15, 19] and TreadMarks [45] systems exploited both of these ideas, but still incurred some coherence overhead. Munin and Blizzard [64] allowed the tracking of ownership with variable granularity to reduce the cost due to false sharing. Grappa follows the lead of TreadMarks and provides DSM entirely at user-level through a library and runtime. FaRM [27] offers lower latency and higher throughput updates to DSM than TCP/IP via lock free and transactional access protocols exploiting RDMA, but remote access throughput is still limited to the RDMA operation rate which is typically an order of magnitude less than the per node network bandwidth.

Partitioned Global Address Space languages. The high-performance computing community has largely discarded the coherent distributed shared memory approach in favor of the Partitioned Global Address Space (PGAS) model. Examples include Split-C [23], Chapel [20], X10 [21], Co-array Fortran [56] and UPC [30]. What is most different between general DSM systems and PGAS ones is that remote data accesses are explicit, thereby encouraging developers to use them judiciously. Grappa follows this approach, implementing a PGAS system at the language level, thereby facilitating compiler and programmer optimizations.

Distributed data-intensive processing frameworks. There are many other data-parallel frameworks like Hadoop, Haloop [18], and Dryad [43]. These are designed to make parallel programming on distributed systems easier; they meet this goal by targeting data-parallel programs. There have also been recent efforts to build parameter servers for distributed machine learning algorithms using asynchronous communication and distributed key-value storage built from RPCs [7, 8]. The incremental data-parallel system Naiad [52] achieves both high-throughput for batch workloads and low-latency for incremental updates. Most of these designs eschew DSM as an application programming model for performance reasons.

6 Conclusions

Our work builds on the premise that writing data-intensive applications and frameworks in a shared memory environment is simpler than developing custom infrastructure from scratch. To that end, Grappa is inspired not by SMP systems, but by novel supercomputer hardware – the Cray MTA and XMT line of machines. This work borrows the core insight of those hardware systems and builds it into a software runtime tuned to extract performance from commodity processors, memory systems and networks. Based on this premise, we show that a DSM system can be efficient for this application space by judiciously exploiting the key application characteristics of concurrency and latency tolerance. Our data demonstrates that frameworks such as MapReduce, vertex-centric computation, and query execution are *easy to build* and *efficient*. Our MapReduce and query execution implementations are an order of magnitude faster than the custom frameworks for each. Our vertex-centric GraphLab-inspired API is $1.33\times$ faster than GraphLab itself, without the need for complex graph partitioning schemes.

References

[1] Intel Data Plane Development Kit. http://goo.gl/AOvjss, 2013.

[2] Big data benchmark. https://amplab.cs.berkeley.edu/benchmark, Feb 2014.

[3] GraphBench. http://graphbench.org/, 2014.

[4] Snabb Switch project. https://github.com/SnabbCo/snabbswitch, May 2014.

[5] S. V. Adve and M. D. Hill. Weak ordering – A new definition. In *ISCA-17*, 1990.

[6] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[7] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM.

[8] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 37–48, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee.

[9] G. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *SIGARCH Computer Architecture News*, 31:26–38, March 2003.

[10] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 188–197, New York, NY, USA, 1992. ACM.

[11] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[12] AMD64 ABI. http://www.x86-64.org/documentation/abi-0.99.pdf, July 2012.

[13] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[14] R. V. Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281. ACM Press, 2003.

[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '90, pages 168–176, New York, NY, USA, 1990. ACM.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[17] H.-J. Boehm. A Less Formal Explanation of the Proposed C++ Concurrency Memory Model. C++ standards committee paper WG21/N2480 = J16/07-350, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html, December 2007.

[18] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[19] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[20] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[22] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.

[23] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM.

[24] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM – A compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[25] D. E. Culler, K. E. Schauser, and T. v. Eicken. Two fundamental limits on dataflow multiprocessing. In *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, PACT '93, pages 153–164, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX conference on Operating Systems Design and Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[27] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX.

[28] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. *SIGCOMM Comput. Commun. Rev.*, 24(4):2–13, Oct. 1994.

[29] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Labs Linux checkpoint/restart. Technical Report LBNL-54941, December 2002.

[30] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[31] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM.

[32] K. Fatahalian and M. Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, October 2008.

[33] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[34] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 333–346, Berkeley, CA, USA, 2013. USENIX Association.

[35] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[36] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer: Designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175 –189, Feb. 1983.

[37] Graph 500. `http://www.graph500.org/`, July 2012.

[38] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[39] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125, Sept 2008.

[40] B. Holt, P. Briggs, L. Ceze, and M. Oskin. Alembic. In *International Conference on PGAS Programming Models (PGAS)*, Oct 2013.

[41] B. Holt, J. Nelson, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Flat combining synchronized global data structures. In *International Conference on PGAS Programming Models (PGAS)*, Oct 2013.

[42] InfiniBand Trade Association. InfiniBand Architecture Specification, Version 1.2.1. 2007.

[43] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 ACM SIGOPS European Conference on Computer Systems*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[44] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language, C++ (Committee Draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`, 2008.

[45] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, pages 115–131, Berkeley, CA, USA, 1994. USENIX Association.

[46] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on the World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[47] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[48] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[49] R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA'11*, pages 885–902, 2011.

[50] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.

[51] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[52] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[53] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[54] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[55] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out numa. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18, New York, NY, USA, 2014. ACM.

[56] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[57] S. Peter and T. Anderson. Arrakis: A case for the end of the empire. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 26–26, Berkeley, CA, USA, 2013. USENIX Association.

[58] Raco: The relational algebra compiler. `https://github.com/uwescience/datalogcompiler`, April 2014.

[59] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[60] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, Aug. 2013.

[61] L. Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[62] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP 2012)*. ACM, April 2012.

[63] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.

[64] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS VI, pages 297–306, New York, NY, USA, 1994. ACM.

[65] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE 0298, Real-Time Signal Processing IV, 241*, volume 298, pages 241–248, July 1982.

[66] J. Swalwell, F. Ribalet, and E. Armbrust. Seaflow: A novel underway flow-cytometer for continuous observations of phytoplankton in the ocean. *Limnology & Oceanography Methods*, 9:466–477, 2011.

[67] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[68] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing (includes url). In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.

[69] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[70] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS*, pages 1–8. IEEE, 2008.

[71] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[72] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.

[73] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.

[74] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

NightWatch: Integrating Lightweight and Transparent Cache Pollution Control into Dynamic Memory Allocation Systems

Rentong Guo[†] Xiaofei Liao[†*] Hai Jin[†] Jianhui Yue[§] Guang Tan[‡]

[†] *Service Computing Technology and System Lab, Cluster and Grid Computing Lab,*
School of Computer Science and Technology, Huazhong University of Science and Technology, China

[§] *Auburn University, USA*

[‡] *SIAT, Chinese Academy of Sciences, China*

{*rtguo, xfliao, hjin*}*@hust.edu.cn jyue@auburn.edu guang.tan@siat.ac.cn*

Abstract

Cache pollution, by which weak-locality data unduly replaces strong-locality data, may notably degrade application performance in a shared-cache multicore machine. This paper presents NightWatch, a cache management subsystem that provides general, transparent and low-overhead pollution control to applications. NightWatch is based on the observation that data within the same memory chunk or chunks within the same allocation context often share similar locality property. NightWatch embodies this observation by online monitoring current cache locality to predict future behavior and restricting potential cache polluters proactively. We have integrated NightWatch into two popular allocators, *tcmalloc* and *ptmalloc2*. Experiments with SPEC CPU2006 show that NightWatch improves application performance by up to 45% (18% on average), with an average monitoring overhead of 0.57% (up to 3.02%).

1 Introduction

Modern multicore processors usually have a shared last level cache, where all cores place their data to improve cache utilization. This, however, creates a new challenge of cache management, due to cache pollution. One major problem is that data with weak locality may unduly evict other data with strong locality, if both are mapped into the same cache set [6, 12, 16, 18, 24, 30].

The major challenge to mitigate cache pollution is that the locality property of an application is implicitly determined by the runtime behavior. There has been much work [6, 16, 27, 28, 31] that has demonstrated the great potential of performance improvement via cache-aware memory allocation. However, they fall short in several aspects, such as requiring off-line analysis [3, 5, 16, 21, 23, 25, 26], special hardware support [7, 22, 27, 29, 31], or changing allocation interfaces [6, 28]. Hence, these techniques can hardly be used for general, unmodified applications.

This paper presents NightWatch, an online, transparent cache management subsystem for memory allocators. NightWatch dynamically characterizes the locality properties of allocated memory chunks, and provides hints to the memory allocator about the proper cache assignment for future allocation requests.

There are two main challenges to implement NightWatch efficiently. First, monitoring locality properties online is usually expensive and may easily cancel out the benefit from mitigated cache pollution. Second, it is hard to determine a priori whether a requested memory chunk will be a polluter before its actual use, due to the fact that the memory allocator has no knowledge about the application logic. NightWatch addresses the challenges by leveraging two new insights on the locality correlation among memory chunks[1]:

1. *Insight 1: Intra-chunk locality similarity*, where different pages in the same memory chunk tend to have similar locality properties;
2. *Insight 2: Inter-chunk locality similarity*, where different memory chunks in the same *allocation context*[2] tend to have similar locality properties.

Based on these insights, NigthWatch is built with mechanisms to monitor the access behavior of allocated memory chunks in a lightweight way, and to predict the behavior of new chunks with high reliability. Based on Insight 1, NightWatch infers the locality properties of a memory chunk by monitoring and sampling only a small part of each chunk. Based on Insight 2, NightWatch leverages the locality properties of previously allocated chunks to predict the locality properties of new chunks in the same allocation context.

We have integrated NightWatch into two popular memory allocators, *tcmalloc* [9] and *ptmalloc2* [10]. Specifically, NightWatch analyzes the historical locality

*Corresponding author

[1] A chunk is a memory block returned by malloc.

[2] The allocation context is identified as the call stack when the allocation function is called.

profile of allocated chunks and provides advices to the allocator on the proper cache assignment for the current allocation. In summary, we make three contributions:

- We demonstrate locality similarity within the same memory chunk, and between chunks in the same allocation context. These findings lay the foundation of practical online cache pollution control, which completely frees common programmers from cumbersome tasks of analyzing the program's cache demand.
- We present an open source implementation of NightWatch[3]. The cache management support is orthogonal to the traditional memory management techniques, and can be easily integrated into popular memory allocators [1, 8, 9, 10, 15].
- We have performed extensive evaluation of NightWatch with 27 programs from the SPEC CPU2006 benchmark suite. Compared with the popular allocator implementation of *tcmalloc*, NightWatch helps improve performance by up to 45%, with an average prediction accuracy of over 93%. NightWatch incurs very small extra overheads: the overhead is only 0.57% on average (up to 3.02%).

2 Motivation and Background

This section describes conventional ways of memory mapping, potential issues with cache pollution, and the concept of *restrictive mapping*.

2.1 Conventional Mapping

Since physical pages and cache sets are both physically indexed[4], the allocation of physical memory automatically determines the allocation of CPU caches. This relation is illustrated in Figure 1. A physical memory address is divided into two parts, i.e., page offset and physical page number. Several lower bits of the physical page number also serve as cache set index. The common bits divide cache sets and physical pages into different colors. Pages sharing a color are mapped to the same cache sets.

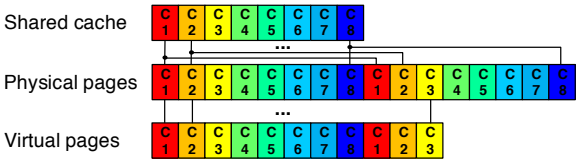
The memory allocator, as part of the runtime system, works at user level and is unaware of the mapping between cache and pages. The mapping is transparently handled by the *operating system* (OS). When the allocator acquires free memory, the OS returns memory with a unified mapping type. Conventional memory mapping techniques [20] attempt to maximize the number of cache sets assigned to consecutive virtual pages. As shown in Figure 2(a), physical pages in different colors are mapped to virtual pages in a round-robin manner. Such mapping allows even distribution of adjacent

[3]https://github.com/grtoverflow/PC-Malloc

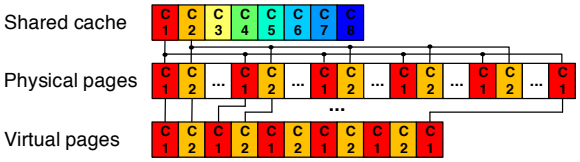
[4]CPU caches are commonly physically indexed [2, 11]. However, there are also some designs using virtual address as cache index. In this case, NightWatch's cache control mechanism will NOT work.



Figure 1: The relation between physical page number and cache set index



(a) Open mapping



(b) Restrictive mapping

Figure 2: Two types of memory mapping. Each bin in the shared cache represents a group of cache sets in the same color, and the bins in the following two rows are consecutive virtual pages and physical pages. The pages' labels distinguish the color.

data over the cache for the benefit of load balance between cache sets. The downside of this approach is that it allows weak-locality data to spread all over the cache, causing cache pollution to a maximum degree. Since there is no restriction on the pollution scope, we call this approach of memory mapping *open mapping*.

2.2 Issues with Conventional Mapping

The coexistence of both strong-locality and weak-locality data is very common. During execution, programs may have a large number of memory chunks with various locality properties.

Figure 3 shows the cache miss rate of the memory chunks of eight memory-intensive applications. It can be seen that the chunks differ in locality properties significantly. Take *dealII* as an example, 10% of its chunks have a miss rate below 10%, while 50% have a miss rate above 90%. In addition, the portion of the weak-locality chunks can be fairly large – two out of eight programs find a miss rate over 90% for more than half of their chunks. If not properly handled, the weak-locality chunks can overuse cache for little benefit, and leave little cache space to strong-locality chunks, thus degrading overall system performance.

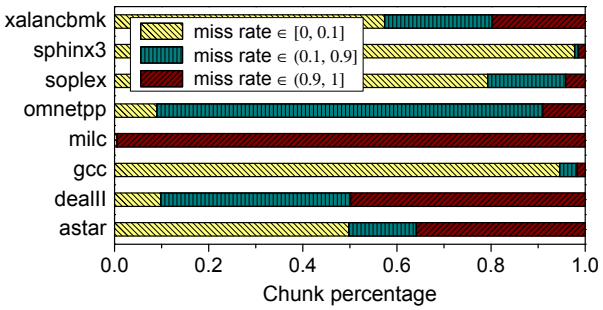


Figure 3: Chunk distributions of eight memory intensive programs. The cache size is 8MB.

2.3 Pollution Restrictive Mapping

A possible solution to the cache pollution issue would be to restrict the size of cache to which weak-locality chunks are mapped, and leaves more cache space to strong-locality ones. In this solution, page mapping is done in a restrictive way for polluters, and an open way for the rest of the chunks. Figure 2(b) depicts the principle of *restrictive mapping* for this solution. In the figure, the selection of physical pages are restricted in a limited *color region* (with two colors in this case) and mapped to consecutive virtual pages. Under such mapping, the weak-locality data are constrained and thus the pollution effect can be largely reduced.

While the idea of dual-mapping is conceptually straightforward, realizing it in an efficient way is non-trivial: the use of both open and restrictive mapping requires the memory allocator to be able to distinguish between polluter and normal chunks at runtime. As memory allocator is a core routine in most programs, such locality identification process needs to be performed in a lightweight manner. Otherwise, the monitoring overhead may easily nullify the benefit from improved cache locality.

3 Locality Similarity

Underlying our design are two observations of locality correlation between memory units. The first observation is concerned with locality similarity between pages within the same chunk, and the second on the locality similarity across different chunks within the same allocation context.

We use the SPEC CPU2006 benchmark suite to study the locality similarities. The benchmark consists of 27 programs[5]. We employ PIN [17] to collect the programs' memory allocation events and the full trace of data accesses. We then feed the data accesses to a cache simulator to track the cache miss rate of each memory chunk, and of each page within each chunk. The data

[5]Two programs, 434.zeusmp and 458.sjeng, are excluded, as they do not use dynamic memory allocation, and optimizing the cache allocation of stack and data segments are beyond the scope of this work.

```
img->mb_data
   = calloc(img->FrameSizeInMbs, sizeof(Macroblock));
......
/* encode a picture */
while (NumberOfCodedMBs < img->total_number_mb) {
   ......
   /* encode a macroblock in img->mb_data */
   encode_one_macroblock ();
   NumberOfCodedMBs++;
}
```

Figure 4: An example of intra-chunk access similarity

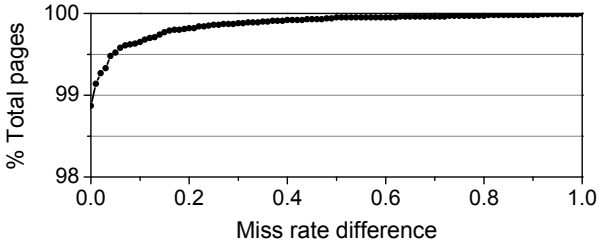


Figure 5: CDF of cache miss rate difference between pages and chunks

accesses are first filtered by a 256KB private cache, and then tracked in an 8MB set-associative shared cache. All the caches are configured with the LRU replacement policy, with 64B cache line size.

3.1 Intra-Chunk Locality Similarity

As discussed in Section 2, page is the basic unit for a memory allocator to manipulate a chunk's cache resource allocation. A chunk commonly consists of multiple pages, but managing cache allocation on a per-page basis is costly – that needs online monitoring for every page throughout the chunk's life cycle. Such an approach is necessary only if pages differ significantly in their locality properties.

Figure 4 shows an example of locality similarity within a single memory chunk, which is taken from *h264ref*, an implementation of the H.264/AVC (*Advanced Video Coding*) standard. The memory chunk, *img->mb_data*, is used to hold a whole frame of data during encoding. For an input video with 512x320 resolution, *img->mb_data* contains around 100 4KB pages. The frame is divided into macroblocks, each with 632B in size. In the main encoding iterations, each of the macroblocks is processed by *encode_one_macroblock ()*, with identical intra-frame and inter-frame compression algorithms. As a consequence, all the pages within *img->mb_data* will share similar locality properties. This implies that manipulating cache allocation for *img->mb_data* on a per page basis is unnecessary, since we can take the whole chunk as a basic cache allocation unit.

To verify the generality of the intra-chunk locality similarity, we examine all the chunks comprising more

```
for (img->number=0; img->number < input->no_frames;
      img->number++) {
   ......
   buf = malloc (xs * ys * symbol_size_in_bytes);
   /* read one frame */
   read(p_in, buf, bytes_y);
   /* convert file read buffer to source picture structure */
   buf2img(imgY_org_frm, buf, xs, ys, symbol_size_in_bytes);
   ......
   free (buf);
}
```

Figure 6: An example of inter-chunk access similarity

than one page in the 27 programs. We record the difference between each page's miss rate and its chunk's. The *cumulative distribution function* (CDF) of the difference is illustrated in Figure 5. The figure shows that most of the pages share a very similar miss rate with their chunks. Specifically, more than 98% of the pages have an identical miss rate as their chunks', and less than 0.5% of the pages have a miss rate difference from their chunks by 10%. This suggests that it is possible to manage cache allocation on a per chunk basis. More importantly, it is safe to reduce the number of monitored pages for lower overhead with little sacrifice of monitoring quality.

3.2 Inter-Chunk Locality Similarity

The locality monitor works only for allocated data. For a new memory request, the allocator has to perform cache mapping in a default way. If the monitoring results later on suggest that the default mapping does not fit the data's cache behavior, then a mapping switch, or remapping, is needed. This operation is prohibitively expensive: in the OS, data on the original pages needs to be copied to the new pages with proper physical indexing, followed by the update to the corresponding page table entries. In our experiments, for example, the time of remapping 1MB of data is as long as 1.8ms. Worse still, the effort may turn out not worthwhile, as many chunks are short-lived. For example, for the 27 programs we tested, over 90% of the chunks have a lifetime less than one second. After the monitoring and remapping phases, the cache allocation adjustment is too late to take effect, bringing very limited benefit compared with the cost.

To avoid cache remapping for a chunk, the allocator should make the initial mapping match the chunk's locality property, which calls for a reliable prediction of the chunk's access behavior. Fortunately, we find that the *allocation context*, defined as the call stack of an allocation request, provides important hints of the future behavior of to-be-allocated chunks, due to inter-chunk locality similarity.

Within an allocation context, a program commonly triggers memory allocation more than once. From Figure 7, we can see that for the 27 programs we tested, 99% of the chunk allocations fall in contexts that contain more

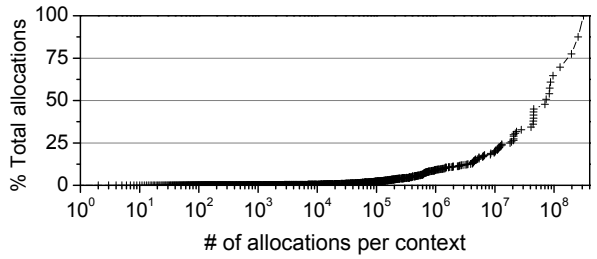


Figure 7: CDF of number of allocations

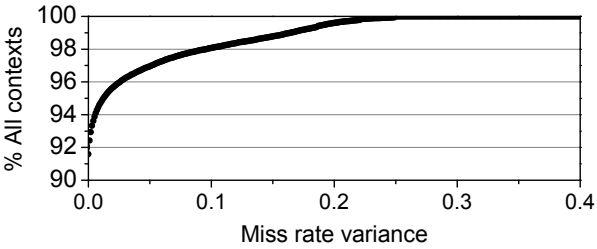


Figure 8: CDF of cache miss rate variance of the chunks sharing the same context

than 100 times of allocations. This indicates that most of the chunks are generated in "big" contexts, where they can find sufficient previously allocated chunks for locality prediction.

Figure 6 illustrates an example of chunk allocations in an allocation context, which is taken from *h264ref*. In the example, a group of memory chunks named *buf* are involved. Each of them lives in one encoding iteration: when a frame encoding iteration begins, *buf* is allocated to load one frame of data from the input file, then the data is converted to source picture format. After that, the memory chunk is freed back to the memory allocator. All the *buf* share the same allocation context, because the call stacks of *malloc()* in each encoding iteration are identical. Furthermore, these chunks also exhibit similar data access patterns – each of them serves one round of data installation and data conversion. As such, it is possible to use previously allocated chunks for locality prediction in later memory allocations.

In addition, the inter-chunk locality similarity also provides opportunities for reducing monitoring overhead. For contexts with good locality similarity, only a small part of the chunks need to be monitored to maintain a high prediction success rate.

To confirm the inter-chunk locality similarity in the same context, we calculate the variance of miss rate of chunks in each context. The CDF of miss rate variance is shown in Figure 8. Over 90% of the chunks share an identical miss rate with other chunks in the same context; less than 2% of the contexts have a miss rate variance greater than 0.1. As we will show later, this high level of locality similarity among the chunks leads to a high prediction success rate of 95.5% on average.

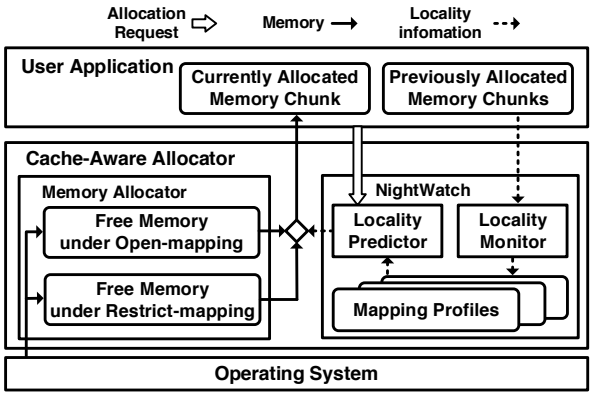


Figure 9: System overview of a NightWatch assisted allocator

4 Overview

NightWatch leverages the two observations of locality similarity, which inspire the design of efficient monitoring mechanisms, and help to make correct locality predictions. The system framework is shown in Figure 9.

The *locality monitor* collects locality information from previously allocated chunks. It periodically samples the references from the target chunks, and evaluates the chunk's locality property, which is sent to the *locality predictor*. Based on the historical locality information, the locality predictor determines the proper mapping for pending allocation requests. When a new request arrives, the predictor first checks its allocation context, and uses its predecessor chunks' locality profiles to predict the pending chunk's locality property. Then, the predictor notifies the memory allocator to perform the allocation.

The *memory allocator* is an extended allocator (i.e. extended from *tcmalloc*, *ptmalloc*, *ssmalloc,* etc). In most cases, a traditional allocator is unaware of cache pollution, and serves allocation requests with unique mapping (commonly the open mapping). To work with NightWatch, the allocator needs to be slightly modified to support both *open mapping* and *restrictive mapping*. The original allocator includes a set of data structures to maintain memory, for example, size classes used for containing small chunks [4, 9], and thread-local cache designed to manage per thread allocations [10, 15]. We extend the allocator to use two sets of these data structures, with each set maintaining memory under one of the two mapping types. The modified allocator uses an internal interface to serve allocation with both memory and cache demand, for instance, *malloc(mem_size, cache_map)*, where the *mem_size* is the user program's memory requirement, and the *cache_map* comes from the NightWatch's advice for cache mapping. The design of NigthWatch is general and portable so that it is quite convenient to extend the original allocator to support NightWatch. For example, it takes less than 700 lines of code modification to integrate NightWatch into *tcmalloc*, and 500+ lines of code modification for *ptmalloc2*.

The *operating system* manages the memory mappings. We modify the Linux kernel to support both open mapping and restrictive mapping, as discussed in Section 2. When the memory allocator runs out of certain type of memory, it acquires more memory via system calls. In our design, we extend the *mmap* system call as the interface to return free memory with multiple mappings.

5 Design and Implementation

In this section, we describe in detail the design and implementation of NightWatch's main components.

5.1 The Locality Monitor

Types of chunks. The locality monitor aims to evaluate a chunk's locality property by sampling its data accesses, and then classifies the chunk into two types: *polluter chunk* and *normal chunk*.

A polluter chunk is one with poor locality and caching it brings little performance gain, and therefore should be mapped to cache in a restrictive way. In practice, only a small fraction of the polluter chunks are completely non-temporally accessed: many of them still receive burst temporal accesses. As long as the burst accesses can fit in a cache region provided by the restrictive mapping, the chunks are treated as polluters. The remaining chunks are normal chunks. Data of normal chunks can get timely reuse before getting evicted from the cache. They are the normal users of the cache, and the potential victims of the polluter chunks. If normal chunks are identified, NightWatch will suggest to allocate them with open mapping.

Monitoring and mapping. The monitor samples a subset of the pages in each chunk periodically (i.e., every five seconds in our implementation). For each sampled page, the monitor records a hit or miss according to its access events and obtains a miss rate for the chunk. To obtain a stable result, multiple rounds of sampling are performed within each *sample collection* period, until the miss rate converges. Here, the condition of convergence is that the recent rate differs from the historical average rate by at most a given margin (0.1 in our case).

After the sample collection phase, the current miss rate of the chunk is generated. The miss rate is used to determine the *chunk type* and the proper mapping type, called the *target mapping type*. The current mapping type of the chunk may not match the target mapping type, and calls for a mapping switch. NightWatch does not notify the allocator to switch the mapping immediately, as many chunks are short-lived and may exhibit short-term locality variation; instead it waits until the next monitoring phase and checks the situation again. If the mismatch persists, it starts mapping switch.

In practice, a chunk's access pattern may change over time and remain unstable in the long term. In this case,

NightWatch does not keep switching between the mapping types, as remapping is expensive. There are two possible ways in which a chunk's role starts alternating: polluter→normal→ ... and normal→ ..... In the first case, NightWatch performs remapping only once, that is, from restrictive to open mapping, and stops monitoring the chunk afterwards. In the second case, NightWatch only performs the first open mapping and ignores future changes. The principle of mapping switch is that NightWatch would rather treat an unstable or primarily polluter chunk as a normal one, than restrict a normal one to its disadvantages. The consequence of this conservative mapping policy is that, for these particular cases, NightWatch degenerates the cache-aware allocator to a traditional memory allocator, and does not perform worse than a cache pollution unaware allocator. The conservative policy also has downsides. For example, for chunks with infrequent changing locality properties, the benefit from cache control may exceed the remapping overhead. One possible approach is to use methods similar to Branch History Table to detect phases of stable chunks and to make more aggressive cache control.

For small chunks below one page, NightWatch performs page alignment before monitoring. When a small chunk is selected as a sample, NightWatch will force the allocator to align the chunk to page size, so that the locality information monitored at page granularity can still represent the sampled chunk. The alignment will not cause much space waste, as NightWatch only needs to sample a very small percentage of the allocations.

Identification of access misses. To evaluate whether a sampled page encounters an access hit or miss, the monitor records a pair of successive references for the page, and estimates whether the second reference is timely enough to hit the CPU cache. There are two issues to be considered here. First, due to the nature of access locality, the reference events to a certain page tend to be clustered. As a result, a simple sampling procedure may find most of the collected reference pairs falling into the page's burst access interval. For mapping selection, such samples are meaningless, because they are frequent enough to hit the cache in any of the mapping types.

The second issue is the locality measurement. Access locality is commonly measured by *reuse distance*, which is defined as the number of distinct data accessed since the last access to the sampled data. Reuse distance is the same as LRU stack distance [19]. Although reuse distance provides a basis for precise cache miss prediction, it is very costly to be used directly for on-line monitoring – measuring reuse distance requires tracing the full data references. At present, no commodity hardware supports such measurement, and there is no efficient software approach either.

In NightWatch, we exploit the relation between cache events and reuse distance to address the two issues. Let Δn and Δm be the number of cache accesses and the number of cache misses between a reference pair, respectively. Also, let $C_{restrict}$ and C_{open} be the cache spaces (at cache line granularity) assigned by restrictive mapping and open mapping, respectively. Then we have

1. If $\Delta n < C_{restrict}$, the data reuse is frequent enough to fit either of the mapping types;
2. If $\Delta m > C_{open}$, the data will be evicted from cache before its reuse even with open mapping.

Note that the reuse distance measures distinct data accesses on the cache, and Δn is an access measurement without the *distinct* condition. Hence, Δn is an upper bound for the data's reuse distance. If Δn is less than $C_{restrict}$, the reuse distance will not exceed $C_{restrict}$. On the other hand, if Δm is larger than C_{open}, the distinct data installed on the cache is beyond C_{open}. Thus the reuse distance is already larger than C_{open} before the data reuse. So, even with open mapping, the data reuse will still trigger a cache miss.

The sampling process is as follows. In each monitoring cycle, a sampled page will be set with read/write protection for trapping page references. When the first reference arrives, the protection will be removed until the cache access volume reaches C_{open}, so that meaningless burst accesses can be skipped. Then, the monitor records the current number of cache misses, and waits for the second reference. When the second reference is trapped, the monitor checks the increment of cache misses Δm, and compares Δm and C_{open} to estimate whether the second page reference misses the cache or not.

Effectiveness of sampling. To evaluate the effectiveness of our sampling mechanism, we evaluate SPEC CPU 2006 benchmark suite. There are 1 million data access randomly sampled from each of the programs. For each of the sample, we compare our estimation of cache miss with the precise result given by off-line reuse distance analysis. Figure 10 shows an average false rate of

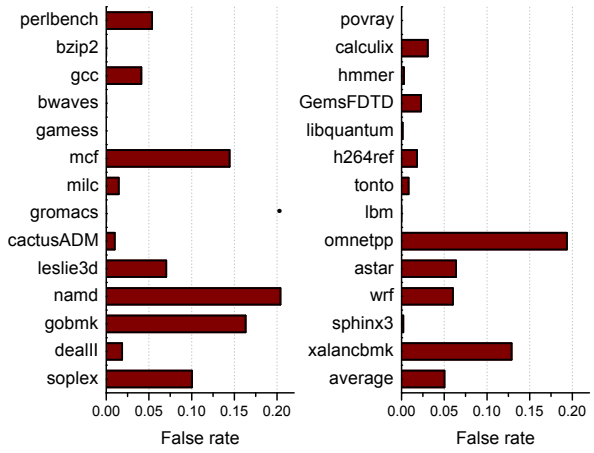


Figure 10: Cache miss estimation false rate

6.0%. Six out of the 27 programs have a false rate over 10% (*mcf, namd, gobmk, soplex, omnetpp, xalancbmk*). Compared with the reuse distance analysis, our method is much more lightweight and thus practical. As we will show later, it is accurate enough for the purpose of cache management. More importantly, since $\Delta m > C_{open}$ is a sufficient condition for cache miss events, our monitor can only take a miss event for a hit mistakenly, and may further regard a polluter chunk as a normal one, which is in line with our conservative mapping principle.

Minimizing monitoring overhead The optimization of monitoring overhead is carried out at two levels: 1) Page level. Due to high locality similarity between pages within a chunk, page sampling rate does not need to be high to allow accurate locality estimation. As we will show later in Section 6.3, for an estimation error below 5%, the required number of sampled pages is approximately $s^{0.65}$, where s is the total number of pages in a chunk. This sublinear relation means that the sampling method can scale to large chunks. 2) Chunk level. NightWatch tries to skip chunks in the request stream when it finds a chunk's prediction type to match the monitoring result. Specifically, upon each successful prediction, NightWatch doubles the sampling interval for reduced overhead. On the other hand, when a prediction is found to be false, the sampling interval falls back to zero. Based on inter-chunk locality similarity, this mechanism dramatically reduces chunk sampling rate, while guaranteeing a high prediction success rate.

5.2 The Locality Predictor

As we have described, once the locality monitor detects that a chunk's current mapping type does not match its actual locality pattern, the monitor will look for chances of mapping switch. The mapping switch mainly targets the first few chunks in their allocation context. For subsequent chunks in the context, their locality information can be predicted from the previously allocated chunks and thus mapping switch becomes much less necessary.

The prediction process is as follows. The predictor first analyzes the call stack of the allocation request to determine its allocation context. Then, it checks the mapping type of its two preceding chunks to make a prediction. If the current allocation request is the first one of its context, or the monitor has not yet determined the chunk type of its predecessors, the predictor assigns open mapping to the current request. If one of the predecessors is a normal chunk, the open mapping will also be applied. For other cases, restrictive mapping will be applied.

Notice the conservative policy of mapping that the predictor performs. Whenever there is inconsistency, the predictor chooses open mapping for the allocation request, for the same reason of the locality monitor's bias toward normal chunks in chunk type determination.

Accelerating locality prediction. NightWatch uses call stack as the identifier of an allocation context. We trace the 10-depth call stack of the current allocation function, and hash together all the program counters in every stack frame, so that the allocation context of a chunk can be determined by the hash value.

Some programs, especially those programmed with object-oriented languages, may extensively use small chunks. For those small chunks, we use a size-to-mapping table to provide a faster way to give the mapping type predictions. In our design, the table contains 64K entries, each corresponding to a set of chunks whose size equals the entry's index. The monitor's results are used to determine whether to invalidate a table entry or not. If the chunks in a table entry have the same mapping type, NightWatch directly returns a mapping type prediction for the coming allocation requests; otherwise, if the locality monitor detects inconsistency in this respect later on, the corresponding entry will be invalidated, in which case NightWatch will resort to the call stack tracing approach to find the allocation context and make prediction.

6 Evaluation

6.1 Experiment Setup

We conducted our experiments on a machine with four 2.13GHz quad-core Intel Xeon E7420 processors. The four cores in each of the processor share a set-associative 8MB cache, with 16,384 sets in total. Both stride prefetching and adjacent-line prefetching are enabled on the processors. The cache contains 256 colors; open mapping can use all the colors, while restrictive mapping can use only 32 colors. The server has 16GB memory, with eight 2GB fully buffered DIMMs. The operating system is CentOS 6.0, with Linux kernel 2.6.32-71.

We compare the NightWatch assisted *tcmalloc* (hereinafter *nw_tcmalloc* for short) with the original *tcmalloc* implementation. The *tcmalloc* used in our experiment is gperftools-2.4. The benchmark set consists of 27 programs of the SPEC CPU2006 benchmark suite. These programs are compiled with gcc 4.4.4.

6.2 Performance Improvement

We classify the 27 programs into three categories: polluters, victims, and neutral, based on two metrics: cache sensitivity and cache access rate. Cache sensitivity is defined as $T_{open}/T_{restrict}$, where T_{open} and $T_{restrict}$ are the program's execution times under open mapping only and restrictive mapping only, respectively. The cache access rate is the number of cache accesses per 1K cycles. The polluter programs have $cache_sensitivity < 10\%$ and $cache_access_rate > 5\%$. Their data can hardly get timely reuse after installed into the cache. The second category is victim programs, with $cache_sensitivity >$

Category	cache sensitivity	cache access rate (access per 1k cycle)	Benchmark
Polluter	$< 10\%$	> 5	410.bwaves 433.milc 459.GemsFDTD 462.libquantum 481.wrf
Victim	$> 20\%$	—	401.bzip2 403.gcc 429.mcf 447.dealII 450.soplex 470.lbm 471.omnetpp 473.astar 482.sphinx3 483.xalancbmk
Neutral	$[10\%, 20\%]$	< 5	400.perlbench 416.gamess 435.gromacs 436.cactusADM 437.leslie3d 444.namd 445.gobmk 453.povray 454.calculix 456.hmmer 464.h264ref 465.tonto

Table 1: Categories of benchmark programs. Cache sensitivity reflects the slowdown of program execution under pure restrictive mapping compared with under conventional (open) mapping.

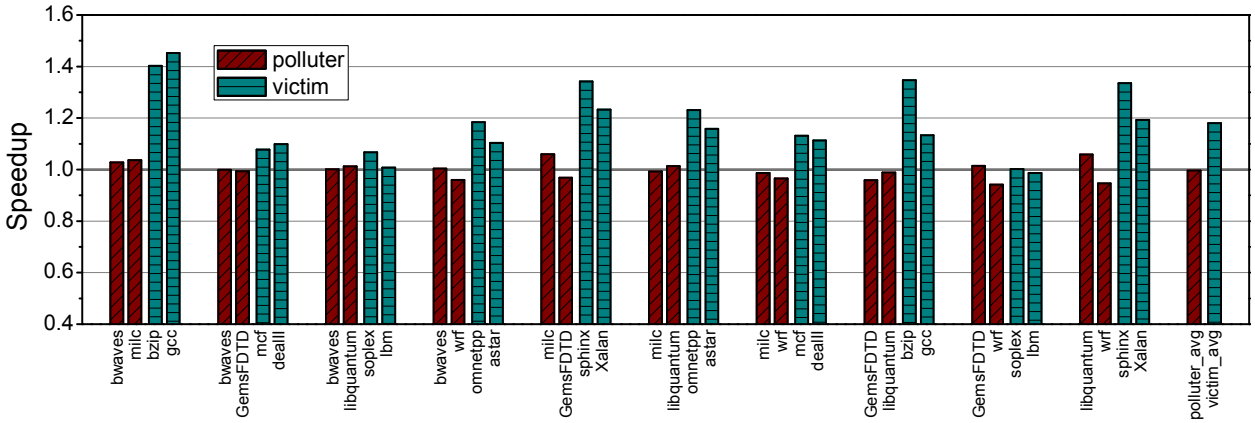


Figure 11: Performance speedup of polluter-victim combinations

20%, meaning that they have a high demand on cache, and are vulnerable to cache pollution. The remaining programs fall in the neutral category, which has limited demand on cache. They neither pollute cache, nor get polluted. The program classification is listed in Table 1.

Each workload used in our performance evaluation is a unique combination of four programs selected from two categories. Programs from different categories are combined only once to increase variety. There is one quad-core processor used in our experiments. We bind the programs to the four cores on this processor, so as to avoid the overhead of task migration. In addition, to make sure every program has three co-runners throughout the execution, we restart the early terminated programs until the longest one is completed, following the approach in previous work [14, 22].

In our experiment, we mainly evaluate the *polluter-victim* combination, in order to highlight the impact of polluters. The result is illustrated in Figure 11. As we can see, most of the victim programs can benefit substantially from NightWatch, with an average speedup of 1.18. In the combination $(bwaves, milc, bzip, gcc)$, for example, *gcc* achieves the highest speedup of 1.45. On the other hand, NightWatch has little impact on the polluter programs' performance in general. Compared with *tcmalloc* – which uses open mapping only – *nw_tcmalloc*'s dual-mapping scheme imposes little side effect on the polluters, due to two reasons. First, by leveraging intra- and inter-chunk locality similarities, *nw_tcmalloc* incurs very low overhead. As we will show later, for most of the programs, the overhead is less than 1%. The second reason is that *nw_tcmalloc* is able to distinguish between polluter and normal chunks and map them to cache in different ways. Thus, it does not harm the performance of the normal chunks used by the polluter programs.

It is worth noting that instead of staying unaffected or showing slight slowdown, several polluter programs, for example *bwaves* and *milc*, get noticeable speedup from *nw_tcmalloc*. This is somewhat counterintuitive, since for these programs, almost all the chunks are polluters. After applying restrictive mapping, the available cache space assigned to these programs is reduced from 8MB (under open mapping) to 1MB. We have run the two programs separately on two cores using *nw_tcmalloc* in comparison with *tcmalloc*, and observed speedups of 0.996 and 0.984 for *bwaves* and *milc*, respectively, which confirms the negative effect of reduced cache resource. Yet, when run with the victim programs *bzip* and *gcc*, the speedups surprisingly exceed one. A similar phenomenon was also observed by Lin et al. [14]. The reason behind this phenomenon is as follows: once the polluter chunks get restricted in cache usage, the normal chunks will get more cache space and thus their cache miss rate will be reduced. This results in a reduction of memory bandwidth pressure, and a smaller queuing delay at the memory controller. For example, for the first workload we tested, when *nw_tcmalloc* is applied, the overall cache miss rate is reduced by 4%. This reduction

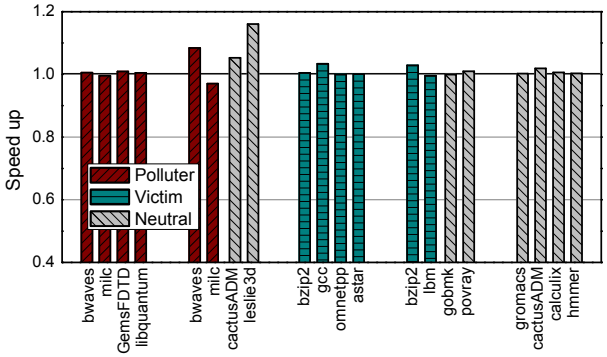


Figure 12: Performance speedup of five program combinations

turns out to be beneficial to the overall system performance, positively affecting *bwaves* and *milc* as well.

Figure 12 shows the performance of other combinations of benchmark programs, including all-polluter, all-victim, all-neutral, polluter-neutral, and victim-neutral. Overall, the programs experience very small changes in their performance, agreeing with the nature of these combinations, and suggesting that *nw_tcmalloc* retains system performance when it cannot bring improvement.

6.3 Efficiency of Locality Monitor

The locality monitor needs to collect access information of a random subset of pages in a chunk to determine the locality property of the chunk. The number of pages sampled, or sampling count, reflects a trade-off between sampling overhead and accuracy of locality estimation. Given a target upper bound of estimation error (e.g., 5%), we want to find the minimum sampling count, *MSC*, as a function of chunk size (in number of pages). We collect the page access statistics of all the programs' chunks, and for each chunk size, we experiment with increasing sampling count until the estimation error rate drops below 5%. For clarity purpose, we divide the chunk sizes into intervals, and calculate the average *MSC* for chunk sizes within each interval. Figure 13 shows the sampling count against chunk size in dot line. The measured curve can be roughly approximated by a straight line with slope 0.65, which translates to a sub-linear sampling count function $MSR(s) = s^{0.65}$, where s is the chunk size. This means the required sampling overhead grows much slower than a linear function, and thus can scale to large chunks while ensuring good accuracy of locality estimation.

6.4 Accuracy of Locality Predictor

For backward compatibility, *nw_tcmalloc* adopts the standard allocation interfaces. Since these interfaces only allow the programmer to specify the request's memory demand, *nw_tcmalloc* has to rely on NightWatch's locality predictor to infer the implicit cache demand. Therefore, a high prediction success rate is crucial to the

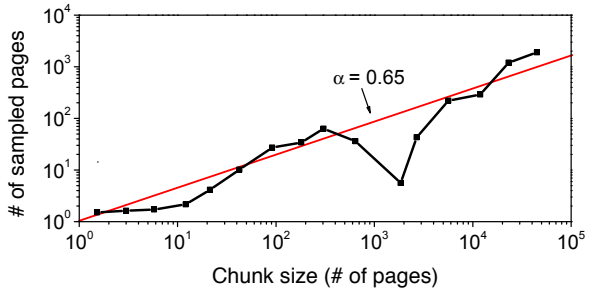


Figure 13: Number of sampled pages vs. chunk size

Benchmark	Succ. rate	Benchmark	Succ. rate
perlbench	99.8%	povray	99.7%
bzip2	95.8%	calculix	92.9%
gcc	96.2%	hmmer	99.5%
bwaves	99.3%	GemsFDTD	100.0%
gamess	98.6%	libquantum	77.8%
milc	99.7%	h264ref	92.2%
gromacs	100.0%	tonto	100.0%
cactusADM	100.0%	omnetpp	100.0%
leslie3d	100.0%	astar	97.5%
namd	58.1%	wrf	99.4%
gobmk	92.8%	sphinx3	100.0%
dealII	93.6%	xalancbmk	100.0%
soplex	94.3%	**Average**	**95.5%**

Table 2: Mapping type prediction success rate

efficiency of *nw_tcmalloc*.

If a chunk's predicted mapping type matches its actual locality property, the prediction is considered a success. Since NightWatch does not monitor every chunk, we collect prediction results and monitoring results in different runs in order to calculate prediction success rate. In the first run, the prediction results are collected in standard system configurations. In the second run, we force NightWatch to monitor every chunk throughout the benchmark's execution, and use the monitoring results to evaluate the predictor's accuracy. There are two programs (*mcf* and *lbm*) with no prediction information, because they do not provide any prediction opportunities. For example, *lbm* allocates two 214400KB chunks, each with an exclusive allocation context. Since there is no chunks previously allocated in the contexts, there is no clue for prediction.

Table 2 shows the prediction success rate of NightWatch. Due to inter-chunk locality similarity, a high prediction success rate is attained for most of the programs – 14 out of 23 programs have a prediction success rate over 99%. Note that the predictor falls short for *namd*, with a low success rate of 58%. This is due to the adaptive sampling method of the locality monitor, which doubles the sampling interval (i.e., number of skipped chunks in a row) upon every successful prediction. While helping to reduce monitoring overhead, such

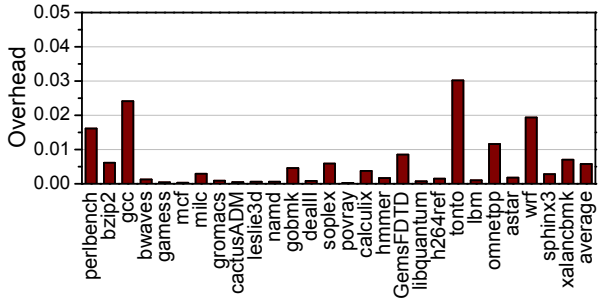


Figure 14: The overhead of *nw_tcmalloc*, defined as the fraction of a program's execution time spent on cache management

a design responds slowly to locality property change occurring across chunks within an allocation context. If mapping switching happens during a large sequence of skipped chunks, all follow-up chunk requests will receive a wrong prediction, until the next sampling point is reached. Nevertheless, such a case rarely happens for the 27 benchmarks; for most of the programs inter-chunk locality similarity remains valid, yielding an average prediction success rate of 95.5%.

6.5 Overhead Analysis

nw_tcmalloc's integrated cache management adds an extra time cost to a program's execution. The ratio between this time cost and a program's overall execution time defines *nw_tcmalloc*'s *overhead*. Specifically, *nw_tcmalloc*'s cost consists of three parts: the monitor's time cost T_{mon}, the predictor's time cost T_{pred}, and the time spent on mapping switching $T_{mswitch}$. Due to the high prediction success rate, $T_{mswitch}$ is negligible – compared with the overall execution times of the 27 benchmarks, which range from 250 to 1000 seconds, $T_{mswitch}$ is less than 1 second.

Figure 14 presents the overheads of *nw_tcmalloc*. On average, the overhead is only 0.57%, with a maximum 3.02%. Furthermore, for 22 of the 27 programs, *nw_tcmalloc*'s overhead is less than 1%. This indicates that *nw_tcmalloc* is highly efficient while offering performance benefits.

The monitor's cost, T_{mon}, is caused by locality evaluation for sampled pages and chunks, and thus depends on the total size of allocated chunks, denoted by $SIZE_{total}$. NightWatch implements a number of optimization strategies to reduce the cost. Figure 15 shows T_{mon} of the 27 benchmark programs sorted by $SIZE_{total}$. It can be seen that T_{mon}, which varies across two orders of magnitude, increases much more slowly than $SIZE_{total}$, which spans four orders of magnitude. This sublinear growing trend suggests that the monitoring cost scales very well with total allocation size. Notice that T_{mon} does not always increase with $SIZE_{total}$, because T_{mon} also depends on the size distribution of chunks, and how chunks are clustered

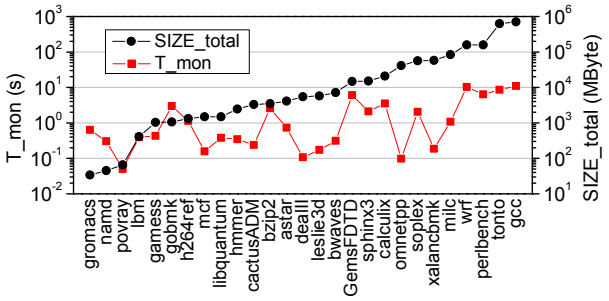


Figure 15: T_{mon} vs. $SIZE_{total}$ for 27 programs, sorted by $SIZE_{total}$

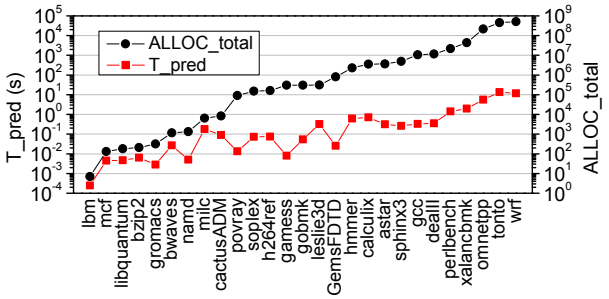


Figure 16: T_{pred} vs. $ALLOC_{total}$ for the programs, sorted by $ALLOC_{total}$

in contexts: small sized chunks and low clustering effect in contexts will reduce the benefit of locality similarities, thus causing a higher time cost.

The predictor's cost of a program, T_{pred}, is determined by the total number of allocations, denoted by $ALLOC_{total}$. Figure 16 presents T_{pred} of the 27 benchmark programs sorted by $ALLOC_{total}$. With the help of size-to-context lookup table, T_{pred} grows much more slowly than a linear function of $ALLOC_{total}$. On average, one second's cost allows NightWatch to make 34 million predictions, which make it suitable for programs that involve extremely frequent chunk allocation. For example, the *wrf* program takes 1000.4 seconds to complete, making a total of 500 million allocations. It takes NightWatch only 23 seconds to make all the predictions, accounting for 2.3% of the program's execution time.

6.6 Evaluation on Ptmalloc2

Due to space limits, we only briefly report on our evaluation on *ptmalloc2*. For the *pollter-victim* combinations, the NightWatch integrated allocator improves *victim*'s performance by up to 51% (18% on average), with an average overhead of 0.6% (up to 3.0%).

7 Related Work

There has been extensive research on improving cache efficiency for multi-programmed workloads. In this section we discuss the main technical approaches used and related work in the area. For clarity, we list the representative work in Table 3.

Cache-aware memory allocator: From the perspective of resource management, perhaps the closest systems to NightWatch assisted allocators are *ULCC* [6] and *ccontrol* [28]. Both designs attempt to incorporate cache control into the memory allocator in order to achieve higher cache efficiency. They enable cache control by providing special interfaces to programmers to modify the program's source code for improved resource utilization. Apparently this approach requires a deep understanding of the program's data access behavior. Adding to the complexity of this problem is the fact that many programs' data locality property varies at run time due to multiple factors (such as input, software configuration). A proper grasp of these complicated issues is thus beyond the capability of common programmers. Given the nonstandard interfaces, these designs also fail to provide transparent support for legacy programs.

Our work is the first to *transparently* integrate cache management into dynamic memory allocation. Compared with previous solutions, NightWatch assisted allocators hide the complexity of cache management and provides standard interfaces. This makes it easy to use and fully compatible with legacy software.

Cache bypassing: Commodity processors provide cache bypass instructions to bypass weak-locality data. Rus et al. [23] propose to develop automatical tools to help identify weak-locality string operations and transform them to cache bypass instructions. Sandberg et al. [25] employ off-line reuse distance analysis to characterize the access locality of the overall program, and replace non-temporal data accesses with bypass instructions. These approaches need off-line analysis and hence fail to adapt to a program's dynamic runtime behavior.

Software cache partitioning: Cache partitioning has been proven an effective approach to improve cache utilization [7, 13, 14, 22, 27, 32, 33]. Lin et al. [14] propose to partition the shared cache for co-running programs and isolate cache pollution from weak-locality data accesses. Their dynamic partitioning technique adjusts the cache partitions to accommodate locality changes of data. RapidMRC [29] guides cache partitioning to achieve optimal speedup, with the help of the *Miss Rate Curves* (MRC) of each program. The hot-page coloring method [31] enforces cache partitioning for hot pages to reduce the overhead of cache re-partitioning. ROCS [27] clusters weak-locality pages to a dedicated pollution buffer on cache. Soft-OLP [16] analyzes the reuse distance of each major data object, and performs proper cache allocation based on the object's locality properties and interactions. These techniques suffer from a number of limitations. First, some techniques need specialized hardware support from the processor. For example, [27, 29] use POWER processor's *Sampled Data Address Register* (SDAR) to evaluate MRC and pages' miss rate, which is not available in other processors. Second, it is hard to effectively obtain data reuse information [5, 16, 26]. It is reported in [21] that the collection of reuse information can degrade the performance by a factor of 13 to 50, even with accelerated instrumentation on 64 processors. Third, re-partitioning cache is expensive. Page recoloring for cache re-partitioning incurs significant overheads [31].

	Locality analysis	Dynamic memory allocation	Transparent to user	Specific hardware support
ULCC [6]	manually	yes	no	no
ccontrol [28]	manually	yes	no	no
Soft-OLP [16]	off-line	yes	no	no
Sandberg et al. [25]	off-line	no	yes	no
Rus et al. [23]	off-line	no	yes	no
RapidMRC [29]	on-line	no	yes	yes
hot-page color [31]	on-line	no	yes	yes
ROCS [27]	on-line	no	yes	yes
NightWatch	on-line	yes	yes	no

Table 3: Comparison of shared cache pollution management techniques

Compared with the above solutions, NightWatch provides the benefits without their limitations. It can efficiently obtain the data locality information on commodity processors and reduce the cache re-partitioning overhead with the help of data locality prediction.

8 Conclusion

In this paper we have presented NightWatch, an efficient cache management subsystem designed for memory allocators. The distinguishing feature is that NightWatch provides runtime support for pollution control, using only standard allocation interfaces and assuming no special hardware support. At the heart of the solution are two observations about locality correlation of data that are exploited to realize highly efficient locality monitoring and prediction. We have demonstrated the efficacy of NightWatch through extensive experiments, showing speedup of up to 1.5X for pollution victim programs at very low overheads. It should be noted that NightWatch is not limited to multi-program workloads; the multithreaded version is also open-sourced. In future, we plan to conduct a comprehensive evaluation on the effectiveness of the multithreaded version.

9 Acknowledgements

This paper is supported by China National Natural Science Foundation under grant No. 61322210, 61379135, 61433019, Doctoral Fund of Ministry of Education of China under grant No. 20130142110048, National Hightech Research and Development Program of China (863 Program) under grant No. 2015AA015303, US National Science Foundation grant ACI-award No. 1432892.

References

[1] AKRITIDIS, P. Cling: A memory allocator to mitigate dangling pointers. In *Proc. USENIX Security Symposium 2010*, pp. 177–192.

[2] AMDINC. Amd64 architecture programmers manual volume 2: System programming.

[3] BERG, E., AND HAGERSTEN, E. Fast data-locality profiling of native execution. In *Proc. SIGMETRICS 2005*, ACM, pp. 169–180.

[4] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices 35*, 11 (2000), 117–128.

[5] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. In *Proc. PLDI 2003*, ACM, pp. 245–257.

[6] DING, X., WANG, K., AND ZHANG, X. Ulcc: A user-level facility for optimizing shared cache performance on multicores. In *Proc. PPoPP 2011*, ACM, pp. 103–112.

[7] DUONG, N., ZHAO, D., KIM, T., CAMMAROTA, R., VALERO, M., AND VEIDENBAUM, A. V. Improving cache management policies using dynamic reuse distances. In *Proc. MICRO 45* (2012), IEEE Computer Society, pp. 389–400.

[8] EVANS, J. Scalable memory allocation using jemalloc. `http://www.canonware.com/jemalloc/`, 2011.

[9] GHEMAWAT, S., AND MENAGE, P. Tcmalloc: thread caching malloc. google performance tools. `https://code.google.com/p/gperftools/`, 2004.

[10] GLOGER, W. Ptmalloc. `http://www.malloc.de/en/`, 2006.

[11] INTELINC. Intel® 64 and ia-32 architectures software developers manual.

[12] JALEEL, A., NAJAF-ABADI, H. H., SUBRAMANIAM, S., STEELY, S. C., AND EMER, J. Cruise: cache replacement and utility-aware scheduling. In *ACM SIGARCH Computer Architecture News* (2012), vol. 40, ACM, pp. 249–260.

[13] KIM, H., KANDHALU, A., AND RAJKUMAR, R. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *Proc. ECRTS 2013*, IEEE, pp. 80–89.

[14] LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. HPCA 2008*, pp. 367–378.

[15] LIU, R., AND CHEN, H. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Asia-Pacific Workshop on Systems* (2012), ACM, p. 15.

[16] LU, Q., LIN, J., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Proc. PACT 2009*, pp. 246–257.

[17] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI 2005*, ACM, pp. 190–200.

[18] MANIKANTAN, R., RAJAN, K., AND GOVINDARAJAN, R. Nucache: An efficient multicore cache organization based on next-use distance. In *Proc. HPCA 2011* (2011), IEEE, pp. 243–253.

[19] MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[20] MAUERER, W. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

[21] NIU, Q., DINAN, J., LU, Q., AND SADAYAPPAN, P. Parda: A fast parallel reuse distance analysis algorithm. In *Proc. IPDPS 2012*, pp. 1284–1294.

[22] QURESHI, M. K., AND PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO 39* (2006), IEEE Computer Society, pp. 423–432.

[23] RUS, S., ASHOK, R., AND LI, D. X. Automated locality optimization based on the reuse distance of string operations. In *Proc. CGO 2011*, IEEE Computer Society, pp. 181–190.

[24] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 57–68.

[25] SANDBERG, A., EKLÖV, D., AND HAGERSTEN, E. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proc. SC 2010*, IEEE Computer Society, pp. 1–11.

[26] SCHUFF, D. L., KULKARNI, M., AND PAI, V. S. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proc. PACT 2010*, ACM, pp. 53–64.

[27] SOARES, L., TAM, D., AND STUMM, M. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proc. MICRO 41* (2008), IEEE Computer Society, pp. 258–269.

[28] SWANN PERARNAU, M. T., AND HUARD, G. Controlling cache utilization of hpc applications. In *Proc. ICS 2011*, ACM, pp. 295–304.

[29] TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proc. ASPLOS 2009*, ACM, pp. 121–132.

[30] WU, C.-J., JALEEL, A., MARTONOSI, M., STEELY JR, S. C., AND EMER, J. Pacman: prefetch-aware cache management for high performance caching. In *Proc. MICRO 44* (2011), ACM, pp. 442–453.

[31] XIAO, Z., SANDHYA, D., AND KAI, S. Towards practical page coloring-based multicore cache management. In *Proc. EuroSys 2009*, ACM, pp. 89–102.

[32] XIE, Y., AND LOH, G. H. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 174–183.

[33] YE, Y., WEST, R., CHENG, Z., AND LI, Y. Coloris: A dynamic cache partitioning system using page coloring. In *Proc. PACT 2014* (2014), ACM, pp. 381–392.

Secure Deduplication of General Computations

Yang Tang, Junfeng Yang
Columbia University
{ty,junfeng}@cs.columbia.edu

Abstract

The world's fast-growing data has become highly concentrated on enterprise or cloud storage servers. Data deduplication reduces redundancy in this data, saving storage and simplifying management. While existing systems can *deduplicate computations* on this data by memoizing and reusing computation results, they are insecure, not general, or slow.

This paper presents UNIC, a system that securely deduplicates general computations. It exports a cache service that allows applications running on behalf of mutually distrusting users on local or remote hosts to memoize and reuse computation results. Key in UNIC are three new ideas. First, through a novel use of code attestation, UNIC achieves both integrity and secrecy. Second, it provides a simple yet expressive API that enables applications to deduplicate their own rich computations. This design is much more general and flexible than existing systems that can deduplicate only specific types of computations. Third, UNIC explores a cross-layer design that allows the underlying storage system to expose data deduplication information to the applications for better performance.

Evaluation of UNIC on four popular open-source applications shows that UNIC is easy to use, fast, and with little storage overhead.

1 Introduction

The world's data has been fast exploding for many years. It is estimated that in 2011 alone, 1.8 zettabytes of data were created, and the overall data will grow by 50× by 2020 [21]. This massive amount of data comes in greatly varying forms, ranging from personal photos and videos, to office documents and web pages, to source files, binary programs, and virtual machine images, and to data collected from user clicks or physical sensors.

Meanwhile, the storage of this data has become highly concentrated. It is common practice for enterprises to store data on centralized, powerful storage servers for ease of management [34]. The cloud computing paradigm has migrated data into the cloud so that the computations can be closer to the data. For instance, several organizations have put 56 public data sets totaling 761.2TB onto Amazon Web Services [2]. Even consumers are beginning to aggregate their personal data into the cloud for convenience. For instance, Google, Dropbox, Amazon, and Microsoft all provide the option for users to automatically upload pictures and videos shot using their mobile devices. Facebook stores over 260 billion personal photos [6].

This highly concentrated, massive data poses challenges for storage provisioning and management. Fortunately, prior work has shown that a significant portion of the data is redundant [22] and that *data deduplication* can hugely reduce the storage needed to hold the data and simplify management [13]. For instance, *file deduplication* detects when multiple files have the same data and stores the unique data only once [8]. This scheme is particularly useful when the same file is copied, such as when a user makes a copy of her friend's shared video on Dropbox. *Block deduplication* breaks files down to variable [20, 24] or fixed [36] size blocks and stores each unique block of data once. This scheme is particularly useful for files that are similar but not exactly identical, such as different versions of a document and virtual machine images built from the same OS family. These deduplication schemes have been long prevalent in enterprise storage servers [13]. With the trend of moving consumer data into the cloud, these schemes have also become popular among cloud storage providers such as Dropbox [31].

Not only can data be redundant, the computations on top of the data can also be redundant. For instance, a user may scan her Dropbox files for viruses, while another user runs the same virus scanner on a similar set of files. Different users may be doing the same computations on the public data sets in AWS, such as building an inverted

index for the web pages in CommonCrawl [11]. Given the same input data, the same deterministic computation always produces the same result. Thus, if the computation is slow, it is typically more efficient to memoize [23] and reuse the result than redoing the computation. We term this technique *computation deduplication*.

Several prior systems deduplicate computations (*e.g.*, [9, 15]). However, three main challenges prevent these systems from effectively deduplicating computations in today's cloud or enterprise environments:

First, how can we deduplicate computations done by *mutually distrusting* users? Storage providers such as Dropbox aggregate data from many users who do not necessarily trust each other. Even in an enterprise setting, users frequently have different data access permissions. One naïve approach is to memoize computation results in a cache every user can read or write, but this approach provides neither integrity or security. A malicious user can easily poison the cache, by for instance marking files that contain viruses safe. She can also read results in the cache even though she has no permission to access the actual data in the results. Although this challenge may be solved with information flow tracking or access control systems, these systems are known to be difficult to configure and use.

Second, how can we deduplicate *general* computations? Prior systems deduplicate computations purely at the system level, assuming no cooperation from application developers. As a result, they handle only specific computations. For instance, `ccache` [9] deduplicates only the compilations of C/C++ programs, and Nectar [15] deduplicates the computations of programs written only in DryadLINQ [35], a specially designed language for large scale data-parallel workloads. However, the computations that users want to do on their data can be extremely rich, and it is unrealistic to require storage providers to understand all of them. For instance, while it may be feasible for Amazon to run some basic virus scanning software on the files it hosts, it is impossible for Amazon to understand every advanced virus scanner, every compression tool, and every image/video manipulation utility users want to run on their data.

Third, how can we effectively deduplicate computations on top of *deduplicated data*? Prior systems rely on custom methods to detect that data is redundant. For instance, `ccache` computes a hash of a preprocessed C/C++ source file and uses this hash to search its compilation cache. These methods incur unnecessary overhead when the data is deduplicated because the underlying storage system already knows what data is redundant.

This paper presents UNIC,[1] a system that securely deduplicates general computations. It exports a cache service that allows applications running on behalf of mutually distrusting users on local or remote hosts to memoize and reuse computation results. Key in UNIC are three new ideas:

First, through a novel use of code attestation, a classic primitive to attest what code is running to a (remote) party [29, 30], UNIC achieves both integrity and secrecy. To insert or query the result cache that UNIC maintains, UNIC generates a secure, non-forgeable key that attests to both the application code and the input data. This key strongly isolates applications from each other in the result cache. For instance, if a malicious user modifies the code of a virus scanner in attempt to poison the cached results of this virus scanner, the attempt would fail because the modified code leads to a different key. In addition, since this key is not forgeable, a malicious user cannot query UNIC's cache without already knowing the application code and the input. Since the user knows the code and input already, she can already compute the result by herself.

Second, UNIC provides a simple yet expressive API that enables applications to deduplicate their own rich computations. From a high level, this API supports an application to (1) insert *input* $\rightarrow$ *result* to the result cache UNIC maintains; and (2) query the cache with *input* and get back the cached *result* if any. This application-level computation deduplication design is much more general and flexible than prior system-level designs.

Third, UNIC explores a cross-layer design that allows the underlying storage system to expose data deduplication information to the applications for speed. Applications thus do not need to re-detect whether the input data is redundant. For instance, suppose two files A and B are identical so the filesystem deduplicates them, and UNIC exposes this data deduplication information to the applications. After a virus scanner scans file A, it can immediately skip file B without reading any data from B, significantly increasing its scanning speed.

Our implementation of UNIC stores cached results in Redis, a fast, scalable, replicated key-value store [27]. UNIC implements code attestation in a dynamically loadable Linux kernel module and considers the kernel to be trusted. It implements the computation deduplication API as a library, which applications link with. UNIC leverages ZFS [36], a file system that supports both file and block deduplication, to detect when data is deduplicated on behalf of the applications running with UNIC.

Evaluation of UNIC on four popular open-source applications shows that (1) it is easy to use (to support each application, we needed to change fewer than 1% lines of source code); (2) it is fast (it sped up applications by up to 21.4×); and (3) it incurs little storage overhead (it needed only 3.45% additional storage to cache the results).

The remainder of this paper is organized as follows.

[1] We name our system UNIC (pronounced "unique") because it is conceptually similar to the Unix `uniq` utility applied to computations.

The next section discusses the security model and UNIC's design. §3 describes UNIC's API and usage. §4 presents how UNIC leverages deduplicated data. §5 describes the implementation. §6 shows evaluation results. §7 discusses UNIC's security implications, §8 describes related work, and §9 concludes.

2 Security Model and Design

We begin with UNIC's assumptions, threat model, and the design of UNIC's protocol.

2.1 Assumptions and Non-assumptions

First, UNIC relies on a code attestation mechanism for integrity and secrecy of the cached results. It leverages this mechanism to bind a result to the code and input data that together produce the result. This mechanism can be implemented in multiple ways with different security strengths. For instance, UNIC could use TPM and isolation technologies such as Intel TXT [18] to realize code attestation, but doing so would incur both deployment and runtime overhead, negating our goal of being easy to use and fast. Therefore, for practical reasons, UNIC assumes that the OS is trusted and provides a function to attest the application code, and that the user does not have superuser privileges to interfere with that mechanism. This assumption matches well with many of today's mobile devices that run Chrome OS [14], iOS, and Android.

Second, UNIC assumes correct application code. For instance, when using UNIC, an application developer should use UNIC's API correctly. She should only memoize computations with deterministic results. UNIC also assumes that the application is free of vulnerabilities such as buffer overflows. We note that this assumption is common to almost all prior code attestation work.

Third, UNIC assumes that its underlying storage system provides reasonable security guarantees. To reuse results across sessions, UNIC persists them in an underlying storage system such as a file system. UNIC assumes that this storage system is properly configured such that an attacker cannot access the data stored without going through UNIC. This guarantee and UNIC's security mechanisms described in §2.3 together ensure the integrity and secrecy of its cache of computation results.

2.2 Threats

UNIC enables deduplicating computation among mutually distrusting users. Two attacks are particularly serious for UNIC: *cache poisoning* attacks UNIC's integrity, and *query forging* attacks UNIC's secrecy.

Cache poisoning. A malicious user may write a new application or modify an existing application in an attempt to poison the result cache. Her application may attempt to insert or overwrite entries belonging to a legitimate application. UNIC prevents this attack by isolating applications in the result cache: it guarantees that the cached data for one application can never be accessed by another application. Specifically, UNIC securely binds the computation code and the input data to the computation result leveraging a code attestation mechanism.

Query forging. A malicious user may write a new application or modify an existing application in attempt to query entries in the result cache that she cannot access, and gain information. UNIC prevents this attack again by isolating applications. When an application queries the cache, UNIC generates a search key that attests to both the code and the input data that generate the query. This key is unique to each application. One application thus cannot query entries of another application.

Several other attacks are possible, some of which can be prevented using simple mechanisms such as rate-limiting queries sent to UNIC. We briefly describe how they can be prevented in §7, and leave the implementation for future work.

2.3 Design

UNIC novelly leverages code attestation to cryptographically bind the *result* with the *code* and the *input* that produced the *result*, preventing cache poisoning and query forging attacks.

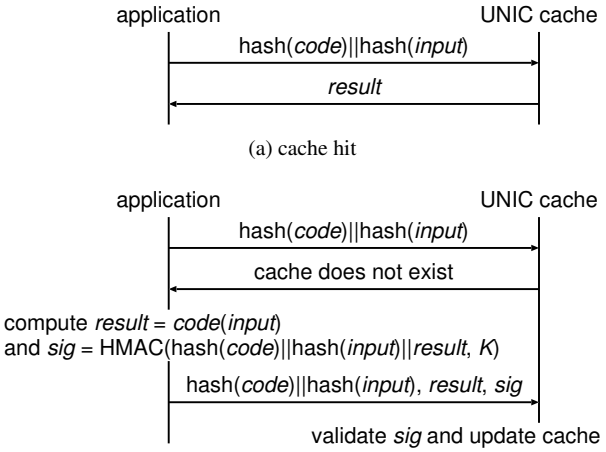
UNIC assumes a trusted OS that securely computes SHA-1 hash and HMAC. A secret key K is shared among trusted OSes. (Existing work [30] details how to distribute this key. We use symmetric key for efficiency; however asymmetric key works, too.) An attacker cannot forge HMAC($data, K$) without knowing K.

UNIC leverages code attestation to bind *result* to *code* and *input* that produced *result*. Specifically, it uses code attestation to compute two things:

(1) $result = code(input)$
// Run *code* on *input* to compute *result*.

(2) $sig = \text{HMAC}(\text{hash}(code)||\text{hash}(input)||result, K)$
// Bind *code*, *input*, and *result*. We use $||$ as the concatenation operator.

The assumptions on trusted OS, unprivileged user, and correct application code together guarantee that *result* is the correct result of running *code* on *input*. This code attestation mechanism further guarantees that (a) *sig* cryptographically attests that *result* is indeed produced by running *code* on *input*, which anyone with access to *code*, *input*, *result*, and K can verify; and (b) *sig* cannot be forged.



(a) cache hit

(b) cache miss

Figure 1: UNIC *protocol.*

UNIC protocol. The UNIC cache is a mapping of

$$\text{hash}(code)||\text{hash}(input) \rightarrow result$$

Since the hash function is collision resistant, the cache space for different computations are isolated.

When an application wants to compute $code(input)$, it sends $\text{hash}(code)||\text{hash}(input)$ to the UNIC cache. If cache exists (Figure 1a), UNIC sends back *result*. If cache does not exist (Figure 1b), the application computes both *result* and *sig*, and sends $\text{hash}(code)||\text{hash}(input)$, *result*, and *sig* to the UNIC cache. The UNIC cache validates that *sig* is indeed $\text{HMAC}(\text{hash}(code)||\text{hash}(input)||result, K)$, and updates the cache.

2.4 Security Analysis

The design of UNIC prevents cache poisoning as follows. Suppose an attacker replaces *result* with *bad_result* when inserting into UNIC. Because of code attestation, she cannot forge *sig*, so UNIC cannot validate *sig*. Suppose she modifies *code* into *bad_code* and computes *bad_result* to poison the cache. Because UNIC validates *sig*, she can only insert

$$\text{hash}(bad_code)||\text{hash}(input) \rightarrow bad_result$$

which cannot affect the cache entry of $\text{hash}(code)||\text{hash}(input)$. To avoid a malicious client from polluting the cache space, UNIC can employ a quota mechanism to limit the cache space for each client application.

This design also prevents an attacker from forging a query to steal *result*. To query cache, she must send $\text{hash}(code)||\text{hash}(input)$, so she must already have *code* and *input* because otherwise she would not be able to compute the hashes. Once an attacker has *code* and *input*, she can already compute *result* simply by running *code* on *input* herself. Thus, she cannot gain additional information with this query other than whether there is a result in the cache. §7 further discusses its implications.

```
1: void simple_virus_scanner(file, options) {
2:     buffer = read(file);
3:     result = scan_signature(buffer, options);
4:     print(result);
5: }
```

Figure 2: *A simple virus scanning application.*

3 UNIC API and Usage

UNIC provides a simple yet expressive API for applications to deduplicate their own rich computations. We first motivate our API design through an example, and then formally describe its interface.

3.1 Example

We motivate the design of UNIC API through a step-by-step example showing how a simple virus scanning application could use memoization to deduplicate computation. Conceptually, the application works like Figure 2. It reads the file content into a buffer, executes virus scanning algorithm on the buffer, and outputs the result.

In this piece of code, line 2 reads the file content from disk, potentially a time-consuming I/O operation. Line 3 performs some CPU-bound virus signature matching algorithm, potentially another time-consuming operation. Line 4 prints the result, which is relatively fast because the length of the scanning result (*e.g.*, "no virus found") is much smaller than the original file content. Therefore, we want to improve the performance on lines 2 and 3.

Memoizing Computations. We first examine how to use memoization to avoid duplicate computation on line 3. Since `scan_signature()` is a deterministic function over the input buffer and the signature-scanning options, if we could memoize the result the first time we perform the computation, we would be able to safely reuse the result later on the same input. To do so, we modify the application into Figure 3, using three functions that UNIC provides: `exists()`, `get()`, and `put()`. It first checks if the computation for the given buffer and options exists in the result cache (line 3). If so, it simply gets the memoized result (line 4). Otherwise, it performs the computation as before (line 6) and then puts the result into the cache (line 7).

As discussed in §2.3, the cache is not merely a mapping from the input to the result, but binds the computation code together with them. UNIC internally computes

```
1 : void simple_virus_scanner(file, options) {
2 :   buffer = read(file);
3 :   if (exists(scan_signature, buffer, options)) {
4 :     result = get(scan_signature, buffer, options);
5 :   } else {
6 :     result = scan_signature(buffer, options);
7 :     put(scan_signature, buffer, options, result);
8 :   }
9 :   print(result);
10: }
```

Figure 3: *First step: memoize the computation result.*

a non-forgeable authentication code that guarantees that the result (`result`) is indeed generated by the computation code (`scan_signature()`) over the input (`buffer` and `options`). The result cache is updated only if it can verify this authentication code.

Reducing I/O Operations. Memoizing the computation is good, but it would be better if we could also eliminate the need of reading the file content on line 2. This is not trivial because if we did not read the file in the first place, we would never know if the signature scanning is performed on the same content. Fortunately, it is possible if the file is stored on a deduplication-enabled storage.

A deduplication-enabled filesystem, such as ZFS [36], stores all files with the same content as a single copy. It does so by identifying the file content using a cryptographically collision-resistant hash (*e.g.*, SHA-256), and mapping all files with the same content to the same hash. These hashes are stored on the filesystem metadata, separate from the actual file content. Therefore, it creates a perfect opportunity for our application to tell if the file contents are the same without actually reading them.

Figure 4 shows the final version of the application. Instead of reading the file content up front, it now gets the unique hash of the file directly from the filesystem metadata using UNIC's `get_file_hash()` function (line 2), and uses the hash to identify the memoization (lines 3, 4, and 8). Since getting the hash is much faster than reading the whole file, we have further avoided the slow I/O operation when reusing a previously cached computation.

In practice, when using UNIC, the application developer does not need to worry whether the storage has deduplication enabled or not — she should always follow the final version in Figure 4 and use hash to identify the memoization. This is because UNIC *transparently* leverages storage deduplication information. Where such information is absent, UNIC computes and caches the hash by itself. This process is detailed in §4.

3.2 The API

The previous example illustrates the usage of the UNIC API which we now formally describe. It wraps OS-

```
1 : void simple_virus_scanner(file, options) {
2 :   hash = get_file_hash(file);
3 :   if (exists(scan_signature, hash, options)) {
4 :     result = get(scan_signature, hash, options);
5 :   } else {
6 :     buffer = read(file);
7 :     result = scan_signature(buffer, options);
8 :     put(scan_signature, hash, options, result);
9 :   }
10:   print(result);
11: }
```

Figure 4: *Final version: use filesystem metadata to further reduce I/O operations.*

and filesystem-specific details by exporting the following functions:

- `init()` initializes UNIC.
- `get_file_hash(file)` returns the hash of a file, where `file` can be the name of a file, a file descriptor, or an inode number. If the underlying filesystem has deduplication enabled (*e.g.*, ZFS), it gets the hash of the file from the filesystem metadata without reading the file content. Otherwise, it computes the hash from the file content using libcrypto.
- `get_block_hash(file, block)` is similar as above, but returns the hash of a block of a file, where `block` specifies the block number. This is particularly useful if the application's computation is based on blocks, such as a `bzip2` compression. The application should decide whether to use `get_file_hash()` or `get_block_hash()` based on its own logic, which is discussed in §4.
- `exists(computation, hash, id)` checks if a given computation and input exists in the result cache. The parameter `hash` is the hash of input data. The parameter `id` is an optional string identifier defined by the application, used for differentiating multiple computations performed on the same input. For example, the virus scanning application may let `id` be the signature-scanning options.
- `get(computation, hash, id)` gets the result of a given computation and input from the result cache.
- `put(computation, hash, id, result, ttl)` puts an entry of computation, input, and result into the result cache. An optional `ttl` specifies its time-to-live in seconds, and the result cache automatically deletes the entry upon expiration.

4 Leveraging Storage Deduplication

UNIC explores a cross-layer design allowing underlying storage system to expose data deduplication information to the applications.

Typically, a deduplication-enabled filesystem maintains the hash of each file as its metadata. Since UNIC also uses hash to identify the memoization input, it is both convenient and efficient to leverage such filesystem metadata. Therefore, when an application needs to get a hash, UNIC automatically detects the underlying storage system type, and returns the hash directly from the metadata if the filesystem has enabled deduplication. If not, UNIC reads the file content and computes the hash itself. In this way, UNIC provides a consolidated interface for both scenarios, making the storage system details transparent to the applications.

Furthermore, the application does not need to know whether the underlying storage system is file-level or block-level deduplicated. It should decide whether to use `get_file_hash()` or `get_block_hash()` solely based on the application's own logic. Generally, if the application's computation works with the file on a block-by-block basis, such as the `bzip2` compression algorithm, it should use `get_block_hash()`. Otherwise, if the application's computation uses the file as a whole or randomly accesses the file, such as an anti-virus program, it should use `get_file_hash()`.

5 Implementation

We now describe UNIC's components and implementation details.

5.1 UNIC Components

Figure 5 shows the architecture of UNIC. It is deployed on a network of multiple hosts. Each user can log into multiple hosts, and each host can have many users logged in. Because of UNIC's security design (§2), different users do not need to mutually trust each other.

The UNIC module on each host handles application's memoization requests. Since memoization works best when the reuses of computations are frequent, reading data from the result cache should be more common than writing data to it. In light of this, we design UNIC to make read operations as fast as possible. A trusted master cache server handles all write operations. It can be either standalone or co-located with the enterprise's storage (*e.g.*, NFS) server. Each host has an optional read-only slave cache, which periodically syncs from the master cache server. If the slave cache is present, all read operations happen locally. For security, all network communications are encrypted with SSL/TLS. To reduce the handshake latency, the UNIC module on each host establishes a connection with the master cache server when the host boots up, and keeps the connection alive.

Because data updates on the slave caches happen asynchronously, it is possible that a host does not have the latest cached results. However, we point out that memoized computations are deterministic (§2.1), therefore the consistency on the slave caches should not affect the integrity of computations. The only contingency would be that an application may not be able to leverage recently cached results but have to compute on its own.

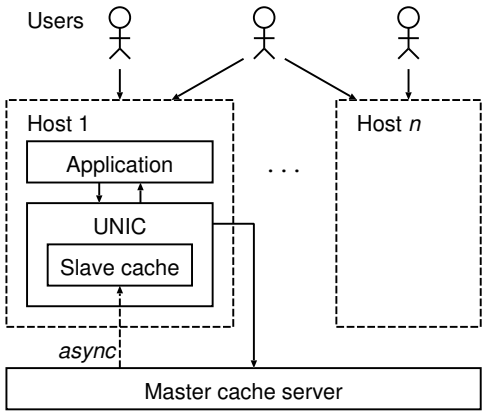


Figure 5: UNIC *architecture.* Additional hosts each have the same architecture as Host 1, and are omitted here due to limited space.

UNIC inserts a kernel module into the Linux kernel as a virtual device for computing hash($code$) and sig. It represents $code$ by the image of the executable process, with all libraries statically linked. The secret key K is inaccessible to the user space. The user-space application talks to the kernel module via `ioctl`. For improved performance, the kernel module internally caches hash($code$) for each caller.

UNIC uses a modified Redis key-value store [27] as the result cache. It modifies Redis to support UNIC's protocol (§2.3), and removes nonessential functions (such as `KEYS` which can list all cache entries) from Redis for security. Therefore, users cannot access the result cache except through UNIC.

5.2 Opportunistic Memoization

When using UNIC, the application developer needs to judge the best opportunity to use memoization because of two reasons. First, memoizing an already-fast computation may not justify the overhead of accessing the result cache. Second, abusing memoization for low-redundancy computations could result in exceeded overhead for entries that are never reused later. However, making the optimal decision at compile time is usually hard because input data cannot be predicted. Therefore, UNIC provides an optimization to opportunistically enable memoization only when the computation is slow and its reuse happens to be frequent at runtime.

To do so, UNIC internally has a model of T_{put}(`result_size`) and T_{get}(`result_size`), meaning

how long it would take to put and get a certain size of result, respectively. This model is independent of the actual content of the result, and it can be learned from a microbenchmark upon the installation of UNIC (see §6.2.1 for our evaluation). UNIC also maintains an accumulator t_{save} for each computation, initialized to 0, for the total time that could have been saved for the future.

UNIC further provides two functions for an application to mark the boundary of a computation. An application calls `begin()` to indicate that a computation starts, and UNIC records the current timestamp as t_{begin}. An application calls `end()` to indicate that the computation has finished, and UNIC records the current timestamp as t_{end}. When `put()` is called, UNIC does not put the data into the result cache immediately, but updates t_{save} to be

$$t_{save} = t_{save} + t_{end} - t_{begin} - T_{get}(\texttt{result_size})$$

Therefore, the slower and the more frequent a computation is, the larger t_{save} becomes. UNIC only performs the `put()` operation when t_{save} is greater than $T_{put}(\texttt{result_size})$, *i.e.*, the time that could have been saved from a computation is greater than the time that would be spent for memoizing the computation. In the case that $t_{save} < T_{put}(\texttt{result_size})$, UNIC ignores the `put()` request, and simply updates t_{save}.

6 Evaluation

We evaluated UNIC on a workstation with an Intel Core i7-2600 CPU and 32GB RAM, running Fedora 20 with Linux 3.16.2. The cache server was running Redis 2.6.17. Our goal is to show that UNIC significantly improves performance with memoization while requiring minimal developers' effort and storage space.

The rest of this section focuses on three questions:
§6.1 Is UNIC easy to use?
§6.2 Does UNIC reduce computation time?
§6.3 What is UNIC's storage overhead?

6.1 Application Adaptation Effort

To evaluate whether UNIC is easy to use, we picked four popular open-source applications that we use daily: (1) `clamav`-0.98.1, an anti-virus software that scans a directory for viruses [10]; (2) `pbzip2`-1.1.8, a multi-threaded compression utility that compresses a single file [25]; (3) `grep`-2.18, a tool that searches for a regular expression within one or many files; and (4) the compiler `gcc`-4.8.3. We adapted them to use UNIC's API[2]. We used file-level memoization for `grep`, `clamav`, and `gcc`, and block-level memoization for `grep` and `pbzip2`.

[2]Our adaptation of `gcc` is based on `ccache` [9].

Application	Total LoC	Changes	Percentage
`clamav` (file)	1,732,762	12	<0.01%
`pbzip2` (block)	4,376	18	0.41%
`grep` (file)	9,658	35	0.36%
`grep` (block)	9,658	69	0.71%
`gcc` (file)	29,023	30	0.10%

Table 1: *Lines of code changed for each application.* Parenthesis indicates whether the adaptation uses file-level or block-level memoization. The numbers for `gcc` are based on `ccache`.

Table 1 shows the lines of changed code for each application to use UNIC's APIs. Changing dozens of lines (<1% of total lines) suffices for all these applications.

To further illustrate, we next present how we adapted `grep`, the application with the most code changes.

6.1.1 Case Study: `grep`

GNU `grep` is a line-based pattern searching utility. To invoke `grep`, the user specifies a search pattern and the path to a file or directory. Then `grep` iterates through all files in the directory and search for the pattern.

Common to all applications, the first step is to add a call to `init()` at the beginning of `main()` in order to initialize UNIC. For `grep` specifically, there are two design choices: we can memoize either at file-level or at block-level. Memoizing at file-level is faster when the whole file is unchanged, whereas memoizing at block-level can exploit sub-file similarities for different files. Next we discuss each of them.

File-level Memoization. Adapting `grep` for file-level memoization is relatively straightforward. When `grep` works on a new file, we call `get_file_hash()` to get the hash of the file from ZFS and call `exists()` to check if there is a corresponding entry in the result cache. If so, we call `get()` to retrieve the memoized result, output it, and move on to the next file. If not, we follow the original algorithm and call `put()` to memoize whatever is output. We also call `put()` to memoize the number of matched lines in the current file, which `grep` uses for internal bookkeeping purposes.

Block-level Memoization. Adapting `grep` to memoize at block-level requires tighter integration with its workflow. For each file, `grep` reads its content in 32KB chunks, and performs pattern searching one chunk at a time. However, since the searching is line-based (delimited by '\n'), it is possible that lines are not well-aligned with chunk boundaries. For example, one line may span across the end of the previous chunk and continue at the following chunk. In this case, `grep` adjusts its chunk boundary to include the residue of the line in the previous chunk and exclude the partial line at the end of current chunk, as shown in the shaded region in Figure 6.

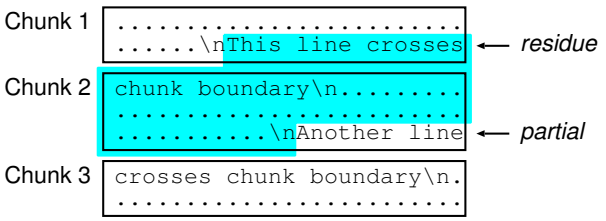


Figure 6: *Misalignment between line and chunk boundaries in* `grep`. Shaded region is the adjusted chunk for computation.

Unfortunately, this poses a challenge to using UNIC directly, because ZFS keeps hash metadata only for entire aligned 32KB disk blocks. On the other hand, we cannot simply use the hash of the unadjusted chunk to address the cache, because this would err if two chunks were the same but their residues in the previous chunk differed. Our solution is to combine the hash of all chunks from the beginning of the residue until the current chunk. Note that this may lose the rare opportunity of reusing memoized results for chunks who only differ at the last partial line, but it preserves correctness nevertheless.

Our experience with adapting the other three applications were straightforward. Overall, we found UNIC easy to use and the adaptation effort was generally little.

6.2 Performance

To understand the performance of UNIC, we first use microbenchmarks to evaluate the throughput of UNIC's basic operations. We then run UNIC on four real-world applications to see how UNIC reduces application running time. Next, we study how UNIC is able to reuse previous computation results for some evolving data. Finally, we study how UNIC performs with a group of multiple users whose data are similar yet different.

6.2.1 Microbenchmark

We first use microbenchmarks to evaluate the throughput of the `get()` and `put()` operations. We wrote a program that calls `put()` 10,000 times followed by calling `get()` 10,000 times. The hashes of the 10,000 entries are all different, and we varied the result size from 1KB to 1MB.

Figure 7 shows the results, where each data point is an average of 10 individual experiments with an error bar showing the maximum and minimum value in the 10 experiments. The x-axis is the size of the memoized result. The y-axis is the total time in performing the 10,000 operations. The solid line is for `put()` and the dashed line is for `put()`. From the results we find that the time for an operation is on the order of ten microseconds when the memoized result is small in size ($<$10KB), which is mostly the case (see §6.3). Even if the memoized result is as large as 1MB, the time to get a memoized entry is only

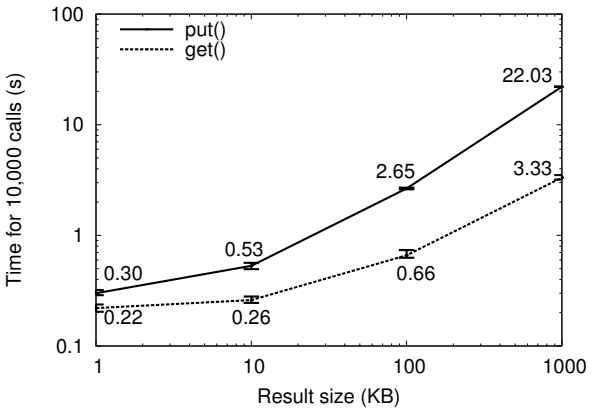


Figure 7: *Throughput of* `put()` *and* `get()` *operations.* The x-axis is the size of memoized result. The y-axis is the total time in performing 10,000 `put()` (solid line) and `get()` (dashed line) operations.

0.33ms, which is normally much faster than doing real computation on that size of data. Therefore, UNIC's basic operations are sufficiently fast for doing useful caching of computations.

6.2.2 Application Performance

We next show how real-world applications benefit from UNIC, and how storage deduplication further helps. We conducted the following experiments. (1) We used `clamav` to scan for viruses on two data sets. The first is the linux-3.12 kernel source code tree. The second is the Dropbox folder for one of the co-authors, which contains 10.8GB of documents, music, pictures, videos, and applications. (2) We used `pbzip2` to compress `linux-3.12.tar` into `linux-3.12.tar.bz2`. (3) We ran `grep` on two data sets. The first is the linux-3.12 kernel source code tree, which consists of 47,336 small files totaling 508MB. The second is the tags file of the linux-3.12 kernel source code generated by `ctags -R`, which is a single text file of 250MB. For each data set, we ran a simple query ('`void`') and a complex query ('`^\s*struct\s+\w+\s+\**\s*\w+\s*=\s*\w+\((\w+(,)*)+\);`' for the source code tree, which matches declaring and initializing a structure pointer to the return value of a function, such as "`struct task_struct *task = get_proc_task(inode);`", and '`/[A-Za-z]+\.c.*d.*file`' for the tags file, which matches a specific type of tag). (4) We used `gcc` to compile linux-3.12 kernel with the `allnoconfig` configuration. Because `gcc` has a nontrivial way to represent input dependencies for cache reusability rather than a file hash, our adaptation does not leverage storage deduplication information. All data files are on a freshly-formatted ZFS disk with cold buffer cache.

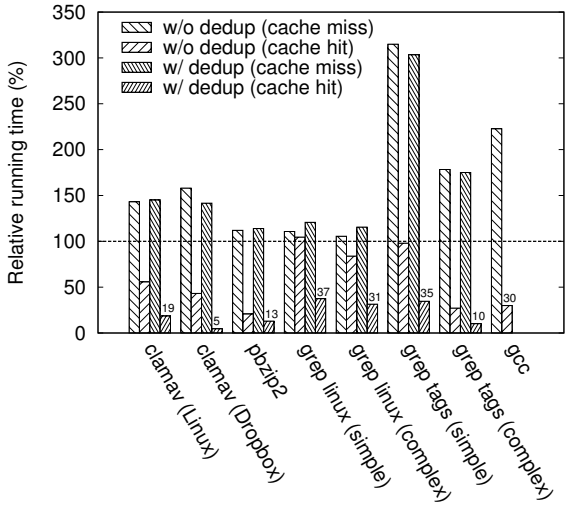


Figure 8: *Relative running time of applications.* The y-axis is the running time relative to the original application. For each cluster, the first bar is cache-miss execution without FS deduplication, the second bar is cache-hit execution without FS deduplication, the third bar is cache-miss execution with FS deduplication, and the fourth bar is cache-hit execution with FS deduplication. The dashed line at 100% shows the running time for the original application.

For each application, we compared the running time (1) without UNIC (the baseline), (2) with UNIC but without filesystem deduplication (the first and second bars on Figure 8), and (3) with both UNIC and filesystem deduplication support (the third and fourth bars). For experiments with UNIC, we further compared the running time (1) for execution on an initially empty result cache, causing cache misses and thus putting entries to the cache (the first and third bars), and (2) for execution when the result cache had already been pre-populated, causing cache hits (the second and fourth bars).

Figure 8 shows the running time for each experiment. Each number is an average of 10 individual runs. Although running applications on an empty result cache incurs an average overhead of 68.2%, running them on a warm result cache gives an average speedup of 2.39×. If filesystem deduplication is available, the average overhead of cache-miss execution drops to 59.3% and the average speedup with memoization increases to 7.58×. Furthermore, complex computations (*e.g.*, scanning for viruses or compressing a file) benefit the most from memoization (up to 21.4× speedup), while simple computations (*e.g.*, searching for a short string) suffer more from the cache-miss overhead. Therefore, opportunistically enabling memoization would be the best practice. With our strategy described in §5.2, memoization is enabled at the second occurrence of `put()` for one application ("`grep` tags" with simple query), and at the first occurrence for all other applications.

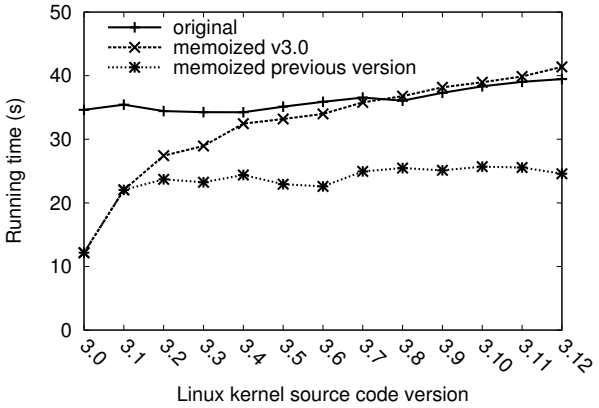


Figure 9: *Effectiveness of memoization with evolving data.* Solid line is the original `grep` without memoization. Dashed line has the result cache populated with v3.0. Dotted line has the result cache populated with the immediate previous version.

6.2.3 Effectiveness with Evolving Data

The previous evaluation focused on the memoization benefit on exactly the same computation. Next we show the effectiveness of memoization if the input data is evolving, *i.e.*, if UNIC has memoized computation on an old version of data, how it can speed up computation on a new version of the data.

We used `grep` to search for '`void`' on thirteen major versions of the Linux kernel source code, from v3.0 to v3.12. All files are on a freshly-formatted deduplication-enabled ZFS disk with cold buffer cache. We performed three sets of experiments. The first one used the original `grep` without UNIC. In the second experiment, we first populated the result cache when running `grep` on v3.0, and then measured the time for running `grep` on each version based on the same memoization of v3.0. In the third experiment, we ran `grep` on each version in a "rolling" manner, *i.e.*, each execution was based on the memoization of the immediate previous version, which resembles a more practical scenario.

Figure 9 shows the running time for all executions, where each number is an average of 10 runs. With a single memoization of v3.0, the speedup is significant for running on v3.1 (1.61×), but diminishes along the increment of version number, and eventually becomes ineffective after v3.8, because the source code differs significantly from the memoized version and the cache hit rate drops below 0.3. On the other hand, when memoized the immediate previous version, the speedup is almost constant, with an average of 1.50×. The reason is that the amount of source code difference is almost constant between each two consecutive versions, and many mem-

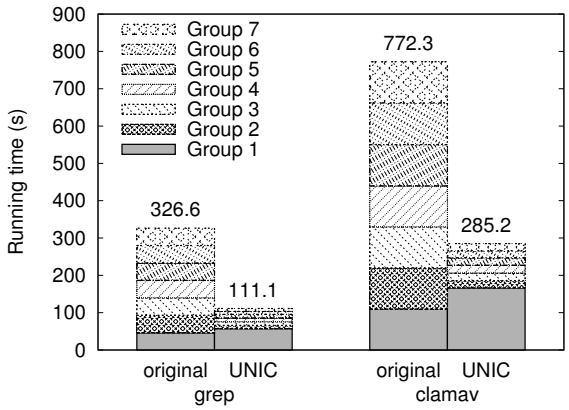


Figure 10: *Effectiveness of memoization across users.* For each cluster, the first bar is the original application, and the second bar is the application modified to use UNIC. Each bar shows the breakdown of running time on each group, the number on top showing the total time.

oized results can be reused (hit rates are between 0.73 and 0.81). Therefore, UNIC is more effective when the divergence of the actual input data from the memoized data is small, which is likely true in a practical scenario.

6.2.4 Effectiveness with Multiple Users

We next evaluate the memoization effectiveness for multiple users with similar yet different data. We took the project directories of seven groups of students in a graduate-level operating system course offered by our university. The average size of each directory is 1.6GB. We performed two executions on each group's directory: (1) use `grep` to search for '`void`', and (2) use `clamav` to scan for viruses. This resembles the enterprise setting where multiple people working on the same project have similar data and perform common computing tasks such as virus scanning. The result cache was originally empty, and was gradually filled by UNIC during the process.

Figure 10 shows the breakdown of each application's running time on each group. The trend is that the original application takes almost the same amount of time for all groups. With UNIC, although the first group takes longer time to execute (24.1% for `grep` and 51.9% for `clamav`), all subsequent groups consistently take a much shorter time (5.17× speedup for `grep` and 5.57× speedup for `clamav`). This is because for the first group, all computations are new and UNIC needs to insert them to the result cache. Once this is done, all subsequent groups can benefit from it. The overall speedups for the executions on all seven groups are 2.94× for `grep` and 2.71× for `clamav`. We foresee that with more number of groups the overall speedup should be even higher. Therefore, UNIC is practical for a group of users working together or doing similar tasks.

6.3 Storage Space

We now evaluate the storage overhead of UNIC. For each application we used for the performance evaluation in §6.2.2, we examined the number of entries in the result cache. To study the total space used for memoization, we also let Redis dump a snapshot of all data and measured the size of the dump file.

Table 2 shows the results. Column (a) is the number of input files. Column (b) is the total size of input files. Column (c) is the number of entries in the result cache. Column (d) is the size of the Redis dump file. The relative storage overhead is thereby Column (d) divided by Column (b), which is shown in Column (e). The results depict that the average overhead of the memoization storage for all applications is 3.45%, negligible compared with the storage of all file data. Therefore, UNIC incurs little storage overhead.

7 Discussion and Limitations

We discuss UNIC's security implications and limitations.

Denial-of-service attacks. A malicious user may issue a large number of put requests on manufactured inputs, and pollute the result cache with useless results. Several approaches can be used to defend against it. For example, UNIC may rate-limit puts to the result cache, employ a quota mechanism to limit the cache space for each client application, or enforce time-to-live limits on cached results. We argue that even if the result cache is full, the worst outcome would be that future computations cannot be memoized and have to be recomputed, yet the secrecy and integrity of computations are not violated.

Side-channel information leakage. A malicious user may enumerate through a large set of inputs on an application, and observe if some executions are significantly faster than others. Based on the observed timings, she may infer what computations have been done by other users and what have not. While defending against this side-channel attack is out of the scope of this paper, we note that the application developers may defend against it by rate-limiting queries to the result cache or randomly forcing cache misses even if the result exists in the cache.

Brute-force attacks. A malicious user may enumerate through all possible hash values of the application code and input, in hopes of getting cached results. We argue that the possibility for an unprivileged user to get a valid hash is minimal. Even if she manages to get an entry, she only knows the result, but she cannot generate the original code and input from the hash. In the example of virus scanning, she might brute-force a hash and discover the result of scanning some file, but she cannot determine the original content of that file. Again, UNIC

Application	(a) File count	(b) File size	(c) Entry count	(d) Dump size	(e) Overhead
`clamav` (Linux)	47,336	508.1MB	44,277	2.8MB	0.55%
`clamav` (Dropbox)	2,792	10.8GB	82,061	4.4MB	0.04%
`pbzip2`	1	544.0MB	4,151	106.4MB	19.55%
`grep` linux (simple)	47,336	508.1MB	70631	11.2MB	2.21%
`grep` linux (complex)	47,336	508.1MB	51532	4.2MB	0.83%
`grep` tags (simple)	1	250.0MB	2	5.3MB	2.13%
`grep` tags (complex)	1	250.0MB	2	4.5MB	1.80%
`gcc`[3]	47,336	508.1MB	522	2.3MB	0.46%

Table 2: *Storage overhead.* Columns are: (a) the number of input files, (b) total size of input files, (c) number of entries in the result cache, (d) size of the Redis dump file, and (e) relative storage overhead.

may defend against this attack by rate-limiting queries to the result cache. Furthermore, if the result is sensitive by itself (*e.g.*, `cat`), the application developer may encrypt it before putting it to the result cache, or the system administrator may disable UNIC for such applications.

Application bugs. Ensuring bug-free code is a hard problem orthogonal to UNIC and code attestation. If the application contains a bug such as buffer overflow, a malicious user may exploit the bug to poison the result cache. Existing systems such as baggy bounds checking [1] and AddressSanitizer [28] can prevent many memory access bugs. Other countermeasures include letting the application rerun the computation and verify the cached result periodically, and purging the result cache when a bug is found. In addition, using hardware-enforced isolation mechanisms such as Intel TXT [18] with TPM, or Intel SGX [5, 17] may avoid this issue.

8 Related Work

Storage deduplication. Storage deduplication reduces data redundancy at either file-level [22] or block-level [12, 32]. ZFS [36] is a widely used cross-platform filesystem that does block deduplication at the time data is written. These works are orthogonal to UNIC, and UNIC's cross-layer design allows it to transparently leverage storage deduplication information.

Ad-hoc caching. Many applications use ad-hoc caching to improve performance, but they either trust all users, or simply disallow cross-user caching. For example, `ccache` [9] caches compiler outputs on the local filesystem, but the cache can be easily exploited or poisoned by any user. On the other hand, `clamav` [10] only caches virus scanning results within a single session, rendering cross-session and cross-user caching impossible. UNIC improves the status quo with strong security guarantees.

Memoization. Memoization [19, 23, 26] is a technique that reuses prior computation results of functions without side effects. Vesta [16] uses memoization for software configuration management. Nectar [15] memoizes intermediate results from DryadLINQ [35] programs. Incoop [7] uses memoization to build a MapReduce framework for incremental computations. However, these systems handle only specific computations, and it is non-trivial to generalize their use cases. UNIC can be used to deduplicate general computations.

Code attestation. Many code attestation techniques exist to provide integrity of computations. For example, result-checking [33] verifies the result produced by a program by computing it in two ways. Secure boot mechanisms [3, 4] verify the integrity of the software stack after booting. BIND [30] ties the proof of what computation has been run to the result that the computation has produced. Pioneer [29] provides code integrity guarantees for running software on an untrusted system. UNIC makes novel use of the code attestation mechanism to protect the secrecy and integrity of memoization.

9 Conclusion

We presented UNIC, a general system for applications to securely deduplicate their rich computations. It uses code attestation mechanism to achieve both secrecy and integrity. It explores a cross-layer design that allows applications to leverage storage deduplication information for speed. Evaluation results show that UNIC is easy to use, speeds up applications by up to 21.4×, and incurs little storage overhead.

Acknowledgments

We thank Yinzhi Cao, Gang Hu, David Williams-King, Xi Wang (our shepherd), and the anonymous reviewers for their valuable comments. This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

[3] Not all files are used for compilation due to our experiment configuration.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, 2009.

[2] Amazon Web Services. Public data sets. `http://aws.amazon.com/datasets`.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, 1997.

[4] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated recovery in a secure bootstrap process, 1998.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 267–283, Oct. 2014.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[7] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[8] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, 2000.

[9] ccache. `http://ccache.samba.org/`.

[10] Clam AntiVirus. `http://www.clamav.net/`.

[11] CommonCrawl. `http://commoncrawl.org/`.

[12] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, 2009.

[13] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. White Paper 223310, Mar. 2011.

[14] Google Chrome OS. `http://www.google.com/chromeos/index.html`.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[16] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, 2000.

[17] Intel. Software guard extensions programming reference. `https://software.intel.com/sites/default/files/329298-001.pdf,`.

[18] Intel. Intel trusted execution technology: White paper. `http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html,`.

[19] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):546–585, May 1998.

[20] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, 1994.

[21] L. Mearian. World's data will grow by 50x in next decade, IDC study predicts. `http://www.computerworld.com/s/article/9217988/World_s_data_will_grow_by_50X_in_next_decade_IDC_study_predicts`.

[22] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14:1–14:20, Jan. 2012.

[23] D. MICHIE. "memo" functions and machine learning. *Nature*, 218:19–22, Apr. 1968.

[24] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[25] PBZIP2. Parallel BZIP2 (PBZIP2). http://compression.ca/pbzip2/, 2011.

[26] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, 1989.

[27] Redis. http://redis.io/.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX '12)*, 2012.

[29] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005.

[30] E. Shi, A. Perrig, and L. van Doorn. BIND: a fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168, May 2005.

[31] C. Soghoian. How Dropbox sacrifices user privacy for cost savings. http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html.

[32] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, 2010.

[33] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM (JACM)*, 44(6):826–849, Nov. 1997.

[34] Webopedia. Enterprise storage. http://www.webopedia.com/TERM/E/enterprise_storage.html.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level languag. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, Dec 2008.

[36] ZFS: the last word in file systems. http://www.sun.com/2004-0914/feature/.

Lamassu: Storage-Efficient Host-Side Encryption

Peter Shah and Won So
NetApp Inc.

Abstract

Many storage customers are adopting encryption solutions to protect critical data. Most existing encryption solutions sit in, or near, the application that is the source of critical data, upstream of the primary storage system. Placing encryption near the source ensures that data remains encrypted throughout the storage stack, making it easier to use untrusted storage, such as public clouds.

Unfortunately, such a strategy also prevents downstream storage systems from applying content-based features, such as deduplication, to the data. In this paper, we present *Lamassu*, an encryption solution that uses block-oriented, host-based, convergent encryption to secure data, while preserving storage-based data deduplication. Unlike past convergent encryption systems, which typically store encryption metadata in a dedicated store, our system transparently inserts its metadata into each file's data stream. This allows us to add Lamassu to an application stack without modifying either the client application or the storage controller.

In this paper, we lay out the architecture and security model used in our system, and present a new model for maintaining metadata consistency and data integrity in a convergent encryption environment. We also evaluate its storage efficiency and I/O performance by using a variety of microbenchmarks, showing that Lamassu provides excellent storage efficiency, while achieving I/O throughput on par with similar conventional encryption systems.

1 Introduction

Storage users are understandably sensitive to data security on shared storage systems. Adding encryption to an existing solution can help to address such concerns by preventing unauthorized parties from accessing the contents of critical data. One popular approach that seems to have quite a lot of traction is to encrypt data close to the application, or even inside an application itself. This strategy simplifies down-stream security by ensuring that data is in an encrypted state by default as it moves downstream through the stack. This strategy can take many forms, such as built-in application encryption, OS-based file system encryption or VM-level encryption [3, 19, 22]. We term any encryption that runs on the same physical hardware as the primary application *data-source encryption*.

In general, existing data-source encryption solutions interfere with content-driven data management features provided by storage systems — in particular, deduplication. If a storage controller does not have access to the keys used to secure data, it cannot compare the contents of encrypted data to determine which sections, if any, are duplicates.

In this paper, we present an alternative encryption strategy that provides the benefits of upstream encryption while preserving storage-based data deduplication on downstream storage. Based on these conflicting priorities, we name our system *Lamassu*, after the Assyrian guardian deity that combines elements of several creatures. Our system builds upon existing work in convergent encryption [10, 6, 18] to enable deduplication of encrypted data, but extends it to provide its services in a manner transparent to both application and storage, without the need for dedicated metadata storage or additional files. This makes our system flexible and portable, allowing it to be self-contained, and greatly simplifying deployment in existing application environments. Our work also introduces a scheme for providing crash-tolerant data consistency in a convergent system, as well as a mechanism for verifying the integrity of data after a crash.

Lamassu preserves deduplication at the storage back end by using a message-locked, convergent encryption strategy [10] to secure data in a way that preserves block-equality relationships in the ciphertext. In such a scheme, data is encrypted using keys that are derived from the plaintext, thus the message is *locked* under itself [5]. The actual cipher used to secure the data can be any standard encryption scheme, such as the Advanced Encryp-

tion Standard (AES). By using this approach, any two users who have access to the same plaintext will deterministically arrive at the same ciphertext before storing it on the back end. As a result, the storage system receiving that data will be able to identify and deduplicate redundant blocks, even though it is unable to decrypt them.

Convergent encryption provides strong security on data that has high min-entropy, where it is very difficult to guess at the contents of messages accurately. Unfortunately, real production data is often much less random than ideal data; there are often identifiable patterns that an outsider can exploit to guess at data contents with a much higher success rate than random guessing. As a result, this approach is vulnerable to the so-called *chosen-plaintext attack* [6, 20]. In this attack, an adversary takes advantage of the nonrandom nature of production data by guessing the data rather than the encryption key. If attackers guess correctly, they can generate the matching key and verify their guesses by generating ciphertext blocks that match the victim's blocks.

Other work in this field has explored alternative defenses against the chosen-plaintext attack. For example, DupLESS [6] provides a mechanism that uses a double-blind key generation scheme to allow an application host and a key server to cooperatively derive convergent keys. In the DupLESS scheme, the key server never sees the data to be encrypted, and the application host never has access to the secret keys stored on the key server. The disadvantage of that system is that each key generation operation requires multiple network round-trips between the application host and the key server, making it impractical for block-level operation.

We have chosen a relatively simple defense against the chosen-plaintext attack by adding in a secret key to derive the convergent key before using it for encryption [20]. This mechanism is similar to the domain key used to derive keys in DupLESS, but in Lamassu, clients are permitted direct access to the secret key and generate their convergent keys locally. With this mechanism, an attacker executing a chosen-plaintext attack needs to guess both the contents of the plaintext and the secret key in order to generate a matching convergent key, and to succeed.

Lamassu instances that use different secret keys will produce different ciphertext from the same plaintext, and data across those instances will not be deduplicated. On the other hand, if two (or more) clients share a single secret key, they can all read and write data to a shared storage system through Lamassu, and their shared data can be deduplicated by that system. In effect, a set of clients that share a single secret constitute both a security zone and a deduplication group. We collectively term a group of tenants that share a key an *isolation zone*. The details of how this shared secret is implemented is discussed in further detail in §2.

In order to retrieve Lamassu-encrypted data from storage, a user must have access to the encryption keys used to secure that data. Because message-locked encryption produces keys based on plaintext, it produces a large number of keys that must be fetched along with the ciphertext in order to retrieve data. This unbounded mass of keys presents a *metadata management problem* that is intrinsic to a message-locked encryption strategy.

Past solutions have managed this cryptographic metadata by storing keys alongside the encrypted file data [10, 6], or by building a dedicated metadata store that stores the keys separately from the primary data [18]. In both cases, the cryptographic metadata itself must be secured, usually by means of either symmetric or asymmetric key encryption. Such solutions complicate the process of replicating or migrating encrypted data, because the separated key information must be managed in parallel. For cases in which the cryptographic metadata is kept in a dedicated store, that functionality must also be replicated wherever the data is to be housed. Providing full replication or migration capabilities may require either modification to the underlying storage controller's facilities or the addition of external tools to provide those capabilities outside of the controller.

In contrast, Lamassu implicitly inserts the cryptographic metadata generated by encryption into the data stream for each file. In order to avoid polluting the file's data blocks with highly entropic key information, thus hindering deduplication, Lamassu places this data into reserved sections of the file. In effect, a predetermined fraction of the blocks stored at the storage controller will be devoted to encrypted metadata, rather than file data. These encrypted metadata blocks are indistinguishable from random data, and will not be deduplicated by the storage controller.

Contributions. To the best of our knowledge, Lamassu is the first system that achieves the following: First, it provides strong data-source encryption that preserves storage-based block deduplication, without requiring modifications to the application or storage, and without requiring a dedicated metadata store to manage convergent keys; second, by embedding cryptographic metadata inside encrypted files, Lamassu allows both data and metadata to be automatically managed by existing tools and storage features; and third, our metadata structure provides a mechanism for maintaining consistency between a file's data and its cryptographic metadata. In order to accomplish those goals, Lamassu uses these key techniques: block-oriented convergent encryption, insertion of encryption metadata to the data stream, efficient metadata layout and multiphase commit algorithm, and a built-in data integrity check mechanism.

The rest of this paper is organized as follows. In §2, we

will lay out the design of our system, including our threat model, detailed encryption strategy, and metadata layout. We will also describe our consistency and integrity model. §3 provides details on our prototype implementation, followed by §4 which shows our experimental results. Finally, we discuss related work and conclude in §5 and §6, respectively.

2 Design

2.1 Threat Model

Our threat model is informed by the ones used by past secure deduplication work [10, 6], and by our own analysis of the level of security that could make sense in an enterprise environment. Our threat model takes the form of a series of explicit assumptions about the capabilities of a potential attacker, and of the hardware and software available in our expected deployment environment, as follows:

- We assume that the basic cryptographic primitives such as AES that we use, represent an "ideal" encryption function and cannot be broken by attackers. We further assume that any potential attacker is aware of the encryption mechanism that we have chosen and how it works.

- We assume that data will be stored on an untrusted, shared storage system. We further assume that the shared system may behave as an *honest-but-curious attacker* [11], attempting to read stored data, but not acting to maliciously destroy data. An example of such an environment might be a public cloud storage system, which can reasonably be expected to preserve stored data, but which must be prevented from viewing data contents.

- We assume that the storage system will have full access to all data that it stores, but that it will not have any prior knowledge of the contents of encrypted data, or of the keys used to encrypt that data.

- We assume that the storage system stores data from multiple tenants, and that those tenants might not trust each other. We assume that these tenants might gain access to any data stored on the storage system, including access to data blocks that they are not authorized to access, such as through improperly applied access control.

- We assume that the data-source systems that belong to a single trust domain may share secret information through some mechanism, such as a key server and KMIP (Key Management Interoperability Protocol).

Convergent encryption, applied upstream of the storage system, effectively prevents that system from reading the contents of the data. We assume that an attacker cannot compromise the key manager shared by clients to gain access to their shared master keys. If that happens, the attacker can effectively read the data stored by clients sharing that trust domain. Note that it would be feasible to adapt our system to use a double-blind key generation system that protects against that sort of attack, such as that described by Bellare et al. [6] at the cost of reduced I/O performance. However, we have not pursued this option due to the large performance overhead involved.

The work presented here focuses on protecting the contents of user data from an outside attacker, while preserving deduplication, but does not include protection for directory structure information. It should be possible to improve on this limitation by adding encryption for file and directory names in a future revision.

2.2 Encryption

The term *convergent encryption* describes any encryption scheme that preserves the following property: Given a particular plaintext, it will always generate the same ciphertext. In every other respect, a convergent encryption scheme should share the same properties as standard encryption schemes. Lamassu exploits this property to enable deduplication of encrypted data by ensuring that identical plaintext blocks are stored as identical ciphertext blocks. This means that Lamassu exposes information about block equality to any potential outside observer, but does not expose any additional information about the data. Existing work on convergent encryption strategies discusses the cryptographic security of this approach [10, 6]. In general, larger block sizes reduce the granularity of information exposed to a potential attacker, and reduce the amount of information that can be gleaned from the pattern of blocks stored on disk.

Lamassu uses a two-tier encryption strategy, laid out in Figure 1. The first tier is the convergent encryption applied to the application data written to each file. To protect against the chosen-plaintext attack, described previously, Lamassu uses a secret key in the process of deriving each convergent key. The second tier is standard (nonconvergent) encryption applied to the cryptographic metadata stored inside each file by using a second secret key. Data encrypted by separate Lamassu instances can be read or written by either instance, provided that those two instances share both of these secret keys.

The first of the two secret keys used by Lamassu is an inner key (K_{in}), used when encrypting file data blocks. When Lamassu writes a block to storage, it starts by taking a cryptographic hash[1] (H) of the data block in

[1] Our current prototype is using SHA-256 in order to generate 32-byte hashes from fixed-size data blocks.

memory. The convergent encryption key for that block (*CEKey*) is derived from the hash value and the inner key by the following equation:

$$CEKey_i = F(H(Block_i), K_{in}) \quad (1)$$

where F represents a *key derivation function (KDF)*. In our current implementation, this is accomplished by AES-encryption of the block hash using the inner key, but other key derivation functions could also work. This modified *CEKey* is used to encrypt the actual data block before it is sent to disk as shown in the following equation:

$$CipherBlock_i = E_{AES}(Block_i, CEKey_i, IV_{fixed}) \quad (2)$$

where E_{AES} represents the AES encryption function. For data block encryption, Lamassu uses AES-256 in CBC mode. As with previous convergent encryption systems, Lamassu uses a fixed initialization vector[2] (IV_{fixed}) for this process, so that future encryption of the same data will result in identical ciphertext [10]. The block key is stored inside the file so that it can be easily retrieved when reading the file, allowing Lamassu to decrypt the data block and recover its contents. The block key is stored in a reserved metadata section of the encrypted file. Further details will be described in §2.3.

Because the convergent keys are derived with key derivation function that uses a secret inner key (K_{in}), it is extremely unlikely that at encrypted block will match any data encrypted with the same technique, but using a different inner key. This property allows the inner key to be used to define a deduplication domain for data by restricting deduplication to just the data encrypted with the same inner key. In addition to preventing unauthorized parties from decrypting stored data, this also prevents an attacker from learning anything about secured data through the behavior of deduplication on multi-tenant, shared storage. This property allows tenants to define their own security isolation zone through the use of secret keys that are kept outside of the shared storage system.

The second shared secret used by Lamassu is an outer key (K_{out}) that is used to secure the metadata stored inside specially reserved sections of the file, including the per-block keys described previously. Lamassu encrypts the metadata blocks by using the AES in Galois/Counter mode (GCM), rather than in CBC mode as when encrypting data blocks. Lamassu also seeds its metadata block encryption with a randomly generated initialization vector (IV_{rand}) like conventional encryption systems, as shown in the following equation:

$$CipherMeta_i = E_{AES}(Meta_i, K_{out}, IV_{rand}) \quad (3)$$

[2]Standard AES-CBC takes a randomly generated initialization vector (IV) as well as a key as inputs. Convergent encryption uses an invariant IV to preserve data equality in the ciphertext [10].

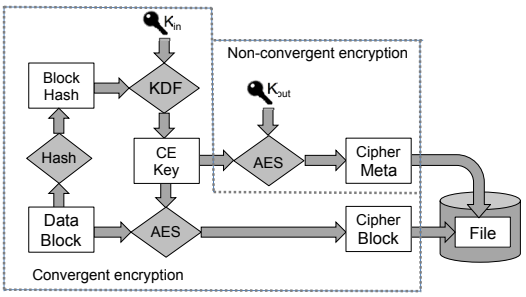


Figure 1: Lamassu's two-tier encryption model

where $Meta_i$ denotes a metadata block. A message authentication code (or tag) generated from AES-GCM will be added to each metadata block and used for anintegrity check. (The details will be discussed in §2.4.)

In order to read any of the data in the file, a Lamassu instance must have access to the outer key, thus defining a trust domain based on access to this key. Note that the outer key does not affect the boundaries of data deduplication, only data access. It would be possible to broadly share the inner key among many clients, while giving each one a separate outer key. The result would allow all of those clients to share a single deduplication isolation zone, while restricting them to reading and writing only their own private data. However, note that cryptographic security among those clients would be exactly equal to that of basic convergent encryption. They would no longer have any protection against chosen-plaintext attacks executed by their peers.

The inner and outer keys dictate how Lamassu would approach periodic key rotation. Our experimental system does not include a mechanism to re-key Lamassu files, but it would be possible to approach the problem by rotating the secret keys stored in the key server. Key rotation would have to be initiated by a higher layer in the application with the ability to update the key server and to identify which files or directories need to be reencrypted with the new keys. An interesting side effect of Lamassu's encryption model is that it is possible to perform a less secure, but much faster partial re-keying of Lamassu data by changing the outer key, but not the inner key. In that case, only the metadata blocks in each file would need to be re-keyed, rather than entire files.

2.3 Metadata Layout

Lamassu's convergent encryption strategy operates on a per-block basis. This means that the base unit for any read or write is a full block. It is not possible for Lamassu to update a piece of a block without fully reencrypting the whole block with its new data. Furthermore, any change to a data block must be accompanied by a corresponding update to the hash key for that block in the metadata section of the file. We will discuss our strategy

for maintaining consistency between data and metadata in §2.4.

Lamassu embeds its extra metadata into a file's data stream. This metadata is highly entropic by nature, and is extremely unlikely to result in any identifiable redundant sections for the storage controller to deduplicate. Because our metadata is produced in 32-byte sections rather than in full block-sized chunks, writing the data into arbitrary sections of the file can pollute potentially deduplicable chunks, preventing them from matching other, similar blocks. It can also interfere with block-alignment throughout the file, making it harder for a fixed-block deduplication to work.

Our solution is to place metadata in reserved sections of each file, segregating cryptographic metadata from encrypted primary data completely. These sections are designed to align with the underlying file system's block size so that they do not alter the block-alignment of any primary data. Our system is designed so that the chosen block size is easily variable. The chosen block size for our tests is 4096 bytes, with matching, aligned, 4096-byte reserved metadata sections inserted into the stream. This arrangement favors files that are at least a few megabytes in size, because this pre-allocation of space magnifies the space overhead of our solution in very small files. A smaller block size reduces the relative penalty for smaller files, but slightly increases the overall metadata space overhead for each file.

Because a file's size may be very large, and, more importantly, may change over time, Lamassu does not preallocate space for all of a file's metadata in advance. Instead, Lamassu distributes metadata blocks at regular intervals throughout the file, adding more as necessary. For simplicity, these blocks are placed in regular, predictable locations within the file, rather than in dynamically selected positions. Furthermore, each metadata block is placed in a position adjacent to the data blocks whose encryption keys it contains. We refer to a section of a file containing a single metadata block and all of the data blocks associated with it as a *segment*.

Figure 2 shows the internal layout of a Lamassu file, based on a 4KB block size, with the file further broken up into smaller segments and blocks. The size of each segment is defined by the number of 32-byte encryption keys that can be stored inside a single metadata block. That number is affected by several factors, including the amount of space occupied by additional metadata information, and on tunable factors that are outlined in §2.4.

Inside each metadata block, the first 48 bytes of space are used for general file metadata, rather than for encryption keys. Figure 3 shows the contents of this metadata space, which includes the random initialization vector (IV) used to encrypt the remainder of that block, a message authentication tag generated by AES-GCM, and the

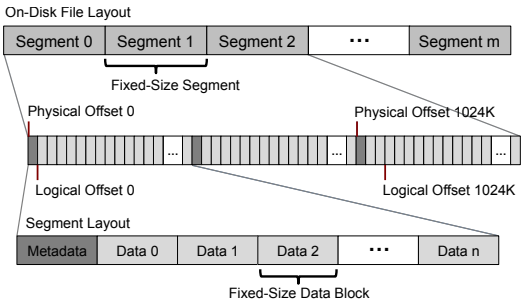


Figure 2: Internal layout of a Lamassu file



Figure 3: Internal layout of a Lamassu metadata block

logical size of the file's contents. The remaining space in each metadata block is taken up by a table of 32-byte encryption keys.

The size of each metadata block's key table determines the number of data blocks that can follow a single metadata block, and, by extension the size of a segment. Because each segment carries a mandatory one-block penalty for storing the metadata block, the most space-efficient arrangement is to maximize the size of each segment by filling as much space as possible with encryption keys. Lamassu trades away some of that space efficiency for better crash consistency, as will be discussed in §2.4. When the size of plaintext data is n bytes and each metadata block can store up to $NumKeys_{MB}$ keys, the number of data blocks (N_{DB}) and metadata blocks (N_{MB}), the size of the encrypted file (n'), and the space overhead of Lamassu can be formulated with the following set of equations:

$$N_{DB} = \lceil n/BlockSize \rceil \quad (4)$$

$$N_{MB} = \lceil N_{DB}/NumKeys_{MB} \rceil \quad (5)$$

$$n' = (N_{DB} + N_{MB}) \cdot BlockSize \quad (6)$$

$$Overhead = n' - n \quad (7)$$

The space overhead is minimized when n is exactly a multiple of the Lamassu block size and the last metadata block has no empty key table slot, as shown in the following equation:

$$Overhead_{min} = n/NumKeys_{MB} \quad (8)$$

As previously mentioned, the metadata stored at the beginning of each metadata block includes a logical file size. The reason for this is that Lamassu always encrypts data in full block-sized chunks, and, therefore, it both

reads and writes data in full-block chunks. As a result, when writing to the end of a file whose size is not an integer multiple of the block size, Lamassu will pad the final block with zeroes before writing it. In order to keep track of this padding and report a correct file size to the application at a later time, Lamassu maintains the logical size of the file without this padding. (The logical size does not include the extra space taken up by key blocks stored inside the file.) This information is stored inside the Lamassu metadata blocks of the file. Since it is highly inefficient to update every such block in a large file whenever the file changes, and it is always necessary to write to the file's last metadata block when changing its size, the updated size is written to the metadata block only for the final segment. The system always treats the file size stored in the final metadata block as the authoritative logical size for the file and ignores stale sizes that might be stored in any other blocks.

2.4 Crash Consistency

Lamassu's encryption model requires that the key for each data block be stored in the corresponding segment metadata block. Without a matching key, a data block cannot be decrypted, and becomes unreadable. This means that there is a critical failure mode wherein Lamassu crashes in between updating a data block in a segment and updating the metadata block for that segment, leaving the two in an unmatched, inconsistent state.

Lamassu addresses the threat of inconsistency due to incomplete writes by implementing a multiphase commit algorithm for writes. The sequence for each update is to first update the metadata block for the affected segment, and mark the segment as being in a midupdate state. When that has been completed, the modified data block is written out, and then, finally, the metadata block is re-marked to indicate that the update has completed.

To enable segment recovery after a failure, Lamassu stores extra key information in each metadata block during the update process. When Lamassu updates the segment metadata at the beginning of a data block write, it stores *both* the new key *and* the existing key for that block in the metadata block. Lamassu overprovisions the key table in each metadata block to provide space for a small number of transient, extra keys, stored during file writes. If the data block write succeeds, the subsequent update to the metadata block clears the update flag to indicate that the keys in the key table and the data in the data blocks are once again in sync. Key table overprovisioning slightly reduces the number of keys stored in each segment, and consequently the amount of data in each segment, but we believe this to be a good trade-off for increased crash resiliency.

If Lamassu fails during the update process, it can recover based on the contents of the metadata block. If the system discovers a segment that is marked as midupdate, it can infer that a data block update was previously interrupted. If that is the case, it can detect which data block was in the middle of an update by reading the block number attached to the key, or keys, stored in the reserved space at the end of the key table. Once it has identified an affected block, it will be able to decrypt it using either the current key, stored in the key table, or the older key, stored in the reserved space, depending on whether or not the new version of the data block made it to disk before the crash.

Lamassu depends on the underlying storage system to provide consistency guarantees on whether or not a single block-level write reaches disk. Therefore, our method does not provide any mechanism for handling a partial-block write failure or disk write failures.

The penalty for the consistency model outlined above is an amplification in the number of disk I/Os that Lamassu has to perform whenever it updates a data block. To ameliorate this draw-back, Lamassu includes the ability to batch updates for multiple data blocks into a single update operation. To do this, Lamassu writes multiple keys to a metadata block as a single block update.

Because each block included in the update must have two versions of its key written to the metadata block during the update, the number of blocks that can be combined into a single update is limited by the number of keys that can fit in the reserved space at the end of each metadata block. The precise amount of extra space reserved is adjustable at build time in our implementation. We use the parameter R to represent the number of extra keys that can be stored in the reserved space for each metadata block. Thus, with a single extra slot reserved (i.e., $R = 1$), Lamassu will update a single data block at a time, requiring three I/Os for each block write: two for the metadata updates, and one for the data block itself. Increasing the number of reserved slots in each metadata block allows Lamassu to batch multiple data block writes into a single commit operation, amortizing the cost of the metadata updates across R block updates.

Batching effectively reduces the system's I/O overhead. However, reserving more key slots for old keys reduces the number of blocks that can be managed in a single segment, increasing the space overhead of Lamassu metadata. We will discuss the space and performance trade-offs introduced by varying R in §4.3 with experimental results. Increasing R also increases the amount of data that might be lost as a result of a midupdate crash.

2.5 Data Integrity

A useful property of Lamassu's encryption strategy is that it can automatically check whether the encryption key it uses to decrypt a data bock is the correct key for

that block. When Lamassu decrypts a data block by using a convergent hash key, it can immediately attempt to re-hash the decrypted block and recompute the hash key based on the resulting plaintext. If the plaintext is correct, the resulting hash key will match the one used in the decryption. If not, the resultant key is extremely unlikely to match the original hash key. Thus, a hash mismatch indicates a block-key mismatch. Lamassu takes advantage of this property to check the integrity of individual data blocks, checking the hash of decrypted data against the hash key stored inside the metadata blocks.

In the event of a crash and recovery, this hash-checking mechanism is what allows Lamassu to determine which of the two keys assigned to a data block matches the contents of that block. If Lamassu detects a block-key mismatch that does not result from an interrupted write, it cannot correct the problem, but it can detect it and notify the client application.

Lamassu also includes integrity checking for metadata blocks, using AES-GCM authenticated encryption. AES-GCM attaches a message authentication code (MAC) to the encrypted metadata block. Decrypting the metadata block requires that the reader provide the MAC as well. In order to do that, the reader must already have access to the encryption key used to secure the block, and the secure hash of the block's original contents with which to verify its integrity.

Our design does not provide file integrity protection beyond the segment level. A malicious or defective storage system could, for example, roll the contents of a segment back from a current valid state to a previous valid state without having to read the contents of that segment. Our scheme would not detect such a change. To provide integrity checking at the level of a complete file, Lamassu would need to store data outside of the primary storage system, such as an on-premises store or, perhaps, in the key server. Lamassu's stackable design makes it possible to add an integrity layer on top of Lamassu, using a new, or existing, integrity checking system.

3 Implementation

The Lamassu prototype system takes the form of a shim layer, sitting in the data path between the application and the back-end storage system. Lamassu encrypts the data written by the application, inserts its metadata into the input data stream, and writes them all to the the backing store. The precise amount of space overhead from metadata depends on the size of the files involved, and on the block size used. Assuming the block size is 4096 bytes and that a single metadata block can store 125 keys per segment (when $R = 1$), the minimum space overhead ratio is 1/125 = 0.8%.

We selected the Linux File System in User Space (FUSE) [2] as the infrastructure for our prototype. This

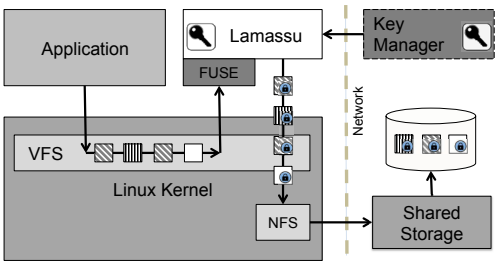


Figure 4: Lamassu prototype architecture

arrangement allowed us to build our prototype as a self-contained user-mode program that can easily be ported into another application or infrastructure in the future. Placing everything in a user space module also simplified our development and experimentation work.

Figure 4 shows the flow of data through the system. At start time, the Lamassu prototype selects a configurable directory, mounted on the native Linux file system, as its backing store. Lamassu will treat all files and directories in that mount point as Lamassu objects for it to manage. The underlying storage infrastructure for that directory can take any form, such as a local Linux file system, or an NFS mount point. Lamassu exports a file system interface to any Linux-resident application through FUSE and the Linux VFS layer. It accepts standard I/O requests and implicitly applies encryption, segmentation, and block chunking to each file before forwarding them to the backing storage system. For most of our experiments, we used a NetApp® clustered Data ONTAP® storage controller mounted over NFS as a deduplicating store. Linux applications can access the encrypted file system through the Lamassu export by using standard file I/O interfaces.

For key management, we used the Cryptsoft KMIP (Key Management Interoperability Protocol) SDK [9]. Two 256-bit AES encryption keys are retrieved at start time from a KMIP server: One is used as an inner key (K_{in}), and the other is used as an outer key (K_{out}), as described in §2.2. Every key created at the KMIP server contains an associated integer attribute called an *isolation zone*: The clients in a single isolation zone obtain the same set of encryption keys. This arrangement allows us to consistently match each Lamassu isolation zone to a KMIP isolation zone.

Our implementation exploits the architectural features provided by Intel processors to accelerate certain cryptographic operations. For the SHA-256 hash function, we make use of the Advanced Vector Extensions (AVX) instruction set by using an assembler library provided by Intel [14]. Where supported, our prototype also takes advantage of Intel's AES acceleration instruction set, AES-NI (Advanced Encryption Standard New Instructions), to maximize encryption performance. In such cases, the

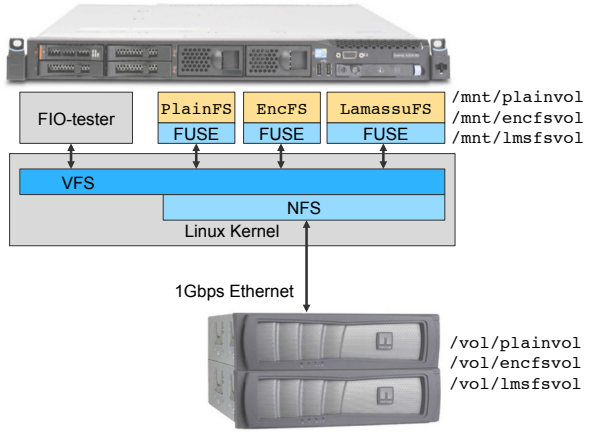


Figure 5: Experimental setup

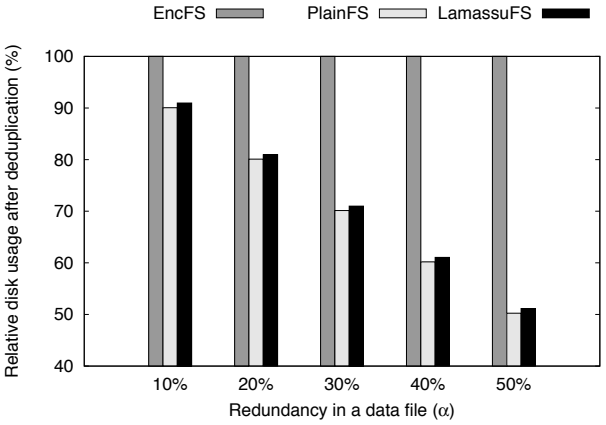


Figure 6: Storage efficiency with synthetic files

prototype uses AES-256 CBC encryption and decryption functions provided by the Intel AES-NI Sample Library [15]. On platforms where hardware acceleration is not available, our prototype defaults to the OpenSSL implementations for these functions.

4 Experimental Results

Setup. For experiments, we set up an IBM server x3550 running 64-bit Linux (Fedora 20, Linux Kernel 3.3) as a host machine. It has an Intel Xeon CPU E5-2630, an 8-core processor supporting AES-NI, which is critical for AES encryption/decryption performance. The host machine is connected with a NetApp FAS3250 controller running clustered Data ONTAP 8 via a Gigabit Ethernet switch. Figure 5 illustrates the experimental setup.

In addition to *LamassuFS*, our FUSE-based Lamassu file system implementation, we set up two additional file systems that operate via FUSE.[3] First, for a comparison with a conventional encrypted file system, we chose *EncFS* [12], an open-source FUSE-based encrypted file system that uses standard AES in CBC mode for encryption.[4] Second, we also set up an unencrypted file system via FUSE, which we refer to as *PlainFS*. This is mainly to provide a fair comparison of performance against an unencrypted system that still inludes the FUSE overhead. PlainFS is a simple pass-through front end for the relevant Linux system calls associated with FUSE operations.[5]

We created three separate volumes — `plainvol`, `encfsvol`, and `lmsfsvol` — to be used as backing stores for PlainFS, EncFS, and LamassuFS, respectively. Each volume is mounted on the host via *NFSv3* at a distinct mount point, and is used as a backing store for the corresponding FUSE-based file system. We used the same 4096-byte block size for both EncFS and LamassuFS. This helped to keep our comparisons fair, and ensured that I/O operations for both would be aligned with the native block sizes of our storage controller. For most experiments, the number of reserved key slots in the metadata block (R) was fixed to 8. With this setup, a single segment is composed of one metadata block followed 118 data blocks, and the minimum amount of space overhead is 0.85%.

4.1 Storage Efficiency

We first evaluated Lamassu storage efficiency to make sure that we could achieve the deduplication goals we had set. To do this, we wrote a simple tool to generate 4GB synthetic data files with various redundancy profiles (as the percentage of redundant 4KB blocks in a file, denoted α) ranging from 10% to 50%. Each data file was copied over NFS to different volumes in the storage system through PlainFS, EncFS, and LamassuFS. When the copy completed, we manually triggered deduplication on the storage system. We measured the difference in disk space usage before and after deduplication using `df`, run on the controller itself.

The relative percentage disk usage after deduplication is plotted in Figure 6. EncFS shows 100% for all cases because no deduplication occurred by using standard AES encryption. For PlainFS where data files are stored as unencrypted blocks, the relative disk usage is exactly $(1-\alpha)$. These results match our expectations, with the space savings from deduplication on unencrypted data mapping 1-to-1 with the known level of data redundancy on the test data. LamassuFS achieved nearly the same storage efficiency as PlainFS, but with a small amount of space overhead due to the embedded cryptographic metadata. This overhead is constant, relative to the nondeduplicated size of a file, but the relative overhead on dedu-

[3] FUSE version 2.9.3-2
[4] EncFS version 1.8-rc1
[5] Most code from *fuse-examplefs* [1]

Table 1: Storage efficiency with VM images

VM image	Size	% deduplicated		Space overhead
		PlainFS	LamassuFS	
FreeDOS.vdi	379M	9.35%	9.18%	1.07%
FreeBSD-7.1-i386.vdi	1.8G	15.40%	15.11%	1.35%
xubuntu_1204.vdi	2.3G	22.07%	21.95%	1.01%
Fedora-17-x86.vdi	2.6G	36.73%	36.46%	1.83%
opensolaris-x86.vdi	3.5G	8.08%	7.87%	1.14%

plicated storage increases as the data file redundancy (α) increases: 1.01%, 1.06%, 1.21%, 1.43%, and 1.81% respectively, i.e., inversely proportional to $(1-\alpha)$.

The same set of experiments was performed by using real virtual machine images[6] with various sizes as shown in Table 1. Note that EncFS results have omitted because they were all zero. The storage efficiency results with real files are completely consistent with the results from synthetic data shown in Figure 6: LamassuFS achieves almost the same amount of deduplication as PlainFS, with a small amount of space overhead of less than 2%.

4.2 Performance

Because AES encryption and decryption are a compute-intensive jobs, using encryption in a file system incurs a performance overhead. In order to examine this, we evaluated the I/O performance of PlainFS, EncFS, and LamassuFS. For a fair comparison, we carefully chose the EncFS configuration parameters: 4096 bytes for a block size, AES-256 in CBC mode for an encryption algorithm, and no file name encryption. We also turned off all EncFS features that insert metadata between blocks. This change caused EncFS to write data in a block-aligned pattern, similar to Lamassu's. We did this because we have observed that EncFS performs quite poorly when allowed to write in an unaligned pattern. EncFS also uses AES-NI through the OpenSSL library on platforms that support it.

In order to examine the performance of Lamassu under a larger variety of circumstances, we used *FIO-tester* [4], which generates various types of synthetic workloads to all 3 file systems. We applied 5 different workloads to a single 256MB file with 4KB-block synchronous I/O: sequential reads (seq-read), sequential writes (seq-write), random reads (rand-read), random writes (rand-write), and mixed random reads/writes with the read/write ratio of 7:3 (rand-rw). The I/O throughput (bandwidth) was measured through 10 runs; the Linux kernel page cache was flushed before each run so that no data block was cached at the host memory. For LamassuFS, one more variation was added: *LamassuFS(meta-only)*, where the read path only checks the integrity of metadata blocks without checking the integrity of data blocks. This would give a good indication of how much performance is penalized when providing a full data integrity check to the system.

[6]Obtained from `http://virtualboxes.org/images/`

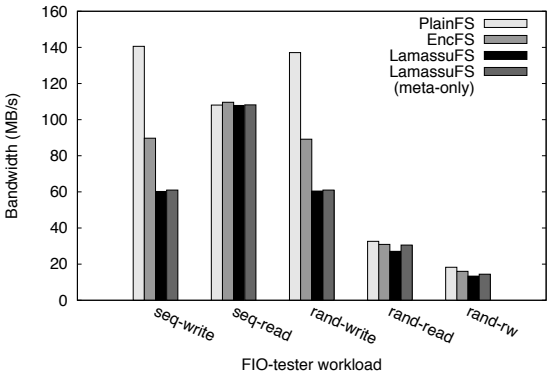


Figure 7: Single-file I/O throughput with a remote filer

Figure 7 shows the single-file I/O throughput of PlainFS, EncFS, LamassuFS, and LamassuFS(meta-only) working with the remote filer via NFS. With pure write workloads (seq-write and rand-write), we see that PlainFS performs much better than both EncFS and LamassuFS. With pure read workloads (seq-read and rand-read), the throughput does not show any meaningful difference across all FS: LamassuFS shows slightly worse performance than EncFS (1.6% to 12.4% worse) with read workloads. However, LamassuFS is noticeably worse than EncFS with write workloads: 32.9% for seq-write and 32.2% for rand-write.

The difference in write performance is due to per-block SHA-256 hash computations that are necessary for convergent encryption. Because this happens at the very beginning of block encryption process, extra latency caused by SHA-256 computation has a direct negative impact on I/O throughput. On the other hand, extra SHA-256 computation that happens during the LamassuFS read path (for data block integrity checking) rarely affects the performance: LamassuFS and LamassuFS(meta-only) do not show any meaningful throughput difference. This suggests that NFS I/O is a dominant performance bottleneck in read workloads, and therefore the rest of the computation that happens after I/O has almost has no impact on overall I/O throughput. This also explains why both EncFS and LamassuFS are as good as PlainFS with read workloads. A possible option for improving the write performance is to increase the number of reserved key slots (R) in a metadata block — with some trade-offs; we will discuss this later in §4.3.

Overall, despite the performance overhead caused by the extra hash computation and metadata I/O, we can say that the performance of LamassuFS is competitive with that of EncFS in an NFS-shared storage environment. Lamassu's strategy of inserting metadata blocks in

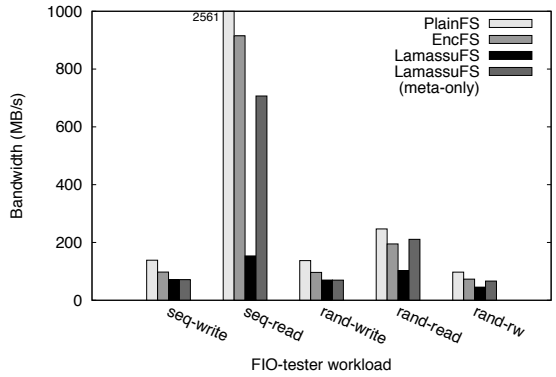


Figure 8: Single-file I/O throughput with a RAM disk

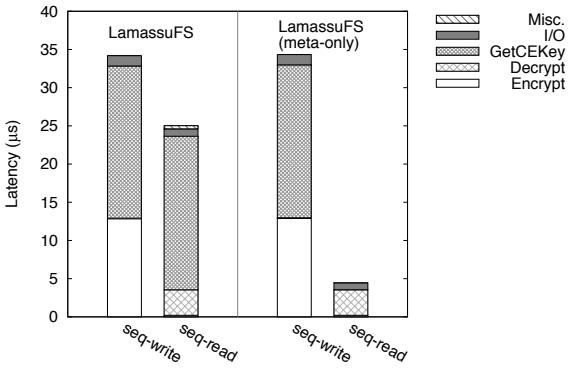


Figure 9: Write/read latency breakdown of LamassuFS with, and without, a full data integrity check, on a RAM disk

a block-aligned fashion turns out to play a significant role in terms of performance in our experimental environment. We have observed that block-unaligned accesses over NFS incur a huge performance overhead. For example, block-unaligned EncFS is at least 10x slower than block-aligned one when used over NFS: 7MB/s versus 85MB/s throughput in the case of seq-write. For this reason, to ensure as fair a comparison as possible, we have configured EncFS so that it does not add any unaligned metadata in our experiments.

In order to evaluate the pure performance overhead of encryption without the impact of NFS I/O affected by the network bandwidth, we ran the same set of FIO tests by setting up all file systems so that they used a local RAM disk (`tmpfs` in Linux) as backing stores, instead of the remote filer. Figure 8 shows the single-file I/O throughput of all FS with a local RAM disk. PlainFS always noticeably performed better than EncFS and LamassuFS across all workloads: The difference is the greatest with seq-read showing 2.80x over EncFS, 16.70x over LamassuFS (note that the graph has a short y-axis).

After removing the NFS I/O bottleneck from the read path, computation that occurs after I/O becomes a dominant bottleneck. In particular, extra SHA-256 hash computation that is added for a data block integrity check negatively affects the read throughput of LamassuFS significantly: LamassuFS performs 83.2% worse than EncFS with a full data integrity check, but it performs only 22.8% worse without it. LamassuFS also performs worse than EncFS with the rand-read workload: 47% worse than EncFS. However, it shows a slightly better (8.1%) throughput than EncFS without full data integrity checking (meta-only): This is due to a small amount of write buffering introduced to provide consistency and reduce overall I/O. (Recall that $R = 8$ in these experiments.) The seq-write and rand-write results are quite similar to those of NFS cases: EncFS performs 26.8% to 27.5% better than LamassuFS with write workloads.

In order to evaluate the impact of SHA-256 hash computation on performance, we broke down the write and read latency of LamassuFS when it operates on a local RAM disk. By inserting the instrumentation code, the time spent on the LamassuFS read or write path is measured and divided into five categories: Encrypt, Decrypt, GetCEKey, I/O and Misc. Note that the major workload of GetCEKey is SHA-256 hash computation. Figure 9 shows seq-write and seq-read latency breakdown of LamassuFS with, and without, full data integrity check. For both writes and reads, GetCEKey consumes the most time: 58% of seq-write, 80% of seq-read latency. Without full data integrity check (meta-only), the read latency reduces drastically (81%) because SHA-256 hash computation is not on the read path.

There are a couple of possible options to improve the performance of LamassuFS. Since we have identified that the SHA-256 hash computation is the biggest performance bottleneck, the first option is using a different cryptographic hash function that consumes fewer CPU cycles. For example, our microbenchmark results showed that OpenSSL SHA-1 consumes 58% fewer, and OpenSSL MD5 consumes 38% fewer CPU cycles for computing the same 4KB block-hash compared with our SHA-256 function using the Intel AVX instruction set. The exact implication of using a less secure hash function (e.g., SHA-1 or MD5 generates 128-bit keys instead of 256-bit keys) for convergent encryption could be understood only with comprehensive cryptographic analysis, and hence we will leave it for future work.

The second option is to forgo data block integrity check in the read path. This will improve the read performance significantly, as shown in Figure 8 with LamassuFS(meta-only). Remember that Lamassu is still doing metadata block integrity checking via AES-GCM: It is always able to detect any data corruption that occurs within or across metadata blocks (e.g., first 4KB block from one hundred and nineteen 4KB blocks if

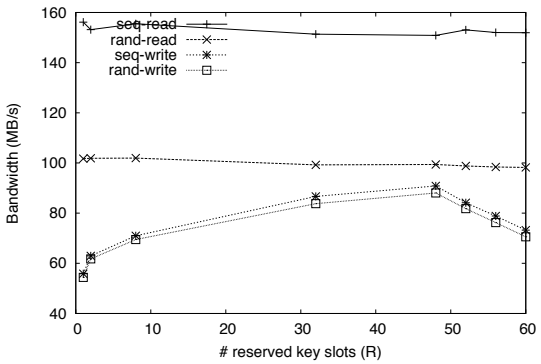


Figure 10: Single-file I/O throughput by varying R

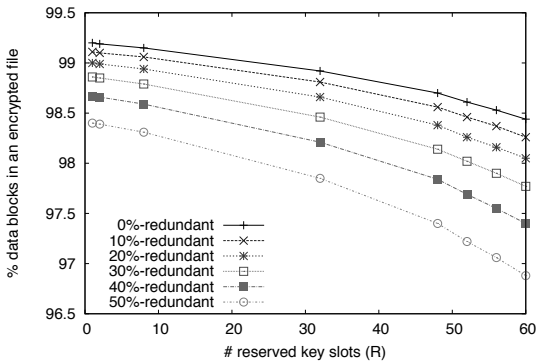


Figure 11: Storage efficiency by varying R

$R = 8$). This covers cases such as accidentally overwriting a whole file, or the beginning of a file. In an enterprise storage environment with no malicious user who intentionally corrupts the user data, this could be a viable option for improving performance while sacrificing a little on security.

4.3 Number of Reserved Key Slots

As described in §2.4, Lamassu maintains a certain number of reserved key slots (denoted R) in a metadata block to maintain consistency. As R increases, the amount of space taken up by Lamassu metadata in each file increases, and thus storage efficiency decreases. On the other hand, increasing R reduces the number of additional metadata I/Os that Lamassu must perform. To maintain consistency, Lamassu caches block writes in memory and writes them to disk along with their metadata as part of a commit operation. That occurs once for every R data block writes. The decreased number of writes positively affects the write throughput, while the increased metadata space overhead negatively affects storage efficiency. In order to understand this trade-off, we evaluated the performance and storage efficiency of LamassuFS by varying $R = 1$, 2, 8, 32, 48, 52, 56, and 60.

Figure 10 shows the single-file I/O throughput in bandwidth when applying four different FIO-tester workloads to LamassuFS with a local RAM disk backend. By increasing R, the write throughput continuously improves up to a certain point and then decreases: The throughput reaches its peak around $R = 48$ achieving 1.60x and 1.57x speedups over $R = 1$ for seq-write and rand-write respectively. For write workloads, the positive impact of buffering and batching of writes (i.e., reduced metadata I/O) is a dominant factor as it increases the write throughput significantly. On the other hand, the read throughput tends to decrease slightly as R increases: from 1 to 60, 4.71% and 4.40% decreases for seq-read and rand-read, respectively. This is because Lamassu must read more metadata blocks per unit file size with a larger R value (i.e., metadata space overhead increases), resulting in a slight increase in I/O overhead.

Figure 11 shows storage efficiency as the percentage of data blocks, excluding metadata blocks, in different encrypted files with various redundancy profiles (denoted as α in §4.1). The storage efficiency decreases as R increases because of the larger space overhead of metadata. The storage efficiency also decreases as there are more redundant blocks in a plaintext file (i.e., α increases) because the metadata blocks are not deduplicated, as previously shown in §4.1.

The right value of R should be chosen with consideration for this trade-off. If an application requires higher write IOPS, a larger R can be chosen while sacrificing a little space efficiency. However, with larger R, the granularity of crash consistency becomes coarser, (i.e., it increases the recovery point objective [RPO] of the system.), and LamassuFS consumes the additional memory space for more write buffers. For the proceeding set of experiments, we fixed R to be 8 to achieve a balanced trade-off.

5 Related Work

5.1 Encrypted File Systems

Full disk encryption (FDE) is a popular choice for the storage encryption. NetApp Storage Encryption (NSE) [16] is a hardware-based implementation of FDE that uses self-encrypting drives (SEDs) from drive vendors. FileVault 2 [3] is a software-based FDE in Mac OS X that uses Intel's AES-NI. As encryption occurs at the lowest stack just before data blocks are written to the disk, FDE is quite a different approach from our data-source encryption strategy.

FDE encrypts whole blocks in a volume, including file system metadata, while file-system-level encryption enables encryption of individual files or directories, offering a finer granularity of control. There are general-purpose file systems that have integrated encryption features, such as ZFS and Encrypting File System (EFS) in NTFS. Some cryptographic file systems — such as

CFS [7], TCFS [8], Cryptfs [22] and eCryptfs [13] — are stackable on top of another general-purpose file system; eCryptFS is widely used, included in Ubuntu's encrypted home directory and Google's Chrome OS. There are a few FUSE-based encrypted file systems available; only EncFS [12] is notable in terms of its maturity and wide acceptance.

With regard to our data-source encryption strategy, any stackable or FUSE-based encrypted file systems can serve the same purpose because they can transparently work on top of an existing system. However, to the best of our knowledge, none of them provides the explicit ability to enable deduplication at downstream storage devices as Lamassu does. We chose a FUSE-based file system for our prototype implementation due to its better debuggability and easier deployment; it can also be implemented as a kernel-level file system if necessary.

5.2 Convergent Encryption

Deduplication of previously encrypted data is normally impossible because of the nature of encrypted data. Convergent encryption (CE) was proposed to address this issue. The concept of convergent encryption was introduced by Douceur et al. [10]: By definition, CE produces identical ciphertext files from identical plaintext files. A common approach is deriving the encryption key from a secure hash of the plaintext. Using convergent encryption results in the ciphertext having the same levels of duplication as the plaintext. However, the weakness of CE is the leakage of information about the plaintext; an attacker can observe the ciphertext and deduce the contents of the plaintext by using a variety of different attacks [20].

While the system described by Douceur, et al.[10] works only with the whole files, Storer et al. [18] later designed a CE solution that provides sub-file granularity encryption and deduplication, in both fixed and variable sized chunks. Bellare et al. [5] formalized CE as Message-Locked Encryption (MLE) with a cryptographic analysis. DupLESS [6] tried to overcome CE's weakness — leakage of information about the plaintext — with an obfuscated key exchange mechanism with a key server in order to achieve stronger confidentiality; however, the performance overhead turns out to be quite costly as it requires 3-way key exchange with a key server for every block access. ClouDedup [17] uses a semi-trusted server between users and the cloud provider to encrypt the ciphertext resulting from CE with another encryption algorithm, and a metadata manager to store encryption keys and block signatures. It introduces much more complexity to the system, compared to Lamassu, and might incur performance penalty due to double encryption.

Lamassu is targeted at enterprise environments in which multiple hosts store data in a large shared storage appliance. Therefore, it tries to achieve a balance between performance and security. Lamassu can be easily added to an existing enterprise environment. It only incurs a mimal performance overhead, and does not require an extra system for a metadata store. In a multitenant system, tenant data can be securely separated by using per-tenant keys to create isolation zones. Tahoe-LAFS [21] used a similar approach of adding a secret during hash key generation [20], but its convergent encryption works on a per-file basis, limiting the storage efficiency compared with Lamassu's per-block approach.

6 Conclusion

In this paper, we presented Lamassu, a new, transparent, encryption system that provides strong data-source encryption, while preserving downstream storage-based data deduplication. Lamassu uses block-oriented convergent encryption to align with existing block-based deduplication systems. It takes a new approach to manage convergent encryption key metadata by inserting it into each file's data stream, eliminating the need for additional infrastructure. Therefore, it can be inserted into an existing application stack without any modification to either host-side applications or the storage controller. We also introduced a strategy for maintaining consistency between file data and convergent metadata, and for providing data integrity checking for application data in a convergent encryption system.

Our results showed that it is possible to insert convergent encryption into an application with a performance overhead similar to non-convergent options, placing encryption near the top of the application stack, and making it easier to provide strong security across the whole stack. Our security model leaks only the information that is absolutely necessary for deduplication to the storage system, resulting in strong encryption, well suited to many applications. Our system provides a clear advantage over analogous solutions by preserving storage-based deduplication without compromising on encryption.

Acknowledgments. We thank Emil Sit, ATC PC members and anonymous reviewers for their feedback and guidance. We also thank James Kelley and James Lentini at NetApp for their help and support for this work.

References

[1] fuse-examplefs: an example C++ FUSE file system. `http://code.google.com/p/fuse-examplefs/`.

[2] FUSE: Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[3] APPLE. OS X: About FileVault 2. `http://support.apple.com/en-us/HT4790`.

[4] AXBOE, J. FIO: flexible I/O tester. `http://github.com/axboe/fio`.

[5] BELLARE, M., KEELVEEDHI, S., AND RISTENPART, T. Message-locked encryption and secure deduplication. *IACR Cryptology ePrint Archive 2012* (2012), 631.

[6] BELLARE, M., KEELVEEDHI, S., AND RISTENPART, T. DupLESS: Server-aided encryption for deduplicated storage. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., 2013), USENIX, pp. 179–194.

[7] BLAZE, M. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (New York, NY, USA, 1993), CCS '93, ACM, pp. 9–16.

[8] CATTANEO, G., CATUOGNO, L., SORBO, A. D., AND PERSIANO, P. The design and implementation of a transparent cryptographic file system for UNIX. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 199–212.

[9] CRYPTSOFT. KMIP C SDK. `http://www.cryptsoft.com/data/KMIP-C-SDK_opt.pdf`.

[10] DOUCEUR, J. R., ADYA, A., BOLOSKY, W. J., SIMON, D., AND THEIMER, M. Reclaiming space from duplicate files in a serverless distributed file system. In *International Conference on Distributed Computing Systems* (2002), pp. 617–624.

[11] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (1987), STOC '87, pp. 218–229.

[12] GOUGH, V. EncFS: an Encrypted Filesystem for FUSE. `http://vgough.github.io/encfs/`.

[13] HALCROW, M. A. eCryptfs: An enterprise-class cryptographic filesystem for Linux. `http://ecryptfs.sourceforge.net/ecryptfs.pdf`.

[14] INTEL. Fast SHA-256 implementations on Intel architecture processors. `http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/sha-256-implementations-paper.html`.

[15] INTEL. Intel AES New Instructions (AES-NI) Sample Library. `http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library`.

[16] NETAPP. NetApp Storage Encryption. `http://www.netapp.com/us/products/storage-security-systems/netapp-storage-encryption.aspx`.

[17] PUZIO, P., MOLVA, R., ONEN, M., AND LOUREIRO, S. ClouDedup: Secure deduplication with encrypted data for cloud storage. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on* (Dec 2013), vol. 1, pp. 363–370.

[18] STORER, M. W., GREENAN, K., LONG, D. D., AND MILLER, E. L. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability* (New York, NY, USA, 2008), StorageSS '08, ACM, pp. 1–10.

[19] VMWARE. VMware Workstation Overview. `http://www.vmware.com/products/workstation`.

[20] WILCOX-O'HEARN, Z. Drew Perttula and attacks on convergent encryption. `http://tahoe-lafs.org/hacktahoelafs/drew_perttula.html`, 2008.

[21] WILCOX-O'HEARN, Z., AND WARNER, B. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability* (New York, NY, USA, 2008), StorageSS '08, ACM, pp. 21–26.

[22] ZADOK, E., BADULESCU, I., AND SHENDER, A. Cryptfs: A stackable vnode level encryption file system. Tech. rep., Columbia University, 1998.

NetApp, the NetApp logo and Data ONTAP are trademarks, or registered trademarks, of NetApp, Inc. in the United States and/or other countries. All other brands or products are trademarks, or registered trademarks, of their respective holders, and should be treated as such. A current list of NetApp trademarks is available on the web at `http://www.netapp.com/us/legal/netapptmlist.aspx`.

SecPod: A Framework for Virtualization-based Security Systems

Xiaoguang Wang[†♮], Yue Chen[†], Zhi Wang[†], Yong Qi[♮], Yajin Zhou[‡]
Florida State University[†] Xi'an Jiaotong University[♮] Qihoo 360[‡]

abstract

The OS kernel is critical to the security of a computer system. Many systems have been proposed to improve its security. A fundamental weakness of those systems is that page tables, the data structures that control the memory protection, are not isolated from the vulnerable kernel, and thus subject to tampering. To address that, researchers have relied on virtualization for reliable kernel memory protection. Unfortunately, such memory protection requires to monitor every update to the guest's page tables. This fundamentally conflicts with the recent advances in the hardware virtualization support. In this paper, we propose SecPod, an extensible framework for virtualization-based security systems that can provide both strong isolation and the compatibility with modern hardware. SecPod has two key techniques: *paging delegation* delegates and audits the kernel's paging operations to a secure space; *execution trapping* intercepts the (compromised) kernel's attempts to subvert SecPod by misusing privileged instructions. We have implemented a prototype of SecPod based on KVM. Our experiments show that SecPod is both effective and efficient.

1 Introduction

With its privilege, an operating system (OS) kernel is critical to the security of the whole system. Unfortunately, modern kernels are too complicated to be secure – they often consist of tens of million lines of source code. Consequently, an increasingly large number of vulnerabilities are discovered in all major kernels each year [10]. These vulnerabilities are routinely being exploited to take over the system. To address that, researchers and practitioners have proposed many solutions. For example, modern kernels all have built-in exploit mitigation mechanisms such as address space layout randomization(ASLR) [26] and data execution prevention (DEP, or $W \oplus X$) [12]. They significantly raise the bar of functioning kernel exploits. However, these systems are built on top of a weak foundation that *page tables, the data structures that control the memory protection, are always writable in the kernel* (to facilitate frequent page table updates). Any in-kernel memory protection accordingly can be circumvented by manipulating page tables. To that end, a stream of research has proposed to deploy memory and other protections "out-of-the-box" in a virtualized environment [22, 27, 28, 31, 33, 35, 37, 45]. For example, Patagonix extends the hypervisor to identify and protect the code running in the VM [28]. NICKLE achieves a similar goal through memory shadowing [31].

Virtualization-based security systems are often at odds with recent advances in the hardware virtualization support: many security tools need to intercept and respond to key events in the VM. Each intercepted event causes one or more expensive *world switches* between the virtual machine and the hypervisor. On the other hand, the hardware virtualization support, such as AMD-V and Intel VT, strives to reduce world switches. In particular, the nested paging allows guests to freely update their page tables without involving the hypervisor. However, the guest page table update is a key event that many security tools are interested in [27, 28, 31, 45]. This forces the hypervisor to run in the less-efficient shadow paging mode where updates to guest page tables are trapped and verified by the hypervisor. To reconcile this conflict, it calls for a new approach that can accommodate the needs of virtualization-based security tools, but also take full advantage of the hardware virtualization support.

In this paper, we propose SecPod, an extensible framework for virtualization-based security systems. SecPod encapsulates a security tool in a trusted execution environment that coexists with and yet is strictly isolated from the vulnerable kernel. Specifically, it creates a dedicated address space (the secure space) in parallel to the existing kernel address space (the normal space). The secure space is rigorously protected from the normal space by the two key techniques of SecPod, *paging delegation* and *execution trapping*: in the former, the kernel delegates all its paging operations, including page tables and their updates, to the secure space. The kernel is deprived of the privilege to directly modify the effective page tables. The secure space enforces a non-bypassable memory isolation by sanitizing the guest page table updates. The latter foils the attacker's attempts to subvert the secure space by misusing privileged instructions. The hypervisor notifies the secure space any such attempts via signals. The secure space can accordingly respond to the event by,

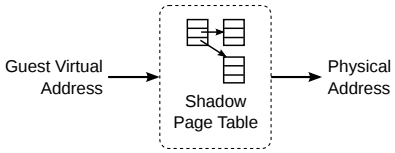


Figure 1: Shadow paging (GPT not in effect)

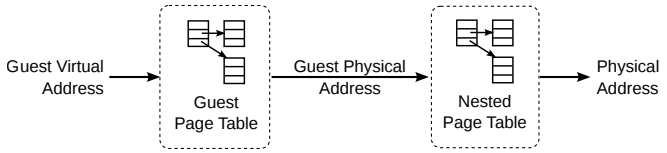


Figure 2: Nested paging (GPT and NPT both in effect)

say, issuing an alert or terminating the VM. The synergy of these two techniques isolates a security tool from the (compromised) kernel.

We have implemented a prototype of SecPod based on the popular KVM hypervisor [24]. Our prototyping efforts show that SecPod can be integrated into an existing hypervisor with a minimal increase to its code base. Our experiments demonstrate the efficiency and effectiveness of SecPod. For example, SecPod introduces about 3% of overhead on average for the I/O-intensive SysBench FileIO benchmark, and about 5% overhead on average for the SysBench online database transaction benchmark.

The rest of this paper is organized as the following: in Section 2, we define the scope of the problem and the threat model. We then describe the design, implementation, and evaluation of SecPod in Section 3, 4, and 5, respectively. Finally, we present the related work in Section 6 and conclude the paper in Section 7.

2 Problem Overview

In this section, we give a brief overview of the hardware virtualization support, particularly the memory virtualization support, and explain how they impact the design of security tools. Early hypervisors for x86 virtualize the guest memory with shadow paging, in which a guest page table (GPT) is superseded by its shadow page table (SPT) [3] (Figure 1). Specifically, the hypervisor manages a SPT for each guest page table. Any changes to the GPT must be synchronized to its SPT to take effect. This provides an opportunity for security tools to examine and control every change to guest page tables [27, 28, 31, 33, 38, 45]. In shadow paging, GPTs translate guest virtual addresses to guest physical addresses, i.e., the virtual and physical addresses from the guest's perspective. Guest physical addresses must be further translated to the actual physical addresses used by the memory controller. Since SPTs are the only effective page tables, they map directly from guest virtual addresses to physical addresses (Figure 1).

Recent x86 processors have the hardware virtualization support. Early extensions focus on trapping sensitive guest instructions, such as `SGDT`, `SIDT` and `MOV to CR3`, to allow the hypervisor to virtualize the related resources. Later revisions aim at improving the performance with the direct support for critical virtualization tasks. Particularly, nested paging is a hardware support for memory virtualization in which the processor translates guest memory accesses with two levels of page tables (Figure 2): the GPT maps guest virtual addresses to guest physical addresses, and the nested page table further maps guest physical addresses to physical addresses (NPT is also called extended page table. For clarity, we use NPT.) The guest has full control over its GPTs, while the hypervisor manages NPTs and is not aware of changes to GPTs. Consequently, memory protection enforced in NPTs can be circumvented by remapping the (protected) guest virtual memory in GPTs. For example, data execution prevention (DEP) enforced in the NPT can be foiled by remapping the guest kernel code to the writable-and-executable physical memory. Because of this, many virtualization-base security systems cannot take full advantage of nested paging, which has tremendous advantages in performance than shadow paging [42].

Threat model: in this paper, we assume a trusted booting protocol, such as tboot [41], is used to securely load the hypervisor, which in turn loads the guest OS and initializes SecPod. The guest kernel is benign but contains exploitable vulnerabilities. After boot, we assume a powerful attacker exists that can change arbitrary memory of the kernel by exploiting some vulnerabilities. Moreover, we consider the hypervisor to be trusted. This can be guaranteed by recent advances in the hypervisor integrity through formal verification and integrity protection and monitoring [25, 29, 40, 44, 46].

3 System Design

3.1 System overview

SecPod aims at providing a trusted execution environment for virtualization-based security tools. Figure 3 gives an overview of SecPod with the two key techniques: *paging delegation* and *execution trapping*. In this architecture, security tools run in a dedicated secure space defined by the SecPod page table, while the kernel runs in the normal space defined by the kernel page table. An entry gate and an exit gate are responsible for switching these two spaces. This is essentially a page table based isolation [35, 39, 46]. To switch the space, the entry or exit gate only needs to load the respective next page table into `CR3`, the page table base register of x86. The entry gate is the only way to enter the secure space from the normal space as

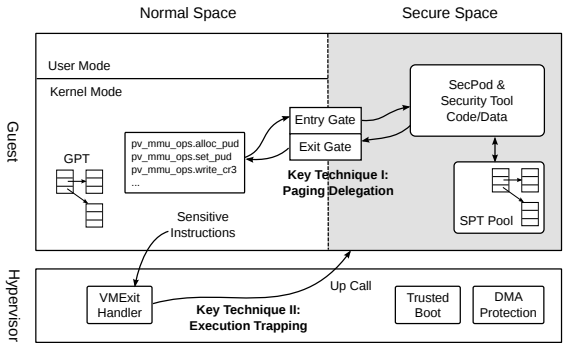


Figure 3: The overview of SecPod

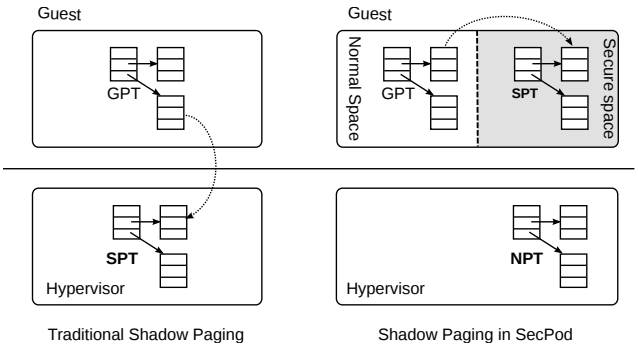


Figure 4: Shadow page table in virtualization & SecPod

guaranteed by execution trapping. SecPod provides one-way visibility into the kernel – a security tool in SecPod can introspect and even modify the kernel memory, but not the other way around.

However, simple page table based isolation is not secure for three reasons: *first*, the kernel still has full control over its page table. This allows the (compromised) kernel to subvert SecPod by mapping and modifying the secure space memory. It is thus critical to validate the kernel's page table updates to enforce strict memory isolation. SecPod solves this challenge with the first technique, *paging delegation*, in which the kernel delegates all its paging operations to the secure space, including page tables, page table updates, and task switches (one step of a task switch is to load the page table of the next process to `CR3`). Accordingly, the kernel, including kernel exploits, cannot modify its page tables. All the updates must be delegated to and sanitized by the secure space. *Second*, the kernel is still privileged and free to execute privileged instructions. These instructions can be misused to compromise SecPod. For example, the kernel could use the `MOV to CR3` instruction to load a crafted page table to bypass the secure space. SecPod relies on the second technique, *execution trapping*, to eliminate this threat. Specifically, the hypervisor intercepts sensitive privileged instructions executed by the kernel, and forwards the captured events to the secure space as signals. The secure space can decide how to respond, for example, by issuing alerts, ignoring them, or terminating the violating kernel. It can also dispatch the events to the security tools. This whole process is similar to the signal handling in traditional OSes. *Third*, the attacker could attempt to subvert SecPod through DMA attacks [47]. DMA operations by hardware devices use physical addresses, and thus are not translated by page tables (page tables are used by the CPU to translate software memory accesses.) The hypervisor should have already employed IOMMU to thwart DMA attacks. The secure space should be excluded from the memory accessible to devices in IOMMU as well. In the rest of this section, we describe these two key techniques in detail.

3.2 Paging Delegation

SecPod delegates the kernel's paging operations to the secure space in order to enforce memory isolation. Specifically, the secure space maintains the shadow page tables (SPTs) for the kernel. SPTs stay synchronized with the kernel's page tables. Any updates to the kernel page tables must be merged to SPTs to take effect because SPTs are the only page tables used by the CPU. The kernel may keep its own page tables to facilitate implementation, but they are never loaded to the CPU for address translation. This is technically similar to shadow paging in the traditional virtualization systems. Figure 4 compares these two shadow paging designs. In virtualization, SPTs are managed by the hypervisor, which is responsible for synchronizing any GPT updates to SPTs. SPTs are the only page tables in use for the guest. Accordingly, SPTs translate guest virtual addresses directly to physical addresses (Figure 1); In SecPod, SPTs are instead managed by the *in-VM* secure space. It is further backed by the nested page tables (NPTs). Both SPTs and NPTs are used by the CPU to translate guest addresses. SPTs thus map guest virtual addresses to *guest* physical addresses. In most cases, a SPT in SecPod is a simple replica of the kernel's page table (unless a memory safety violation is detected and rejected). Shadow paging in SecPod is thus straightforward to implement. This is in stark contrast against shadow paging in virtualization, which is one of the most complicated modules in a hypervisor due to its support of many paging modes of x86 and the intricate out-of-sync shadowing. Shadow paging in SecPod is also more efficient than the traditional shadow paging – updating SPTs in SecPod take a fast context switch, instead of a much slower world switch in virtualization. In short, SecPod keeps both the simplicity and efficiency of the nested paging. Even though shadow paging has long been used in virtualization, it is, to the best of our knowledge, the first time to be proposed in this architecture.

The kernel delegates its page tables and all paging-related operations to the secure space, such as page table allocation, page table updates, task switches (to write

to CR3), and TLB flushing. The secure space exposes, through the entry gate, a service for each of these operations. To delegate these operations, we could replace every paging operation in the kernel with a call to the respective service in the secure space. Fortunately, for kernels that can run in a para-virtualized (PV) VM [3], these hooks have already been embedded into the kernel. For example, the Linux kernel has a `pvops` framework that can figure out at run-time whether it is running in a virtualized system and accordingly switch to the optimized low-level operations. The `pvops` framework consists of several groups of low-level operations, such as `pv_time_ops`, `pv_cpu_ops`, `pv_mmu_ops`, and `pv_lock_ops` (defined in file `arch/x86/include/asm/paravirt_types.h`). We can repurpose `pv_mmu_ops` to implement paging delegation (Section 4.1). For a kernel without the PV interface, we can potentially patch the kernel to implement a similar interface.

3.2.1 SecPod Address Space Layout

Figure 5 shows the layout of the normal and secure spaces. The normal space, as usual, consists of the kernel and the user space. The kernel is mapped at the same location in the secure space as in the normal space. Accordingly, a security tool in SecPod can access the kernel as if it is running inside it since key kernel data structures remain at their supposed locations. This helps mitigate the semantic gap problem [5]. The kernel memory is set to non-executable in the secure space to prevent security tools from executing the (untrusted) kernel code. In the secure space, the secure code and its data are placed in the lower address space because the kernel usually sits at the top (e.g., the Linux kernel often occupies the top 1GB of the address space.) The secure code provides security tools with a compact library of useful functions such as `malloc`, `free`, and `string` functions. The secure data includes a repository of shadow page tables and several hash-based data structures for fast index of that repository (Section 3.2.3). The entry gate is the only entrance to the secure space from the normal space, while the exit gate returns to the normal space. Both gates should be mapped at the same location in the normal and secure spaces because the page table is reloaded during each context switch, and the page-table-reloading code is architecturally required to remain unchanged before and after a context switch [20]. There is also a shared page to pass data between two spaces.

The memory for the secure space is allocated from the kernel when the secure space is created. It is subsequently removed from the kernel so that the kernel will not use it for other purposes. We enforce $W \oplus X$ in the secure space; i.e., the secure space can be either writable or executable, but not both simultaneously [12]. This thwarts code injection attacks against the secure space in case the security tool contains exploitable vulnerabilities. Other attack mitigation mechanisms can also be employed to provide stronger protection of the secure space [1, 26].

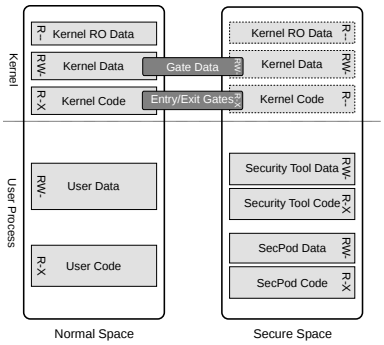


Figure 5: SecPod address space layout

3.2.2 Secure and Efficient Context Switch

SecPod implements the page-table based isolation. To switch the spaces, we need to load the page table of the next space into CR3. The secure space only has one page table, the SecPod page table, but the normal space has many shadow page tables, one for each user process. We need to ensure the security and atomicity of context switches. To this end, the entry gate saves the kernel state to the stack (generic registers and interrupt enable/disable status), clears the interrupt (twice), and then enters the secure space by loading its page table and stack to the processor. This process has been described in detail by earlier papers [35, 38]. Interested readers please refer to those papers. The exit gate performs the opposite operations in the reverse order to return to the normal space.

To prevent the kernel from subverting the secure space by loading a crafted page table, we request the hypervisor to intercept and check every write to CR3 by the guest (Section 3.3). However, trapping every CR3 write could cause substantial performance overhead due to frequent context switches. To reduce the overhead, we leverage a hardware feature called CR3 target-list [20]. Loading CR3 with one of the four page tables in the CR3 target-list will not be trapped by the hypervisor. This feature has been employed by earlier work for similar purposes [35, 38]. The major difference lies in how memory is virtualized. The previous systems use shadow paging to virtualize the guest memory. Guest task switches are thus handled by and in the hypervisor. This provides a convenient opportunity to update the CR3 target-list (CR3 target-list can only be updated by the hypervisor). On the downside, this prevents these systems from taking advantage of nested paging. SecPod is designed to avoid this problem.

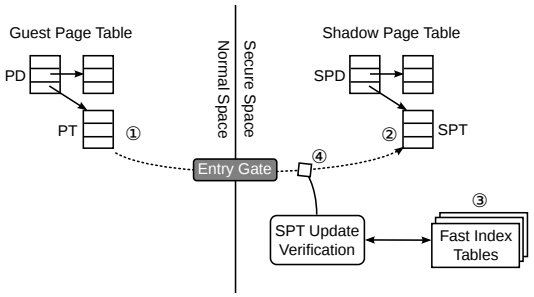


Figure 6: Kernel Page table update verification

The hypervisor in SecPod uses nested paging, and the guest delegates its paging operations to the secure space, including task switches. Ideally, task switches in the guest should not involve the hypervisor, just like in the normal nested paging. However, there are many shadow page tables for the guest yet the CR3 target list can only hold four page table roots. The entry gate will never cause any VM exits because the SecPod page table is locked in the list. But the exit gate will if the SPT for the normal space is not in the list. Neither the kernel nor the secure space can update the CR3 target-list because they both run in the guest mode. To address that, we allocate a fixed top-level page table (FTLPT) in the secure space and copy the top-level page table of the next SPT to it during the task switch. As such, SecPod appears (to the hardware) to be using only two page tables, FTLPT and the SecPod page table. Both of them can be registered in the CR3 target-list. Therefore, legitimate context switches between the normal and secure spaces will not be trapped by the hypervisor. Our prototype uses the PAE (Physical Address Extension) mode of x86 [20], in which the top-level page table consists of four entries and can thus be copied quickly. Most modern Linux distributions by default use the PAE mode in their kernels because the NX (non-executable) bit is only available in this mode. We would like to emphasize that FTLPT is a part of the SPT pool in the secure space and thus is not accessible by the kernel. Note that we cannot use PCID (Process Context Identifier, also known as ASID) to tag the TLB – the TLB needs to be flushed during context switches because FTLPT translates addresses for many processes. Moreover, PCID is set in the CR3 register, but the CR3 target-list can only be changed by the hypervisor.

3.2.3 Page Table Update and Validation

The kernel delegates paging to the secure space to prevent unauthorized modifications to its page tables. It leverages the para-virtualized MMU interface (`pv_mmu_ops`) to forward low-level paging operations to the secure space. Figure 6 illustrates how a new level-3 (L3) page table is created and filled. When the kernel needs to allocate a new L3 page table, it sends the request to the secure space (① in Figure 6), which responds by allocating a blank L3 page table from the SPT pool and linking it to the parent shadow page table (②) . The mapping between the GPT and the SPT is then recorded in a hash table for fast indexing (③). When new page table entries are added to the GPT later, it is synchronized to the associated SPT only if no violation of memory protection is found (④). The verifier uses several hash tables for fast fact checking.

The secure space has full control over the kernel's memory protection. Any updates to shadow page tables must be vetted by the secure space. By default, the secure space enforces the normal/secure space isolation and $W \oplus X$ for the kernel:

Normal/secure space isolation: this policy prevents the (untrusted) kernel from manipulating the secure space memory. Specifically, the kernel is prohibited from mapping any of the secure space memory, except the entry and exit gates at their fixed location. For each request to change a shadow page table, SecPod checks whether the physical page belongs to the secure space and whether the virtual address overlaps with the two gates (one code page and one data page). The update is denied if either test returns true. By doing so, the kernel cannot map the secure space memory or change the gates.

Kernel $W \oplus X$: Kernel code integrity ($W \oplus X$) is essential to many security tools [28, 33, 45]. Previous virtualization-based systems leverage shadow paging in the hypervisor to protect kernel integrity. SecPod provides the same level of protection in the VM. We use a template-based approach to enforce $W \oplus X$. Specifically, modern kernels have already deployed $W \oplus X$ (without protecting the page table) [12]. The initial kernel page table could serve as a template for the kernel memory protection. For each update to the kernel mapping, SecPod only needs to compare the new memory protection against the template. Note that SecPod does not intend to externally address weaknesses in the kernel's original $W \oplus X$ implementation (it is better to root-cause and fix them in the kernel.) Enforcing $W \oplus X$ in the secure space makes it much harder to bypass. Moreover, key kernel data structures like the system call table are also write-protected for both their virtual addresses and the physical contents.

3.3 Execution Trapping

In SecPod, the kernel still has the necessary privilege to execute critical system instructions. Without constraints, this privilege could be misused to subvert the secure space, for example, by loading a malicious page table or even disabling paging. Hence, it is necessary to control the instructions executed by the guest. Simply disallowing these instructions in the kernel's binary does not

Table 1: Trapped Sensitive Instructions

Instruction	Semantics
LGDT	Load global descriptor table
LLDT	load local descriptor table
LIDT	load interrupt descriptor table
LMSW	load machine status word
MOV to CR0	write to CR0
MOV to CR4	write to CR4
MOV to CR8	write to CR8
MOV to CR3	load a new page table
WRMSR	write machine-specific registers

work because the x86 architecture has variable instruction lengths and "unintended" instructions can be created out of legitimate instructions [34]. Previous software fault isolation systems remove unintended instructions through compiler or binary transformations [46, 48]. In SecPod, we instead configure the virtualization hardware to trap these instructions, no matter whether they are benign or "unintended". Table 1 gives a (partial) list of sensitive instructions trapped by SecPod. Each of them controls some important aspects of the processor. For example, LIDT loads the interrupt descriptor table, which determines how interrupts are handled; MOV to CR0 writes to CR0, which consists of switches for many CPU operation modes (e.g., paging enable, protected mode, write-protect bits) [20]. Intercepting these instructions will not cause large performance overhead because most of them are not executed frequently after the kernel has initialized. A notable exception is the MOV to CR3 instruction that is used by the entry and exit gates for context switches. However, our design guarantees that legitimate context switches will not be trapped by the hardware (Section 3.2.2). Note that, SecPod not only protects these registers, but also the associated data structures, such as the global descriptor table and the interrupt descriptor table (Section 3.2.3).

After the hypervisor intercepts a sensitive instruction executed by the guest, it notifies the secure space of the event. This is similar to the signal delivery in traditional OSes [36]. In fact, they both implement an up-call, except that a signal is delivered from the kernel to a user process while an event in SecPod is delivered from the hypervisor to the secure space. When an instruction is intercepted, the hypervisor saves the current virtual CPU state to the virtual machine control block (VMCB) [20], and copies the saved registers to the data page of the entry gate (to provide the context of the violating instruction). The hypervisor then updates the saved instruction pointer in VMCB to the entry gate and returns to the guest. The CPU restores the guest state from the VMCB and continues its execution to the entry gate. The secure space recognizes that this is an up-call from the hypervisor and handles the violation accordingly.

4 Implementation

We have implemented a prototype of SecPod based on the popular KVM hypervisor [24]. Both the host and the guest run Linux. We added about 100 lines of source code to the hypervisor to set the CR3 target-list and trap the execution of sensitive instructions. Another 800 lines of source code were added to the guest kernel for paging delegation. The secure space has about 2,300 lines of source code. In the rest of this section, we describe this prototype in detail.

4.1 Paging Delegation

In SecPod, the guest kernel delegates its paging operations to the secure space. This gives the latter full control over the guest's memory mapping and protection. In our prototype, we leverage the Linux kernel's pvops interface to forward paging requests to the secure space. The pvops interface originates from the Xen project's efforts to create a generic para-virtualized kernel that can adapt to different hypervisors as well as the native, non-virtualized platforms. Pvops groups the key para-virtualization operations into several structures, such as pv_time_ops, pv_cpu_ops, pv_mmu_ops, pv_lock_ops, and pv_irq_ops, and substitutes native operations in the kernel with the corresponding PV operations. For example, the native x86 system uses a single MOV to CR3 instruction to load the page table. Pvops replaces it with an indirect call to the pv_mmu_ops→write_cr3 function. Each virtualization system, as well as the native platform, provides its own implementation of these functions. Particularly, functions for the native platform are simple wrappers of the original native instructions or functions. Pv_mmu_ops has all the necessary functions for SecPod to delegate paging to the secure space. For example, it has functions for write_cr3, set_pte, set_pmd, flush_tlb_kernel, etc. We only need to implement the required functions of pv_mmu_ops with the respective services provided by the secure space. In essence, this creates a MMU-only para-virtualized platform as all the other PV operations remain the same as the native platform.

Pvops replaces the native low-level hardware operations with indirect calls through the pv_xxx_ops structures. This introduces some minor but measurable performance overhead to native systems as some of these functions are frequently used by the kernel. Kernel developers have to reclaim the lost performance for native systems. Observing that these functions remain unchanged after initialization, they patch the kernel code to specialize each indirect pvops call with a direct call to the corresponding native function, and even inline simple operations like write_cr3. Therefore, we need to

replace the function pointers in `pv_mmu_ops` before the specialization. Changes to the `pv_mmu_ops` structure after the specialization will not take effect. To this end, we modify the kernel source code to set up the `pv_mmu_ops` structure early in the boot process. Because the secure space has not been initialized yet, we use a temporary page table as an in-kernel "shadow page table" and commit the page table updates to it. The temporary page table has to be statically allocated because the kernel memory allocator has not be initialized either. After the secure space is ready to run, we copy the temporary page table to a shadow page table in the secure space.

Our guest kernel is essentially a native kernel with the para-virtualized MMU. We intercepts the MMU operations during the early boot stage. However, any page tables created before that have to be manually copied to the secure space. `Swapper_pg_dir` is one such case. It is statically allocated in the kernel and serves as a master page table for the kernel address space [4]. Each process in Linux has its own user space memory mapping but shares an identical kernel part copied from `swapper_pg_dir`. No other processes except the idle task use `swapper_pg_dir` for address translation. If `swapper_pg_dir` is being loaded to `CR3` for the first time, we simply create a new shadow page table for it.

SecPod provides the entry and exit gates for the normal space to call services of the secure space (e.g., to update a page table). Because these gates are the only shared code between the two spaces, context switches have to go through them. The secure space enforces a strict normal/secure space isolation to protect these gates. The implementation details of these gates resemble that of SIM [35]. Specifically, the entry gate first saves the current CPU state to the stack and disables the interrupt with the `CLI` instruction. It then loads the SecPod page table into `CR3` to enter the secure space. The entry gate has to execute `CLI` again in the secure space in case the (untrusted) kernel has skipped the first `CLI` instruction [35]. Without a second `CLI` instruction if the first is skipped, interrupts happened in the secure space halt the (virtual) processor because the interrupt handlers are not executable in the secure space, leading to a denial-of-service attack. Finally, the entry gate loads the secure stack to the stack pointer (the `ESP` register) and calls the service handler. The exit gate performs the opposite operations in the reverse order to return to the normal space. We also fill the unused space around the entry and exit gates with `nop` instructions to avoid accidental instructions out of otherwise random bytes [34].

There is a subtle issue in the implementation of the entry and exit gates regarding TLB (translation lookaside buffer) [19]. TLB is a fast cache of the virtual to physical address translation. To access the memory, the CPU first searches the TLB for a matching virtual address. If a match is found in the TLB (a TLB hit), the resulting physical address is sent to the memory unit to access the data. If the mapping is not cached by the TLB (a TLB miss), the CPU walks the page table to translate the address and saves the result in a TLB entry for future references. Therefore, the TLB ultimately determines accessibility of the memory. Simply reloading a new page table cannot guarantee that the TLB contains fresh address translations because global pages will *not* be flushed out of the TLB during context switches (non-global pages are flushed each time a page table is loaded. For example, one way to flush all the TLB entries for the user-space is to simply reload the current page table.) The Linux kernel sets its kernel pages to global because all the processes share the same kernel memory mapping. It is thus unnecessary to flush the kernel mapping from the TLB during task switches. Note that global pages are accessible regardless of the PCID settings. Therefore using PCID cannot solve this problem.

Global pages could potentially cause serious vulnerabilities in SecPod. For example, an attacker could synthesize[1], in an executable global page, a function that loads the SecPod page table and manipulates the secure space memory. This function remains executable after entering the secure space because its mapping remains in the TLB after the context switch. On the other hand, if the secure space memory is set to global, it remains accessible after returning to the normal space. To address this pitfall, we clear the global bits in both shadow page tables and the SecPod page table, except for the entry and exit gates. By doing so, the TLB will always contain fresh address mappings after context switches, avoiding the aforementioned pitfalls. The entry and exit gates can be set to global because their memory is protected by the secure space and they do not contain enough useful gadgets for return-oriented programming [34]. TLB also allows us to batch page table updates because these updates will not take effect unless the TLB is freshened with new translations. Therefore, we can temporarily delay the page table updates until the TLB is flushed *by the kernel*, either explicitly using special instructions or implicitly through task switches. Our current prototype does not fully support this optimization yet.

4.2 Security Tool Case Study

SecPod is an extensible framework for virtualization-based security tools. A security tool running in SecPod is strictly isolated from the vulnerable kernel, but still has flexible visibility into the kernel. First, the kernel memory is mapped identically in the secure and normal spaces (but with different protection). Key kernel symbols and data structures thus can be accessed at their original locations. Second, any changes to the kernel's memory mapping

can be intercepted and adjusted, if necessary, because the kernel delegates its paging to the secure space. SecPod also has a simple loader and linker to dynamically load security tools, similar to the kernel module support.

To demonstrate the flexibility of the SecPod framework, we have build a security tool for SecPod to detect and prevent unauthorized kernel code from execution (e.g., kernel rootkits) [28, 31]. This tool is relatively simple to implement in SecPod, assuming the cryptographic hashes of benign kernel code are known. Specifically, it registers a call back function for kernel page table updates. If a new executable page is created in the kernel, it verifies whether the hash of the page belongs to the hashes of benign code pages. If so, the page is marked executable in the shadow page table. Otherwise, it has detected an attempt to execute unauthorized kernel code and raises an exception. There are a number of challenges in implementing this system. For example, when a kernel module is loaded, the kernel needs to resolve the called kernel functions (e.g., `printk`) and patches the module with the correct offsets to these functions. This effectively changes the page's hash, leading to a false positive if the hash is calculated on the modified code. We solve this problem by reversing the changes made by the kernel module loader and computing the hash based on the clean code page. After that, we restore the changes and verify that each patched function is an exposed kernel function. Many such challenges have been addressed by previous work [28, 31]. Moreover, we employ a new feature called supervisor mode execution protection (SMEP) in recent Intel processors to prevent the kernel from executing user code. The x86 architecture allows the kernel to execute user code *with the kernel privilege*. SMEP is designed to specifically address this attack. Software based defense is also available [23].

This tool provides a similar security guarantee as Patagonix [28] and NICKLE [31]. Both systems are based on the then-current virtualization technologies, the Xen hypervisor with shadow paging and hypervisors using dynamic binary translation, respectively. In contrast, the implementation based on SecPod can take advantage of nested paging. Note that detecting unauthorized code solely in the NPT is vulnerable unless all the code in the guest is authorized. Otherwise, an attacker can manipulate the GPT, which he has full control over, to map kernel code pages to the unauthorized user code.

5 Evaluation

In this section, we evaluate the security and performance of our SecPod prototype. All the experiments were conducted on a physical machine with a 2.5GHz Intel Core i5 CPU and 8GB of memory. The host system runs Ubuntu 12.04 LTS with a kernel version of 3.11.0. The guest is configured with 2GB of memory, and runs Ubuntu 12.04 LTS Server with a kernel version of 3.10.32.

5.1 Security analysis

We first evaluate the security guarantee of SecPod by analyzing how SecPod can prevent various attacks. We organize these attacks from three perspectives: memory isolation violation, instruction misuse, and malicious devices, with a focus on the first two. Malicious devices can subvert the secure space (and the hypervisor) via DMA attacks. This can be prevented using IOMMU.

Memory isolation violation: a key requirement of SecPod is to strictly isolate the security tool from the vulnerable kernel. This isolation is enabled by the synergy of SecPod's two key techniques: paging delegation and execution trapping. The first category of attacks attempts to maliciously modify the secure space memory. Because the secure space memory is not mapped in the normal space (except the entry and exit gates), the attacker cannot directly change it. Instead, the attacker has to map the secure space memory into the normal space directly or by tricking the secure space to do so. Both attacks are prevented in SecPod. *First*, the kernel delegates its paging operations to the secure space. Its own page tables are never put in effect as prevented by execution trapping. Shadow page tables in the secure space are not directly accessible by the compromised kernel either. *Second*, the kernel might request SecPod to map the secure space memory to the normal space. This is foiled by SecPod's page table update validation which enforces the normal/secure space isolation. Specifically, it disallows the normal space from mapping any physical pages of the secure space, and protects both the virtual address and the physical content of the entry and exit gates.

Instruction misuse: the second category of attacks tries to subvert the secure space by misusing existing instructions. No new code can be injected to the kernel as SecPod enforces $W \oplus X$ for the kernel, but code reuse attacks like return-oriented programming (ROP) [34] may still succeed due to the lack of control flow integrity [1]. In addition, the kernel still has the required right to execute privileged instructions. For example, it could load a crafted page table that allows manipulating the secure space. We address this type of attacks by trapping and vetting the execution of critical instructions by the kernel, such as `MOV to CR3` (Table 1). SecPod ensures that loading a page table other than the two legitimate ones will be trapped and denied. It also protects the associated data structures for instructions like `LGDT`. Since the kernel cannot load arbitrary page tables, it might try to enter the secure space with interrupts enabled. This can be achieved through the entry gate, for example, by skipping the first `CLI` and triggering an interrupt right before

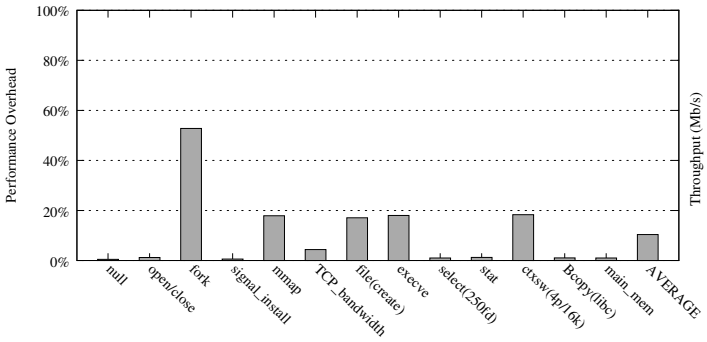


Figure 7: LMBench Overhead

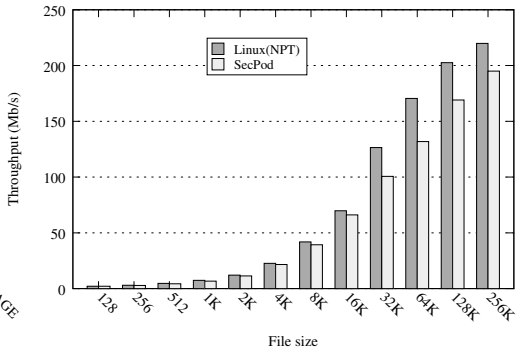


Figure 8: Apache Bench Throughput

the second `CLI`. The CPU would then execute the interrupt handler in the secure space. Our design can foil this attack because the interrupt handler is not executable as soon as the CPU switches to the secure space. Nevertheless, this might cause the virtual CPU to halt because of the non-executable interrupt handler. The attack can also be launched with the return-oriented programming (ROP). Normally, as soon as the CPU enters the secure space, the kernel code becomes non-executable and the ROP program cannot continue. However, there is a subtle case in which the ROP program switches to gadgets in the secure space upon entering it. By doing so, the program can continue running across the context switch because the attacking stack is mapped in the secure space. This attack overall is hard to use because the secure space might not contain enough useful gadgets. It can also be mitigated by applying existing ROP defenses to the secure space, such as control flow integrity [1], code randomization [26], and systematic removal of gadgets [27].

Synthetic attack: to further validate the security of SecPod, we create a synthetic kernel rootkit that hooks the system call table to intercept system calls like `sys_read` and `sys_mkdir`. Our experimental security tool can detect the loading of the malicious rootkit because its hash is not in the list of hashes of benign code pages. Even without this tool, SecPod can detect the rootkit's attempts to modify the (read-only) system call table – the rootkit calls a kernel function to make the syscall table writable. This request is forwarded to the secure space and subsequently denied because the secure space does not allow the syscall table to be changed.

5.2 Performance Evaluation

To evaluate the performance of SecPod, we experimented with micro-benchmarks and system benchmarks. The former measures SecPod's impact to fine-grained operations (e.g., system calls), and the latter measures the overall system performance under SecPod. All the experiments were repeated 10 times and the average results are reported here. The deviation of these experiments is negligible. We compare the performance of SecPod with that of an unmodified VM backed by the nested paging (the baseline). SecPod's VM is also backed by the nested paging. However, its paging operations are expected to be less efficient than the baseline because they are delegated to the secure space. Even though we did not compare the performance of SecPod to that of the VMs backed by shadow paging, previous benchmarks demonstrate that Intel EPT provides substantial performance gains over shadow paging for most tested benchmarks. For example, Intel EPT can achieve an acceleration of up to 48% for MMU-intensive benchmarks [42].

5.2.1 Micro-benchmarks

Figure 7 shows the performance overhead of SecPod for LMBench, a set of benchmarks to measure the system call performance. Our prototype incurs less than 5% overhead for most of the system calls LMBench tests, such as `open`, `close`, `signal_install`, and `stat`. These system calls do not contain operations that require services from the secure space. Consequently, the impact of SecPod over these system calls is minimal. The performance degrade is probably caused by normal task switches (of other processes) during the tests. On the other hand, system calls that involve page table operations suffer most. Particularly, `fork` has the highest overhead (52.8%), followed by `execve`, `mmap`, `file creation`, and `context switch` (all at around 17%). Most of these system calls involve heavy page table operations. For example, the `fork` system call creates a child process that duplicates the parent process's address space (with copy-on-write) [36], and each task switch in SecPod requires an extra loading of the SecPod page table (Section 3.2.2). Our current prototype does not yet support the batch-update of the page table, an optimization that could help reduce the overhead of these cases, especially for the `fork` system call. On average, SecPod introduces about 10% performance overhead for LMBench.

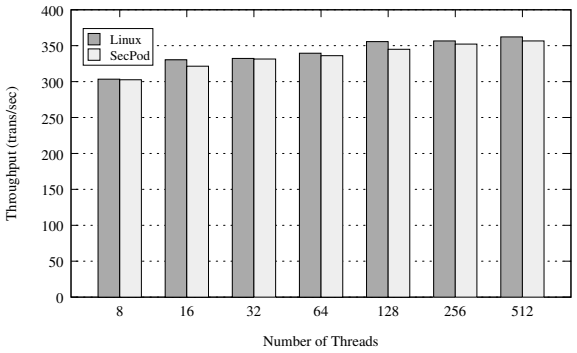


Figure 9: Throughput of SysBench FileIO

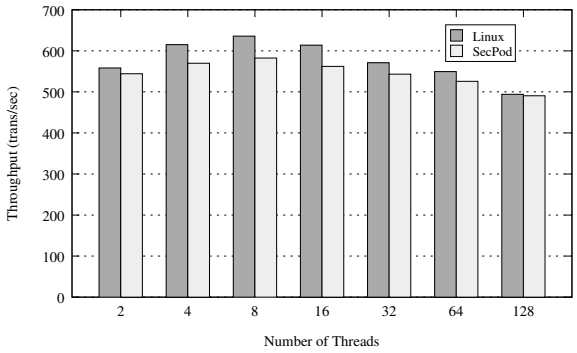


Figure 10: Throughput of SysBench OLTP

5.2.2 Application Benchmarks

To measure SecPod's impact on the overall system performance, we experimented with two benchmarks, ApacheBench and SysBench. ApacheBench is a program to measure how fast the system can process web traffic. In this experiment, we run the Apache server (2.2.22) in the VM, and ApacheBench on another physical machine with a similar hardware configuration. Figure 8 shows the throughput of the Apache server with regard to different file sizes (from 128 bytes to 256KB). Each file was generated by collecting random data from the `/dev/random` device. For file sizes up to 16KB, the overhead of SecPod is less than 9% and increases to about 16% for 128KB files and 11% for 256KB files. When the file size increases, the kernel needs to update the page table more frequently to accommodate frequent file accesses, leading to a relatively high performance overhead. The average performance overhead for ApacheBench is about 9%.

SysBench is a suite of multi-threaded benchmarks to evaluate the performance of a database system under intensive workloads. We use SysBench to measure SecPod's impacts on the file I/O and the MySQL processing. Both experiments are repeated with many different numbers of threads. In the file I/O experiment, we measure the throughput using 128 files (1GB in total) and a block size of 16KB. The results are shown in Figure 9. The largest overhead is 3.25%. We also measure the MySQL performance with SysBench's online transaction processing (OLTP) benchmark. Specifically, we build a MySQL database with 1,000,000 entries and query the database using various numbers of threads. The results are shown in Figure 10. The performance loss is in the range of 2% to 14% with an average of 5%. Interestingly, the performance overhead reduces as the number of threads exceeds 32. This is probably because the performance loss caused by the contention over shared resources outweighs that of SecPod starting at that point. This is reflected in the decreasing numbers of transactions processed per second when more than 32 threads are used.

6 Related Work

Virtualization-based Security: the first category of the related work is a long stream of virtualization-based security systems with diverse focuses, such as malware analysis [13], virtual honeypot [21], kernel rootkit detection and prevention [27, 32] etc. In particular, virtualization has been applied often in the context of virtual machine introspection. Livewire pioneers the concept of "out-of-VM" introspection to understand the in-VM states and activities by parsing the raw VM resources [17]. Semantic gap is one of the main challenges for VMI systems because VMI aims at semantically inferring the in-VM activities and states from the raw VM data (e.g., memory, disk). A number of recent systems try to address this challenge from different perspectives [14, 16, 22, 35]. For example, Virtuoso [14] can effectively automate the process of building introspection-based security tools. SIM is the most closely related system. It firstly leverages the `CR3` target-list to effectively and efficiently turn out-of-VM monitoring in-VM. SIM is a monitoring framework while SecPod targets at supporting generic virtualization-based systems. Particularly, SecPod creates a trusted execution environment for the security tool by combining two key techniques, paging delegation and execution trapping. In addition, SecPod uses the `CR3` target-list differently to support the nested paging (Section 3.2.2). VMI systems can be integrated with and benefit from SecPod's code integrity guarantee and fine-grained page table monitoring.

Virtualization is also a popular choice of platforms to enhance the kernel or application security [6, 28, 31, 38, 45]. For example, Overshadow is designed to protect the secrecy of the user data even if the kernel is completely compromised [6]. Patagonix protects the kernel code integrity through virtualization-based code identification [28]. HookSafe addresses the protection granularity problem through systematic hook redirection [45]. Most of these systems require a reliable kernel code integrity. Otherwise, an attacker could subvert their pro-

tection by injecting malicious code. SecPod is an ideal platform for these systems. Security tools in SecPod are strictly isolated from the vulnerable kernel, but still have the visibility of an in-kernel tool. As a proof-of-concept, we implemented a security tool based on SecPod to prevent the unauthorized code from executing in the kernel. This provides a security guarantee similar to Patagonix [28] and NICKLE [31] (Section 4.2).

Virtualization-based systems, including SecPod, assume that the hypervisor is trusted due to its smaller code base and attack surface. However, the bloated code base of modern hypervisors and recent attacks put this assumption into question. There have been a series of recent efforts in protecting the hypervisor integrity, via formal verification [25, 30], security enhancements [44], and size reduction and disaggregation [7, 29, 40]. These systems can be naturally integrated with SecPod to provide a strong foundation of security.

Kernel/User Application Security: the second category of related work includes a large number of research efforts in the kernel and user application security. Address space layout randomization (ASLR) [18] and data execution prevention (DEP) [12] are two popular exploit mitigation mechanisms in modern kernels. These kernel-level protection schemes suffer from the pitfall that the page table is not protected from exploits. SecPod reliably enforces DEP for the kernel. ASLR and DEP could be bypassed mainly by return-oriented-programming (ROP). Control flow integrity is an effective defense against most control flow attacks, including ROP, by mandating that run-time control flow must follow the program's control flow graph [1, 49, 50]. Recent efforts in CFI has significantly improved its performance and compatibility with commercial off-the-shelf applications. DEP is a prerequisite of CFI. Most of the previous CFI systems target user applications. They rely on the kernel to provide the necessary memory protection of the code and read-only data. Recent efforts to adapt CFI to the kernel turn to virtualization for essential supports [8]. For example, KCoFI [8] leverages the Secure Virtual Architecture [9] to interpose the software and hardware interactions. All software, including the kernel, is compiled to the virtual instruction set of SVA. Kernel CFI can also be support by SecPod as it provides both strong isolation and reliable memory protection for security tools. There is also a series of prior efforts in implementing software fault isolation (SFI) [15, 43, 48]. SFI aims at confining untrusted code in a host application. For example, Native Client [48] uses two layers of sandboxes to safely run untrusted native plugins in a web browser. SFI technologies have been utilized to isolate untrusted device drivers in the kernel [15, 38, 39].

TZ-RKP [2], HyperSafe [44], and nested kernel [11] are three closely related systems. TZ-RKP leverages the ARM TrustZone to protect the kernel running in the normal world. Specifically, it instruments the kernel to prevent it from executing certain privileged instructions or updating page tables. These operations instead must be handled by the secure world. Recently, Intel introduced a security enclave called Software Guard Extension (SGX). However, the instrumentation-based instruction access control of TZ-RKP is not directly applicable to the x86 architecture because x86 has variable instruction lengths and thus unintended privileged instructions can be created out of the existing ones [34]. This problem can be solved by adopting the techniques of NaCl [48]. HyperSafe write-protects the hypervisor page table and uses the x86 write-protect (WP) bit to allow benign page table updates. It further enforces the control flow integrity [1] to prevent that from being bypassed. Nested kernel similarly protects page tables for the OS kernel, but enforces the kernel code integrity and removes unintended privileged instructions from the kernel code (instead of enforcing CFI). SecPod also controls the guest page table updates though paging delegation, but its design revolves around the goal to provide security tools with an extensible framework that is not only compatible with the recent virtualization hardware, but also allows them to intercept key events in the guest kernel. For example, the separation of the normal and secure spaces isolates security tools from the untrusted kernel and simultaneously enables an easy access to the kernel data.

7 Summary

We have presented the design, implementation, and evaluation of SecPod, an extensible framework for virtualization-based security systems. SecPod provides a trusted execution environment for security tools. They are not only strictly isolated from the vulnerable kernel, but also have full visibility into it. Particularly, any updates to the guest's page tables can be intercepted and regulated by these tools, allowing the fine-grained control over the guest kernel's memory protection. By using the in-VM shadow paging, SecPod is fully compatible with the recent advances in the hardware virtualization support, particularly the nested paging.

Acknowledgments: we would like to thank our shepherd, Andy Tucker, and the anonymous reviewers for their insightful comments that greatly helped improve the presentation of this paper. This work is a part of the project supported by US National Science Foundation (NSF) under Grant 1453020. The first author was supported by the grants from CSC (201306280080), NSFC (61272460), and RFDP (20120201110010) to visit Florida State University. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[5] P. M. Chen and B. D. Noble. When Virtual Is Better Than Real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, 2001.

[6] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[7] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, October 2011.

[8] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[9] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007.

[10] CVE Database. Common Vulnerabilities and Exposures Database. `http://www.cvedetails.com/`.

[11] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[12] Data Execution Prevention. `http://en.wikipedia.org/wiki/Data_Execution_Prevention`.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[14] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on*, 2011.

[15] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006.

[16] Y. Fu and Z. Lin. Space Traveling Across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.

[17] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed System Security Symposium*, Febuary 2003.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, 2012.

[19] J. L. Hennessy and D. A. Patterson. *Computer ARchitecture: a Quantitative Approach*. Morgan Kaufmann, 2012.

[20] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual*, Feb 2014.

[21] X. Jiang and X. Wang. "Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, 2007.

[22] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-based "Out-Of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[23] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, 2012.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, June 2007.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[26] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, 2014.

[27] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with "Return-less" Kernels. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference*, April 2010.

[28] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th USENIX Security Symposium*, July 2008.

[29] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2008.

[30] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 2013.

[31] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th Recent Advances in Intrusion Detection*, September 2008.

[32] R. Riley, X. Jiang, and D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th ACM SIGOPS EuroSys Conference*, April 2009.

[33] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM ACM Symposium on Operating Systems Principles*, October 2007.

[34] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[35] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[36] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2012.

[37] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process Out-grafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[38] A. Srivastava and J. Giffin. Efficient Monitoring of Untrusted Kernel-Mode Execution. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, February 2011.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, October 2003.

[40] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, October 2011.

[41] Trusted Boot project. Trusted Boot. http://tboot.sourceforge.net/.

[42] VMware. Performance Evaluation of Intel EPT Hardware Assist. https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[43] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium On Operating System Principles*, December 1993.

[44] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Contr ol-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, May 2010.

[45] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[46] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM european conference on Computer Systems*, April 2012.

[47] Wikipedia. DMA Attack. http://en.wikipedia.org/wiki/DMA_attack.

[48] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, May 2009.

[49] B. Zeng, G. Tan, and U. Erlingsson. Strato: A Retargetable Framework for Low-level Inlined-reference Monitors. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, 2013.

[50] B. Zeng, G. Tan, and G. Morrisett. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

Notes

[1]This can be achieved with the return-oriented programming since SecPod prevents code injection to the kernel.

Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications

Jun Wang, Xi Xiong[†1], Peng Liu
Pennsylvania State University, Facebook Inc.[†]

Abstract

Threads in a multithreaded process share the same address space and thus are implicitly assumed to be mutually trusted. However, one (compromised) thread attacking another is a real world threat. It remains challenging to achieve privilege separation for multithreaded applications so that the compromise or malfunction of one thread does not lead to data contamination or data leakage of other threads.

The Arbiter system proposed in this paper explores the solution space. In particular, we find that page table protection bits can be leveraged to do efficient reference monitoring if data objects with the same accessibility stay in the same page. We design and implement Arbiter which consists of a new memory allocation mechanism, a policy manager, and a set of APIs. Programmers specify security policy through annotating the source code. We apply Arbiter to three applications, an in-memory key/-value store, a web server, and a userspace file system, and show how they can benefit from Arbiter in terms of security. Our experiments on the three applications show that Arbiter reduces application throughput by less than 10% and increases CPU utilization by 1.37-1.55×.

1 Introduction

While multithreaded programming brings clear advantages over multiprocessed programming, the classic multithreaded programming model has an inherent security limitation, that is, it implicitly assumes that all the threads inside a process are mutually trusted. This is reflected by the fact that all the threads run in the same address space and thus share the same privilege to access resources, especially data.

However, one thread attacking another thread of the same application process is a real world threat. Here are a few examples: (1) For the multithreaded in-memory key/value store Memcached [9], it has been shown that many large public websites had left it open to arbitrary access from Internet [2], making it possible to connect to (a worker thread of) such a server, dump and overwrite cache data belonging to other threads [7]. In addition, vulnerabilities [11, 10] could be exploited by an adversary (*e.g.,* buffer overflow attack via CVE-2009-2415) so that the compromised worker thread can arbitrarily access data privately owned by other threads. (2) For the multithreaded web server Cherokee [3], an attacker could exploit certain vulnerabilities (*e.g.,* format string CVE-2004-1097) to inject shellcode and thus access the private data of another connection served by a different thread. Meanwhile, logic bugs (*e.g.,* Heartbleed [8]) might exist so that an attacker can fool a thread to steal private data belonging to other threads. (3) For the multithreaded userspace file system FUSE [6], logic flaws or vulnerabilities might also allow one user to read a buffer that contains private data of another user, which violates the access control policy. This is especially critical for encrypted file systems built upon FUSE (*e.g.,* EncFS [5]), wherein data can be stored as cleartext in memory and a malicious user could enjoy a much easier and more elegant way to crack encrypted files than brute force.

A common characteristic of the above applications is that they may concurrently serve different users or clients, which represent distinct principals that usually do not fully trust each other. This characteristic directly contradicts the "threads-are-mutually-trusted" assumption. Therefore, a fundamental multithreaded application security problem arises, that is, *how to retrofit the classic multithreaded programming model so that the "threads-are-mutually-trusted" assumption can be properly relaxed?* In other words, could different principal threads have different privileges to access shared data objects so that the compromise or malfunction of one thread does not lead to data contamination or data leakage of another thread?

1.1 Prior Work and Our Motivation

From a programmer's point of view, we identify two kinds of privilege separation problems. The first problem is to split a monolithic application into least-privilege compartments. For example, an SSH server only requires root privilege for its *monitor* (listening to a port and performing authentication), rather than the *slave* (processing user commands). Since the two parts are usually closely coupled, developers in the old days simply put the two into one program. Due to the emergence of buffer overflow and other relevant attacks against the root privileged part, however, this monolithic program design is no longer appropriate. Separation of the two parts into different privileged processes with IPC mechanisms in between (*e.g.,* via pipes) becomes a more appropriate approach. Actually, OpenSSH has already adopted this approach.

The second problem is to do fine-grained privilege separation in multithreaded applications. As introduced earlier, threads in a multithreaded program were implicitly assumed to be mutually trusted. However, the evolving

[1] work was done while this author was at Pennsylvania State University.

of multithreaded applications tends to break this assumption by concurrently serving different principals. Usually the principals do not fully trust one another.

The first problem has been studied for many years [29, 27, 19, 15, 14]. Provos *et al.* [27] pioneered the methodology and design of privilege separation. Privtrans [15] can automatically partition a program into a privileged monitor and an unprivileged slave. Wedge [14] combines privilege separation with capabilities to do finer-grained partitioning. For the second one, however, there are no systematic research investigations that we are aware of.

This paper focuses on the second privilege separation problem. Our goal is to apply least privilege principle on (shared) data objects so that a data object can be read-writable, read-only, or inaccessible to different threads at the same time and, more importantly, to require minimum retrofitting effort from programmers. First of all, let's look at existing mechanisms to see whether they can be applied to solve this problem.

1) Process isolation. Process isolation is the essential idea behind existing approaches to the first privilege separation problem. OpenSSH [27] and Privtrans [15] leverage process address space isolation while using IPC to make the privileged part and the unprivileged part work together. However, neither of them handles data object granularity. In addition, when there are many principal threads, IPC might become very inefficient. Wedge [14] advances process isolation with new ideas. It creates compartments with default-deny semantics and maps shared data objects into appropriate compartments. However, Wedge is proposed to address the first privilege separation problem, which has very different nature from the problem we consider, as shown in Table 1. Due to these differences, Wedge's all-or-nothing privilege model with default-deny semantic is not very applicable to a multithreaded program, wherein threads by default share lots of resources. To apply Wedge on our problem, one still needs to address the challenges considered in this paper.

Manually retrofitting a multithreaded program to use multiple processes is possible. However, commodity shared memory mechanisms, such as `shm_open` and `mmap`, do not allow one thread to specify the access right of another thread on the shared memory. Alternatively, designing a sophisticated one-on-one message passing scheme (*e.g.,* using Unix socket) can enforce more control on data. However, the programming difficulty and complexity (*e.g.,* process synchronization, policy handling and checking) could be much higher and thus requires lots of retrofitting effort from programmers.

Another notable idea is to redesign an application from scratch using a multi-process architecture, as what is done in Chrome [4]. However, one of our quick survey reveals that over 80% of existing web servers are multithreaded. It is impractical to redesign all those applications that are already multithreaded.

1st PS Problem (OpenSSH [27], Privtrans [15], Wedge [14], *etc.*)	**2nd PS Problem** (Arbiter)
Sequential invocation of compartments with different privileges	Concurrent execution
Only privileged process/thread can access sensitive data	Data shared among different (unprivileged) principal threads
Static capability policy	Dynamic (label) policy

Table 1: Different assumptions of 1st and 2nd privilege separation (PS) problem

2) Software fault isolation. Address space isolation puts each process into a protection domain, but does not do finer-grained isolation inside an address space. Software fault isolation [30, 17] did an innovative work on making a segment of address space as a protection domain by using software approaches like a compiler. Nevertheless, it is difficult for SFI to map program data objects (*e.g.,* array) into a protection domain: address-based confinement and static instrumentation cannot easily deal with dynamically allocated data. LXFI [24] instruments `kmalloc` so that the principal and address information of dynamic kernel data objects are made aware to the reference monitor. However, this is done only to kernel modules and kernel data. In addition, LXFI focuses on integrity and does not check memory reads due to performance reasons. However, our goal is to prevent both unauthorized reads and writes. Therefore, we need to catch invalid reads as well.

3) Other related mechanisms. We investigate four additional types of related mechanisms to see whether they can handle our problem. (a) OS abstraction level access control has been extensively studied (*e.g.,* SELinux [23], AppArmor [1], Capsicum [31]). However, these mechanisms treat a process/thread as an atomic unit and do not deal with data objects "inside" a process. So a granularity gap exists between these techniques and our goal. (b) HiStar [33] is a from-scratch OS design of decentralized information flow control (DIFC). Perhaps HiStar can meet our goal of privilege separation on data objects. However, HiStar does not apply to commodity systems. Besides, to use HiStar to achieve our goal, there still needs to be a major change in the programming paradigm. Flume [20] implements DIFC in Linux. However, it focuses on OS-level abstractions such as processes, files, and sockets and thus does not address the privilege separation problem at data object granularity within a multithreaded program. It can be complementary to the approach proposed in this paper. (c) With the tagged memory in Loki [34] or the permission table lookup mechanism in MMP [32], as new features to the CPU, access to each individual memory word can be checked. Both methods can enforce privilege separation policy on data objects. However, they require architectural changes to commodity CPUs. (d) Language-based

solutions, such as Jif [26], Joe-E [25], and Aeolus [16] can realize information flow control and least privilege at the granularity of program data object. However, they need to rely on type-safe languages like Java. As a result, programmers have to rewrite legacy applications not originally developed in a type-safe language.

1.2 Challenges and Our Approach

We would like to solve this problem in a new way based on this insight: *we find that page table protection bits can be leveraged to do efficient reference monitoring, if the privilege separation policy can be mapped to those protection bits.* We find that this mapping is possible through a few new kernel primitives and a tailored memory management library. However, doing so still introduces three major challenges:

- **Mapping Challenge (C1)** In the current multithreaded programming paradigm, all the threads in the same process share one set of page tables. This convention, however, would disable the needed mapping from privilege separation policy to protection bits.
- **Allocation Challenge (C2)** To make the protection bits work, data objects that demand distinct privileges cannot be simply allocated onto the same page because this will result in the same access rights. Existing memory management algorithms have difficulty meeting such a requirement because they were not designed to enforce privilege separation.
- **Retrofitting Challenge (C3)** It is challenging to minimize programmers' retrofitting effort to communicate complex privilege separation policies with the underlying system without modifying the source code drastically.

We present Arbiter to address the above challenges. To address the mapping challenge (C1), we associate a separate page table to each thread and create a new memory segment named Arbiter Secure Memory Segment (ASMS) for *all* threads. ASMS maps the shared data objects onto the same set of physical pages and set the page table permission bits according to the privilege separation policy. To deal with the allocation challenge (C2), we design a new memory allocation mechanism to achieve privilege separation at data-object granularity on ASMS. To resolve the retrofitting challenge (C3), we provide a label-based security model and a set of APIs for programmers to make source-level annotations to express privilege separation policy. We design and implement Arbiter based on Linux, including a new memory allocation mechanism, a policy manager, and a set of kernel primitives.

We port three types of multithreaded applications to Arbiter, *i.e.,* an in-memory key/value store (Memcached), a web server (Cherokee), and a userspace file system (FUSE), and show how they can benefit from Arbiter in terms of security. Our own experiences indicate that porting programs to Arbiter is a smooth procedure. The changes to the program source code is 0.5% LOC on average. Regarding performance, our experiments show that the runtime throughput reduction is below 10% and CPU utilization increase is 1.37-1.55×.

2 Overview

2.1 Motivating Examples

Programmers have both *intended privilege separation* and *intended sharing* of data objects when writing multithreaded programs. We classify these intentions into three categories.

- **Category 1**: A data object is intended to be exclusively accessed by its creator thread.

Figure 1(a) shows the request processing code snippet from Cherokee. The data object `buf` is allocated by a worker thread and then used to store the incoming packet. Therefore, this data object belongs to that particular worker thread and other worker threads are not supposed to access it.

- **Category 2**: A data object is intended to be accessed by a subset of threads.

Figure 1(b) and 1(c) show the connection handling code snippets from Memcached. The main thread receives a network request, allocates a data object `item` to store the connection information, selects a worker thread and then pushes the `item` into the thread's connection queue. The worker thread wakes up, dequeues the connection information and handles the request. Ideally, the data object `item` is only intended to be accessed by the main thread and the particular worker thread, excluding any other worker thread.

- **Category 3**: A data object is intended to be shared among all the threads.

This data sharing intention is commonly seen, especially on metadata. For instance, the `struct cherokee_server` and the `struct fuse` store the global configurations of Cherokee and FUSE, respectively, and are intended to be accessible to all the threads.

Overall, Category 1 and 2 are two very representative privilege separation intentions. Unfortunately, there is actually no such enforcement in real world execution environments. Only the intention in Category 3 has been taken care. We propose Arbiter, a general purpose mechanism so that every category is respected.

2.2 Threat Model

We consider two types of threats. First, some threads could get compromised by malicious requests (*e.g.,* buffer overflow attacks, shellcode injection, return-to-libc attacks, ROP attacks). Second, application has certain logic bugs (*a.k.a.* logic vulnerabilities [18] or logic flaws [22]). For example, the logic bug exploited by HeartBleed [8] can potentially lead to a buffer overread attack,

```
process_active_connections(cherokee_thread_t *thd) {
        ...
        buf = (char *) malloc (size);
        ...
        len = recv (SOCKET_FD(socket), buf, buf_size, 0);
        ...
}
```

(a) Cherokee-1.2.2

```
void dispatch_conn_new(...) {
        ...
        CQ_ITEM *item = malloc(sizeof(CQ_ITEM));
        ...
        cq_push(thread->new_conn_queue, item);
        ...
}
```

(b) Memcached-1.4.13 Main thread

```
static void *worker_libevent(...) {
        ...
        item = cq_pop(me->new_conn_queue);
        ...
}
```

(c) Memcached-1.4.13 Worker thread

Figure 1: Motivating examples

which allows an attacker to steal sensitive information of other users from a web server. In reality, both threats can lead to data leakage and data contamination of a victim thread, which usually result in the compromise of end user's data secrecy and integrity. Besides, we assume that the application is already properly confined by well-defined OS level access control policies (*e.g.,* which files the application can access) using SELinux, AppArmor, *etc.* We also assume that the kernel is inside TCB. The fact that the kernel could be compromised is orthogonal to the problem we aim to solve.

2.3 Problem Statement

How to deal with the two types of threats through a generic data object-level privilege separation mechanism so that all of the three categories of how a data object is intended to be accessed by threads can get respected?

2.4 System Architecture

Figure 2 shows the architecture of our system. In Arbiter, threads are created in a new way, resulting in what we call *Arbiter threads*. Arbiter threads resemble traditional threads in almost every aspect such as shared code segment (`.text`), data segment (`.data`,`.bss`), and open files, but they have a new dynamically allocated memory segment ASMS. To give threads different permissions to access the same data object, we maintain a separate page table for each thread and maps the shared data objects on ASMS to the same set of physical pages. To set the needed permissions, protection bits inside each page table will be set up according to the privilege separation policy. In kernel, these are realized by the ASMS Management component, including system call code plus a set of kernel functions, and the corresponding additions to the page fault handling routine. Due to ASMS, two objects with different *accessibility* will be allocated on two different pages. By accessibility, we mean which threads can access an object in what way. However, many pages

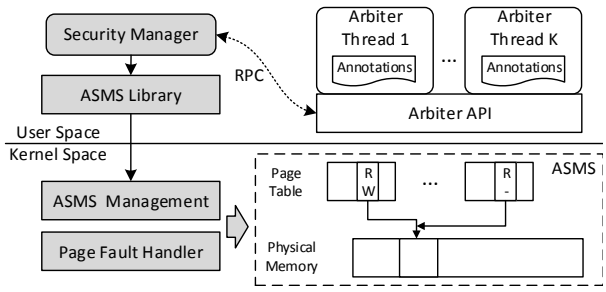


Figure 2: System architecture. Shaded parts indicate Arbiter's trusted computing base (TCB).

	Main Thread	Thread A	Thread B
A's Data `buf`	–	RW	–
B's Data `buf`	–	–	RW
Shared Data `item`	RW	R	R

Table 2: Accessibility generated from Figure 1

could end up with being half empty by doing so. Our solution is to leverage homogeneity, that is, objects with the same accessibility are put into the same page. Such memory allocation is achieved by the ASMS Library.

There are three things a thread needs to go through Arbiter: (1) memory allocation and deallocation, (2) thread creation, and (3) policy configuration. For security purpose, Arbiter threads delegate these operations to the Security Manager running in a different address space via remote procedure calls (RPC).

To specify security policy, programmers will need to make annotations to the source code via the Arbiter API according to our label-based security model. The Security Manager will figure out the permissions at runtime and the page table protection bits will be set up properly before the corresponding data object is accessed by an Arbiter thread.

3 Design

3.1 Accessibility

In our system, accessibility means which threads can access an object in what way. Conceptually, we need to map the aforementioned three categories of intentions onto accessibility before we can enforce fine-grained privilege separation.

Table 2 shows a formally defined accessibility generated from the motivating examples in Figure 1. Accessibility is defined in terms of a set of threads. Given a set of threads $\{th_1, \cdots, th_k\}$, the accessibility of data object x is defined as a *vector* of k elements. For example, the accessibility vector of A's data `buf` is $< \emptyset, RW, \emptyset >$. Two data objects have the same accessibility if and only if they have the same vector in term of all of the k threads.

3.2 Design Goal

At a high level, our goal is that through Arbiter the accessibility originated from the privilege separation intentions can be enforced. This goal boils down to the following three design requirements. (1) From a system's perspective, separate page tables are required in order to enforce accessibility vectors and a synchronized virtual-to-physical mapping is required to make such separation transparent to the threads. (2) From a program's perspective, a smart memory allocation strategy is required in

order to bridge the granularity gap between page-level protection and individual data objects and do it in an efficient way, for which we propose the idea of "same accessibility, same page". These two requirements lead to the kernel-level and user-level design of ASMS (§3.3). (3) From a programmer's perspective, it is important to correctly code accessibility in the program without changing the program drastically. We create a label-based security model and a set of APIs for this purpose (§3.4). In addition, how Arbiter converts accessibility into protection bits is introduced in §3.5. §3.6 discusses the thread creation and context switch issues incurred by our design.

3.3 ASMS Mechanism

Kernel Memory Region Management. To grant threads with different permissions to the shared memory, our initial thought was to leverage the file system access control mechanism `user/group/others` to `mmap` files with allowed open modes so as to realize different access rights. Since this method has to assign a unique UID for each principal thread, however, it would mess up the original file access permission configurations. In addition, `mmap` cannot automatically do memory allocation and configuration for multiple sets of page tables in a single invocation.

We design a new memory abstraction called Arbiter Secure Memory Segment (ASMS) to achieve efficient privilege separation. ASMS is a special memory segment compared to other segments like code, data, stack, heap, *etc.* The difference is that when creating or destroying ASMS memory regions for a calling thread, the operation will also be propagated to all the other Arbiter threads. In other words, ASMS has a synchronized virtual-to-physical memory mapping for all the Arbiter threads, yet the access permissions (page protection bits `Present` and `Read/Write`) could be different. Furthermore, only the Security Manager has the privilege of controlling ASMS. Arbiter threads, in contrast, cannot directly allocate/deallocate memory on ASMS. Neither can they modify their access rights of ASMS data objects on their own.

User-level Memory Management Library. A granularity gap exists between page-level protection (enabled by the per-page protection bits) and individual program data objects. Data objects demanding distinct accessibility can no longer be allocated on the same page. To this end, existing memory allocation algorithms (*e.g.,* dlmalloc [21]) cannot directly work for ASMS. An intuitive solution is to allocate one page per data object. However, this is not preferable mainly because a huge amount of memory will be wasted if the sizes of data objects are much smaller than the page size.

We design a special memory allocation mechanism for ASMS: *permission-oriented* allocation. The key idea is to put data objects with identical accessibility onto the same page, or "same accessibility, same page". When we allocate memory for a new data object x with accessibility vector v, we search for a page containing data objects with the same vector v and put x into that page. If that page is full, we search for another candidate page. If all candidate pages are full, we allocate a new page and put x into it. In practice, we allocate from the system one memory *block* instead of one page per time so as to save the number of system calls. Here a memory block means a contiguous memory area containing multiple pages. Figure 3 demonstrates this idea (further details in §4.1). In this way, both memory waste and performance overhead can be reduced.

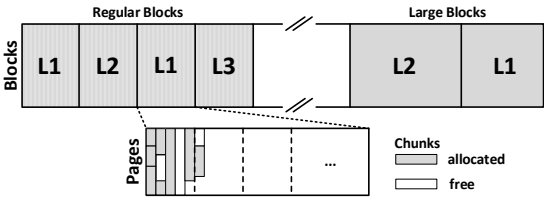


Figure 3: A typical memory layout of ASMS. L1/L2/L3 indicate different accessibility.

3.4 Label-based Security Model

To accommodate programmers' privilege separation intentions, we need a security model for specifying and enforcing accessibility vectors. Our initial attempt was to load the entire accessibility table into the Security Manager as an access control list (ACL) so that it can check and determine each thread's permission for a data object. However, to regulate each thread's capability of making allocation requests (*e.g.,* thread A is not allowed to allocate objects that are accessible by everyone) and to deal with dynamic policies (*e.g.,* thread A first grants thread B permission and later on revokes it), ACL is insufficient and further mechanisms must be employed. It is desirable to have a unified and flexible security model.

To achieve unification and flexibility, we develop a label-based security model wherein threads and data objects are associated with labels so that data access permissions and allocation capabilities can be dynamically derived and enforced. Essentially, it is a special form of "encoding" of the accessibility table. The basic notions and rules follow existing dynamic information flow (DIFC) models [33, 28] with a few adaptations. It should be noted that Arbiter itself is not a DIFC system (see §7 for more discussion).

We use *labels* to describe the security properties of principal threads and data objects. A *label L* is a set that consists of *secrecy categories* and/or *integrity categories*. For a data object, secrecy categories and integrity categories help to protect its secrecy and integrity, respectively. For a thread, the possession of a secrecy category ($*_r$, where $*$ represents the name of a category) denotes its `read` permission to data objects protected by that category; likewise, an integrity category ($*_w$) grants a thread

the corresponding `write` permission. Meanwhile, we use the notion *ownership O* to mark a thread's privilege to bypass security checks on specific categories. A thread that creates a category also owns that category (*i.e.,* has the ownership). Different from threads, data objects do not have ownership.

We define the rules that govern threads' permissions and activities as follows:

RULE 1 – Data Flow: We use $L_A \sqsubseteq L_B$, to denote that data can flow from A to B (A and B represent threads or data objects). This means: 1) every secrecy category in A is present in B; and 2) every integrity category in B is present in A. If the bypassing property of ownership is considered, a thread T can read object A iff: $L_A - O_T \sqsubseteq L_T - O_T$, which can be written as: $L_A \sqsubseteq_{O_T} L_T$. Similarly, thread T can write object A iff: $L_T \sqsubseteq_{O_T} L_A$.

RULE 2 – Thread Creation: Thread creation is another way of data flow and privilege inheritance. Therefore, a thread T is allowed to create a new thread with label L and ownership O iff: $L_T \sqsubseteq_{O_T} L, O \subseteq O_T$. The new thread is not allowed to modify its own labels.

RULE 3 – Memory Allocation: Memory allocation also implies that data flows from a thread to the allocated memory. As the result, a thread T can get a memory object allocated on ASMS with label L iff: $L_T \sqsubseteq_{O_T} L$.

Therefore, one could make the following label assignment to realize the accessibility vectors in Table 2. For instance, Thread *A* can read but not write the Shared Data *item* because of $L_{item} \sqsubseteq_{O_A} L_A$ and $L_A \not\sqsubseteq_{O_A} L_{item}$. Neither can Thread *A* create a thread with the Main Thread's privilege ($O_A \not\subseteq O_{Main}$) nor allocate a forged data *item* ($L_A \not\sqsubseteq_{O_A} L_{item}$). As such, our model unifies permission and capability.

Thread	**Main**	**A**	**B**
label	{}	{mr}	{mr}
ownership	{mr,mw}	{ar,aw}	{br,bw}

Data	**A's Data** `buf`	**B's Data** `buf`	**Shared Data** `item`
label	{ar,aw}	{br,bw}	{mr,mw}

The labels are attached by a programmer to the corresponding threads or data objects through annotating the source code via Arbiter API. Appendix A.1 presents a list of Arbiter API.

3.5 Protection Bits Generation

The Security Manager is responsible for converting labels to page table protection bits. The Security Manager maintains a real-time registry containing label information of every thread and every ASMS memory block. The conversion happens in two occasions: memory allocation and thread creation. First, whenever a thread wants to allocate memory with certain labels, the Security Manager determines the permissions for *every* thread by checking our label model, and then invokes our system calls to construct and configure ASMS memory regions accordingly. Second, when a new thread is created, the Security Manager walks through *every* ASMS memory block, determines the allowed permissions, and initializes the ASMS correspondingly.

Note that in Linux a page table entry is not established until the data on that page is actually accessed. Therefore, the page fault handler will eventually further convert the permissions stored in the flags of ASMS memory regions into page table protection bits (further details in §4.2). As the result, before a data object is accessed by any thread, the page table protection bits would have been set up properly.

3.6 Thread Creation and Context Switch

We identify two options to create an Arbiter thread. Option 1: Conceptually, one can create a new address space for every new Arbiter thread, reconfigure ASMS permissions, and disable copy-on-write for all the other memory segments to retain memory sharing. In this case, although the context switch between two Arbiter threads will lead to TLB flush (which is just like the context switch between two processes), it can be automatically done by existing kernel procedure and requires no further code modification.

Option 2: A possible optimization is to create a new set of page table only for ASMS when creating a new Arbiter thread. Thus only part of the TLB needs to be flushed during context switch between two Arbiter threads. While this can potentially reduce the TLB-miss rate, it would require lots of modifications to the kernel, especially on the context switch procedure to determine the type of context switch, reload the page table for ASMS, and flush the TLB partially.

In sum, there is a trade-off between "TLB-miss overhead" and "how much code modification is needed". Both options have pros and cons. We take the first option and our evaluation shows that the performance overhead is already acceptable.

4 Implementation

We implement Arbiter based on Linux. This section highlights a few implementation details.

4.1 ASMS Mechanism

Kernel Memory Region Management. To properly create or destroy ASMS memory regions in the kernel so as to enlarge or shrink ASMS, we implement a set of kernel functions similar to their Linux equivalents such as `do_mmap` and `do_munmap`. The difference is that when creating or destroying ASMS memory regions for a calling thread, the operation will also be propagated to all the other Arbiter threads. How to configure the protection bits is determined by the arguments passed in from our special system calls (by the Security Manager), including `absys_sbrk`, `absys_mmap`, and `absys_mprotect`. They all have similar semantics to their Linux equivalents, but with additional arguments to denote the permissions.

We add a special flag `AB_VMA` to the `vm_flags` field of the memory region descriptor (*i.e.,* `vm_area_struct`),

which differentiates ASMS from other memory segments. The page fault handler also relies on this flag to identify ASMS page faults. To make sure that only the Security Manager can do allocation, deallocation, and protection modification on ASMS memory regions, we modify related system calls, such as `mmap` and `mprotect`, to prevent them from manipulating ASMS.

User-level Memory Allocation Library. Built on top of our special ASMS system calls is our user-level memory allocation library Memory blocks are sequentially allocated from the start of ASMS. Some data objects might have larger size and cannot fit in a *regular block*. In this case, *large blocks* will be allocated backward starting at the end of ASMS. The pattern of this memory layout is shown in the top half of Figure 3. Inside each block, we take advantage of the dlmalloc algorithm [21] to allocate memory chunks for each data object. The bottom half of Figure 3 depicts the memory chunks on pages inside a block. Further details on our allocation/deallocation algorithms can be found in Appendix A.2.

4.2 Page Fault Handling

A page fault on ASMS typically leads to two possible results: ASMS demand paging and segmentation fault. ASMS demand paging happens when a Arbiter thread legally accesses an ASMS page for the first time. In this case, the page fault handler should find the shared physical page frame and create and configure the corresponding page table entry for the Arbiter thread. The protection bits of the page table entry are determined according to the associated memory region descriptor. In this way, subsequent accesses to this page will be automatically checked by MMU and trapped if illegal. This hardware enforced security check significantly contributes to the runtime performance of Arbiter. An illegal access to an ASMS page will result in a `SIGSEGV` signal sent to the faulting thread. We implement a kernel procedure `do_ab_page` as a subprocedure to the default page fault handler to realize the above idea.

4.3 Miscellaneous

Application Startup. In Arbiter, an application is always started by a Security Manager. A Security Manager first executes and initializes the needed data structures, such as the label registry. Then, it registers its identity to the kernel so as to get privileges of performing subsequent operations on ASMS. We implement a system call `ab_register` for this purpose. Next, the Security Manager starts the application using `fork` and `exec`, and then blocks until a request coming from the Arbiter threads. The application process can create child thread by calling `ab_pthread_create`, which is implemented based on the system call `clone`. The label and ownership of the new thread, if not specified, default to its parent's.

RPC. A reliable RPC connection between Arbiter threads and the Security Manager is quite critical in our system. We implement the RPC based on Unix socket. A major advantage of Unix socket for us is about security: it allows a receiver to get the sender's Unix credentials (*e.g.,* PID), from which the Security Manager is able to verify the identity of the sender. This is especially important in situations where the sender thread is compromised and manipulated by the attacker to send illegal requests or forged information on behalf of an innocent thread.

Authentication and Authorization. The Security Manager needs to perform two actions before processing an RPC: authentication and authorization. Authentication helps to make sure the caller is a valid Arbiter thread. This is done by verifying the validity of its PID acquired from the socket. Authorization ensures that the caller has the needed privilege for the requested operation. For example, RULE 2 must be satisfied for a thread creation request, and RULE 3 must be satisfied for a memory allocation request. If either of the two verifications fail, the Security Manager simply returns the RPC with an indication of security violation.

Futex. Due to our implementation of thread creation, a problem arises with the futexes (*i.e.,* fast userspace mutex) located on data segment (including both `.data` and `.bss`). Multithreaded programs often utilize mutexes and condition variables for mutual exclusion and synchronization. In Pthreads, both of them are implemented using futex. Originally, kernel assigns the key (*i.e.,* identifier) of each futex as either the address of `mm_struct` if the futex is on an anonymous page or the address of `inode` if the futex is on a file backed page. In Arbiter, since data segment is anonymous mapping but the `mm_struct`'s of the Arbiter threads are different, kernel will treat the same mutex or condition variable as different ones. Nonetheless, we can force programmers to declare them on ASMS (which resembles file mapping) that does not have this issue. However, we decide to reduce programmers' effort by modifying the corresponding kernel routine `get_futex_key` and set the key to a same value (*i.e.,* the address of `mm_struct` of the Security Manager). As such, the futex identification problem is resolved.

5 Application

We explore Arbiter's applicability through case studies across various multithreaded applications. We find that the inter-thread privilege separation problem are indeed real-world security concerns. This section introduces our case studies on three different applications: (1) Memcached, (2) Cherokee, and (3) FUSE.

5.1 Memcached

Overview. Memcached [9] is an in-memory data object caching system. It caches data objects from the results of database queries, API calls, or page renderings into memory so that the average response time can be largely reduced. There are mainly three types of threads

in a Memcached process: main thread, worker thread, and maintenance thread. Upon arrival of each client request, the main thread first does some preliminary processing (*e.g.,* packet unwrapping) and then dispatches a worker thread to serve that request. Periodically, maintenance threads wake up to maintain some important assets like the hash table.

Security concern. We identify two potential security concerns. (1) It is reported that a number of large public websites had left Memcached open to arbitrary access from the Internet [2]. This is probably due to the fact that the default configuration of Memcached allows it to accept requests from any IP address plus its authentication support SASL (Simple Authentication and Security Layer) is by default disabled (as Memcached is designed for speed, not security). It has been shown possible to connect to such a server, extract a copy of cache, and write data back to the cache [7]. (2) The vulnerabilities in Memcached [11, 10] could be exploited by an adversary (*e.g.,* buffer overflow attack via CVE-2009-2415) so that the compromised worker thread can arbitrarily access data privately owned by other threads.

Retrofitting. We adapt Memcached to realize the accessibility shown in Table 2. In particular, we assume that a Memcached server is used to serve two applications or two users, A and B. Both A and B privately own their cached data objects that are not supposed to be viewed by the other. For the Shared Data, we make `CQ_ITEM` and a few other metadata read-writable to the main thread but read-only to the worker threads. We slightly change the original thread dispatching scheme so that requests from different principals can be delivered to the associated worker threads. This modification does not affect other features of Memcached.

5.2 Cherokee

Overview. Cherokee [3] is a multithreaded web server designed for lightweight and high performance. Essentially there is only one type of thread in Cherokee: worker thread. Every worker thread repeats the same procedure, that is, it first checks and accepts new connections, adds the new connections to the per-thread connection list, and then processes requests coming from these connections. All the requests coming from the entire life cycle of a connection will be handled by the same thread.

Security concern. (1) An attacker could exploit the vulnerabilities of the Cherokee (*e.g.,* format string vulnerability CVE-2004-1097) to inject shellcode and thus access the data of another connection served by a different thread. (2) Logic bugs might exist in the web server so that an attacker can fool the thread to overread a buffer, which may contain the data belonging to another connection/thread. A recent bug of this type is the Heartbleed bug in OpenSSL [8].

Retrofitting. Our goal is to prevent the threads from accessing each other's private data without affecting the normal functionality. Therefore, we make the buffers allocated for individual connections only accessible by the corresponding thread. Global data structures are made accessible to all the threads, for example, the `struct cherokee_server` which stores the server global configuration, listening sockets file descriptors, mutexes, *etc.*

5.3 FUSE

Overview. FUSE (Filesystem in Userspace) [6] is a widely used framework for developing file systems in user space. Common usages include archive file systems– accessing files inside archives like tar and zip, database file systems– storing files in a relational database or allowing searching using SQL queries, encrypted file systems– storing encrypted files on disk, and network file systems– storing files on remote computers.

When a FUSE volume is mounted, all file system operations against the mount point will be redirected to the FUSE kernel module. The kernel module is registered with a set of callback functions in a multithreaded user space program, which implements the corresponding file system operations. Each worker thread can individually accept and handle kernel callback requests.

Security concern. (1) Logic flaws like careless boundary checking might allow one user to overread a buffer that contains private data of another user. The two users could have very different file system permissions and thus should not share the same set of files. This is especially critical for encrypted file systems (*e.g.,* EncFS [5]), since the intermediate file data is in memory as cleartext. A malicious user can enjoy a much easier and more elegant way to steal data, compared with cracking the encrypted file on disk by brute force. (2) Although the chance is low due to the limited attack surface, we envision a type of attack in which an attacker can compromise a particular thread and inject shellcode. Then the attacker will be able to directly read the data of another user in memory.

Retrofitting. In general, we make the buffers allocated inside `process_cmd()` private to each thread. The global data structure `struct fuse` is shared among all threads, which contains information like callback function pointers, lookup table, metadata of the mount point, *etc.* In addition, we change the thread dispatching scheme from round robin to associating users with threads, which is similar to what we do for Memcached.

5.4 Summary of Porting Effort

Porting these applications to Arbiter was a smooth experience. Actually, most of our time is spent on understanding the source code and data sharing semantics. After that, we define accessibility and devise label assignments accordingly. Finally, we modify the source code, replacing related thread creation and memory allocation functions with Arbiter API. Table 3 summarizes the total LOC and the LOC added/changed for each application.

Application	Total LOC (approx.)	LOC added/changed
Memcached-1.4.13	20k	100 (0.5%)
Cherokee-1.2.2	60k	188 (0.3%)
FUSE-2.3.0	8k	129 (1.6%)

Table 3: Summary of porting effort in the amount of source code change

6 Evaluation

6.1 Protection Effectiveness

As stated in our threat model, we assume that the target application is already properly confined by OS abstraction level access control mechanisms, such as SELinux or AppArmor. To this end, our system can be considered complementary to these OS abstraction level mechanisms. Here our goal is not to evaluate whether our system can achieve the OS abstraction level access control (*e.g.,* preventing a compromised thread from accessing a confidential file). Instead, we want to see under the protection of Arbiter whether a compromised thread can still contaminate or steal the data belonging to another thread.

We assume that an adversary has exploited a program flaw or vulnerability in the three applications ported by us and thus taken control of a worker thread. We simulate various malicious attempts based on the security concerns we presented earlier in §5.

Memcached. We simulate two types of attacks mentioned in §5.1. (1) We simulate an attacker connecting to Memcached via `telnet`. For the vanilla Memcached, the attacker can successfully extract or overwrite any data using the corresponding keys. On the ported Memcached (protected by Arbiter), our attempts to retrieve data belonging to a different user always fail. (2) We then simulate the scenario presented in §5.1 to simulate a buffer overflow attack. We assume that B is an attacker. To simplify simulation, we hard-code our "shellcode" in the source code. Our "shellcode" try to overwrite `CQ_ITEM` and read A's data by traversing the slablist (`(&slabclass[i])->slab_list[j]`). We find that writing to `CQ_ITEM` always fail and traversing the slablist will fail whenever encountering a slab storing A's data.

Note that in both (1) and (2), a failed attempt always triggers a segmentation fault and thus program crash. In practice, the signal handler can be used with Arbiter to deal with such security violations in a more robust way (*e.g.,* sending no response back or dropping the connection). In our experiments, we simply omit this part.

Cherokee. (1) We first simulate the format string attack. We add our "shellcode" to the source code to get another thread's data via the `header` and `buffer` field of the connection structure (`struct cherokee_connection`), which is referenced by the victim thread's active connection list (`&thd->active_list`). We observe that both read and write attempts fail without exception. (2) Then we simulate the logic bug. Particularly, we craft a buffer overread bug by substituting the `buf_size` parameter in the `cherokee_socket_write()` function with a number from our input. When we use a small value for `buf_size`, the buffer overread does not fail in most cases because the adjacent memory is also allocated with the same label. This is tolerable since the attacker only gets the data of his own. When we input a value that is larger than the size of a regular block (*i.e.,* 40KB in our case), the attack always fail. Again, in both (1) and (2), a failure always leads to a segmentation fault in the web server.

FUSE. The simulation of FUSE is very similar to what we do for Cherokee. Arbiter can successfully defeat both (1) logic flaw exploits and (2) code injection attacks.

Counterattacks. We enumerate a few typical counterattacks that are intended to bypass the Arbiter protection.

1) The adversary may want to call `mprotect` to change the permission of ASMS and then access the data.

2) The adversary may attempt to call `ab_munmap` first and then `ab_mmap` to indirectly modify the permission.

3) The adversary may call `fork` or `pthread_create` to create a normal process or thread that is out of the Security Manager's control so as to access the data.

4) The adversary may also want to `fork` a child process and let the child process call `ab_register` to set itself as a new Security Manager. In this way, the adversary hopes to gain full control of the ASMS.

5) The adversary forges a reference and fools an innocent thread to access data on behalf of the adversary.

We try each of the above counterattacks for multiple times, but no one succeeds. The reasons are as below. For 1), it is because Arbiter forbids normal system calls including `mprotect` to operate ASMS. For 2), since the adversary does not have permission to access the data, the Security Manager simply denies the `ab_munmap` request. For 3), unfortunately ASMS will not be mapped to the normal processes or threads. For 4), there do exist ASMS now and the child process does gain full control. However, the ASMS no longer has the same physical mapping. For 5), it would actually have a chance to succeed. However, Arbiter provides an API `get_privilege` which allows the innocent thread to verify if the requesting thread has the necessary permission. As such, Arbiter can still defeat this counterattack. In sum, we believe that within our threat model no counterattack can succeed.

6.2 Microbenchmarks

We build a set of microbenchmarks to examine the performance overhead of Arbiter API. Our experiments were run on a Dell T310 server with Intel Xeon quad-core X3440 2.53GHz CPU and 4GB memory. We use 32-bit Ubuntu Linux (10.04.3) with kernel 2.6.32 and glibc 2.11.1. Since we implement the ASMS Library based on uClibc 0.9.32, we use the same version for comparison on memory allocation. Each result is averaged over 1,000 times of repeat.

Operation	Linux (μs)	Arbiter (μs)	Overhead
(ab_)malloc	4.14	9.09	2.20
(ab_)free	2.06	8.36	4.06
(ab_)calloc	4.14	8.41	2.03
(ab_)realloc	3.39	8.27	2.43
(ab_)pthread_create	91.45	145.33	1.59
(ab_)pthread_join	36.22	41.00	1.13
(ab_)pthread_self	2.99	1.98	0.66
create_category	–	7.17	–
get_label	–	7.65	–
get_ownership	–	7.55	–
get_mem_label	–	7.66	–
ab_null (RPC round trip)	–	5.84	–
(absys_)sbrk	0.65	0.76	1.36
(absys_)mmap	0.60	0.83	1.38
(absys_)mprotect	0.83	0.92	1.11

Table 4: Microbenchmark results in Linux and Arbiter

Table 4 shows the comparison of microbenchmarks. The overhead of memory allocation functions (*e.g.,* `ab_free`) is non-trivial. This is because they have to go through the Security Manager via an RPC round trip, which consists of RPC marshalling, socket latency, *etc.* We find that a pure RPC round trip (`ab_null`) itself already takes 5.84μs, which helps to justify the time consumption of most Arbiter API functions. Due to our implementation of thread creation, we directly use `getpid` to return the thread ID. As the result, `ab_pthread_self` runs even faster than its Linux equivalent. In addition to the RPC latency, the system calls made by the Security Manager also contribute to the API overhead. We examine `sbrk`, `mmap`, and `mprotect` and find that Arbiter incurs 28% overhead on average.

There are two other factors that might affect the overhead of Arbiter API: (1) The number of threads can affect the memory allocation overhead. Figure 4(a) shows that the time consumption of `ab_malloc` is roughly correlated with the number of threads. The time consumption increases by around 5.7% per additional thread. This is because memory allocation on ASMS for one thread is also propagated to other threads. For comparison, we also show the result of `get_label`. This operation does not involve any "propagation" and thus is not affected by the number of threads. (2) The size of allocated ASMS can affect the thread creation overhead. This is because thread creation involves the permission reconfiguration of ASMS. Figure 4(b) shows that the time consumption of `ab_pthread_create` increases along with the size of allocated ASMS (note the logarithmic scale on x-axis). This is also in line with our expectation.

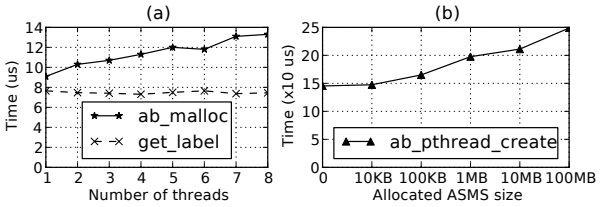


Figure 4: Arbiter API performance regarding number of threads and allocated ASMS size

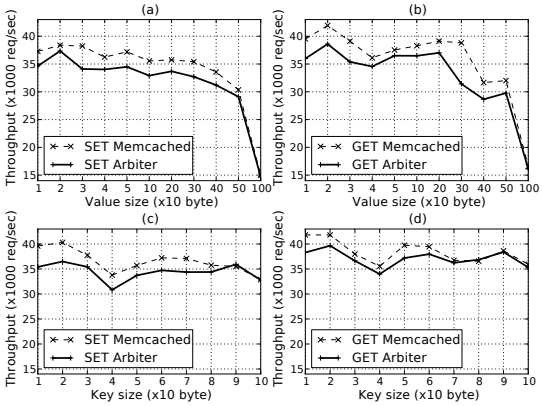


Figure 5: Performance comparison for Memcached

6.3 Application Performance

Memcached. We build a security-enhanced Memcached based on its version 1.4.13 and we use libMemcached 1.0.5 as the client library. We measure the throughput of two basic operations, SET and GET, with various value sizes and key sizes. The results are compared with unmodified Memcached. In Figure 5(a) and 5(b), we anchor the key size to 32 bytes and change the value size. In Figure 5(c) and 5(d), we fix the value size to 256 bytes and adjust the key size. Each point in the figure is an average of 100,000 times of repeat. All together, the average performance decrease incurred by Arbiter is about 5.6%.

Cherokee. We port Cherokee based on its version 1.2.2. We use the `ApacheBench` version 2.3 and static HTML files to measure its performance. First, we measure the influence of file size. We choose files with sizes of 1KB, 10KB, 100KB, and 1MB. Figure 6(a) shows the comparison between vanilla Cherokee and the ported version. The average slowdown is 1.8%. Second, we test the system scalability by tuning the number of threads from 5 to 40. We fix the file size to 1KB during this round of test. The throughput comparison is shown in Figure 6(b). The average performance degradation is around 3.0%. This comparison indicates that running more threads does not necessarily induce more overhead. For each individual test, we set `ApacheBench` to issue 10,000 requests with the concurrency level of 10.

FUSE. We retrofit FUSE based on its version 2.3.0. For the custom userspace file system, we use the exam-

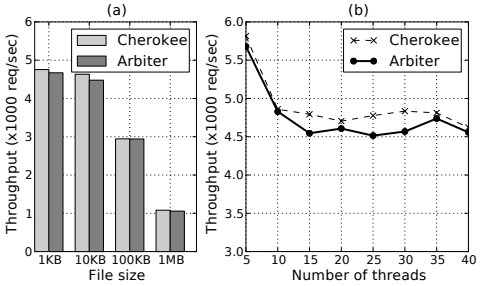


Figure 6: Performance comparison for Cherokee

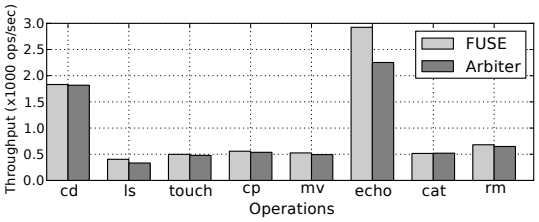


Figure 7: Performance comparison for FUSE

ple implementation `fusexmp` provided by FUSE source package. It simply emulates the native file system. We then select 8 representative commands relevant to file system operations, namely, `cd`, `ls`, `touch`, `cp`, `mv`, `echo`, `cat`, and `rm`. Note that the `echo` command is used to write a 32-byte string to files. Each command is repeated for 10,000 times. Figure 7 shows the comparison between unmodified FUSE and the ported version. On average, the slowdown is 7.4%.

Overall, the application performance overhead is acceptable. This is partially contributed by the fact that the extra cost of Arbiter API calls is amortized by other operations of these programs.

6.4 CPU and Memory Overhead

In addition to the throughput comparison, we further evaluate the CPU cost. As shown in Table 5, Arbiter increases the CPU utilization by 1.29–1.55×. We leverage the CPU time information in /proc/[pid]/stat to do the calculation. We also count the types of labeled objects (not to be confused with runtime instances), shown in the last column of Table 5. Interestingly, the number of labeled objects is roughly correlated with the CPU overhead.

Although our "same accessibility, same page" strategy has already come with much less memory waste than "one object per page", it still incurs some memory overhead. Table 6 shows the average resident memory (RSS) usage of the three applications during the performance test. We measure RSS by checking the `VmRSS` value of /proc/[pid]/status around ten times per second. Given that the policy we used for the three applications are quite typical, we believe real-world memory overhead should be close to the measured overhead.

7 Discussion and Limitations

We believe that Arbiter provides a generic and practical mechanism for inter-thread privilege separation on data objects. Nonetheless, it still has limitations in defending against certain security threats. When two principal users or clients are served by the same thread, Arbiter can no longer enforce privilege separation for the two principals. Thus, programmers have to be very careful dealing with user authentication and thread dispatching to associate principals with appropriate worker threads. To fully address this issue, one possible solution is to have a per-principal-user "virtual" thread to further separate the privileges. We leave this as a future work.

Application	Original	Arbiter	Overhead	Labeled objects
memcached	49.4%	76.7%	1.55×	14
cherokee	58.8%	76.1%	1.29×	8
FUSE	42.3%	58.0%	1.37×	10

Table 5: Comparison of CPU utilization and labeled objects

Application	Original (KB)	Arbiter (KB)	Overhead
memcached	60,664	64,452	6.2%
cherokee	3,916	4,120	5.2%
FUSE	732	760	3.9%

Table 6: RSS memory overhead

One limitation of our implementation is that the user-space memory allocator uses a single lock for allocation/deallocation. Therefore, the processing of allocation and deallocation requests have to be serialized. A finer lock granularity can help to improve parallelism and scalability, such as Hoard [13] and TCMalloc [12]. In fact, Arbiter's memory allocation mechanism inherently has the potential to adopt a per-label lock. We are looking at ways to implement such a parallelized allocator.

Arbiter's security relies on the correctness of privilege separation policy configured by the programmer. However, it may not be that easy to get all the label assignments correct, especially in complex and dynamic deployment scenarios. Actually, DIFC systems also confront similar policy configuration challenges and research efforts have been made to debug DIFC policy misconfiguration [35]. Our system is also able to incorporate a policy debugging or model checking tool that can verify the correctness of label assignments.

Arbiter's security model, including notions and rules, is inspired by DIFC. However, it should be noted that Arbiter does not perform information flow tracking inside a program, mainly due to two observations: (1) For a runtime system approach, tracking fine-grained data flow (*e.g.,* moving a 4-byte integer from memory to a CPU register) could incur tremendous overhead, making Arbiter impractical to use; (2) The fact that information flow tracking can enhance security does not logically exclude the possibility of solving real security problems without information flow tracking. The main contribution of Arbiter is that it provides fine-grained privilege separation for data objects using commodity hardware, while still preserving the traditional multithreaded programming paradigm.

8 Conclusion

Arbiter is a system targeting at fine-grained, data object-level privilege separation for multi-principal multithreaded applications. Particularly, we find that page table protection bits can be leveraged to do efficient reference monitoring if data objects with same accessibility are put into the same page. We find that Arbiter is applicable to a verity of real-world applications. Our experiments demonstrate Arbiter's ease of adoption, effectiveness of protection, as well as reasonable performance overhead.

Acknowledgement

We would like to thank our paper shepherd Xi Wang and the anonymous reviewers, for their insightful feedback that helped shape the final version of this paper.

This work was supported by NSF CNS-1223710, NSF CNS-1422594, and ARO W911NF-13-1-0421 (MURI).

References

[1] Apparmor. http://www.novell.com/linux/security/apparmor/.

[2] Blackhat write-up: go-derper and mining memcaches. http://www.sensepost.com/blog/4873.html.

[3] Cherokee. http://cherokee-project.com.

[4] The chromium projects: Multi-process architecture. http://www.chromium.org/developers/design-documents/multi-process-architecture.

[5] Encfs. http://www.arg0.net/encfs.

[6] Fuse: Filesystem in userspace. http://fuse.sourceforge.net.

[7] go-derper. http://research.sensepost.com/tools/servers/go-derper.

[8] The heartbleed bug. http://heartbleed.com.

[9] Memcached. http://www.memcached.org.

[10] Sa-contrib-2010-098 - memcache - multiple vulnerabilities. http://drupal.org/node/927016.

[11] Security vulnerabilities in memcached. http://www.cvedetails.com/vulnerability-list/vendor_id-9678/producti_id-17294/Memcachedb-Memcached.html.

[12] Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[13] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS* (2000), pp. 117–128.

[14] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 309–322.

[15] BRUMLEY, D., AND SONG, D. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 5–5.

[16] CHENG, W., PORTS, D. R. K., SCHULTZ, D., POPIC, V., BLANKSTEIN, A., COWLING, J., CURTIS, D., SHRIRA, L., AND LISKOV, B. Abstractions for Usable Information Flow Control in Aeolus. In *USENIX ATC '12*.

[17] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *OSDI* (2006).

[18] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium* (2010), pp. 143–160.

[19] KILPATRICK, D. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track* (2003), USENIX, pp. 273–284.

[20] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. SOSP '07.

[21] LEA, D. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[22] LI, X., YAN, W., AND XUE, Y. Sentinel: securing database from logic flaws in web applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012), ACM, pp. 25–36.

[23] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the linux operating system. In *USENIX ATC* (2001).

[24] MAO, Y., CHEN, H., ZHOU, D., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Software fault isolation with api integrity and multi-principal modules. In *In SOSP* (2011).

[25] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-E: A Security-Oriented Subset of Java. In *NDSS '10*.

[26] MYERS, A. C. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), ACM, pp. 228–241.

[27] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing privilege escalation. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), USENIX Association, pp. 16–16.

[28] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: practical fine-grained decentralized information flow control. PLDI '09.

[29] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[30] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. SOSP '93.

[31] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: practical capabilities for unix. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 3–3.

[32] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 304–316.

[33] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. *Commun. ACM 54*, 11 (Nov. 2011), 93–101.

[34] ZELDOVICH, N., KANNAN, H., DALTON, M., AND KOZYRAKIS, C. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 225–240.

[35] ZHAO, M., AND LIU, P. Modeling and checking the security of DIFC system configurations. In *Automated Security Management*. Springer, 2013, pp. 21–38.

A Appendix

A.1 Arbiter API

Figure 8 lists the Arbiter's API, which are used for labeling, threading, and memory allocation. To preserve the multithreaded programming paradigm, the function syntax is fully compatible with the C Standard Library and the Pthreads Library. For example, if a programmer uses `ab_malloc` without assigning any label (`L = NULL;`), it will behave in the same way as libc `malloc`, *i.e.,* allocating a memory chunk read-writable to every thread. This makes it possible for programmers to incrementally adapt their programs to our system.

A.2 ASMS Memory Allocation Algorithm

§3.3 and §4.1 described our permission-oriented allocation mechanism. Here we explain the detailed algorithm shown in Figure 9. For clarity, we omit the discussion on the strategy of memory chunk management adopted from dlmalloc.

- **Allocation** If the size of the data is larger than a regular block size (*i.e.,* threshold), a large block will be allocated using `absys_mmap` (line 5). Otherwise, the allocator will search for free chunks inside blocks with that label (line 7). If there is an available free chunk, the allocator simply returns it. If not, the allocator will allocate a new regular block using `absys_sbrk` (line 12).
- **Deallocation** For a large block, the allocator simply frees it using `absys_munmap` (line 3) so that it can be reused later on. Otherwise, the allocator puts the chunk back to the free list (line 5). Next, the allocator checks if all the chunks on this block are free. If so, this block will be recycled for later use (line 7).

- `cat_t create_category(cat_type t);`
  Create a new category of type `t`, which can be either secrecy category `CAT_S` or integrity category `CAT_I`.
- `void get_label(label_t L);`
  Get the label of a thread itself into *L*.
- `void get_ownership(own_t O);`
  Get the ownership of a thread itself into *O*.
- `void get_mem_label(void *ptr, label_t L);`
  Get the label of a data object into *L*.
- `int ab_pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg, label_t L, own_t O);`
  Create a new thread with label *L* and ownership *O*.
- `int ab_pthread_join(pthread_t thread, void **value_ptr);`
  Wait for thread termination.
- `pthread_t ab_pthread_self(void);`
  Get the calling thread ID.
- `void *ab_malloc(size_t size, label_t L);`
  Allocate dynamic memory on ASMS with label *L*.
- `void ab_free(void *ptr);`
  Free dynamic memory on ASMS.
- `void *ab_calloc(size_t nmemb, size_t size, label_t L);`
  Allocate memory for an array of elements on ASMS with label *L*.
- `void *ab_realloc(void *ptr, size_t size);`
  Change the size of the memory on ASMS.
- `void *ab_mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset, label_t L);`
  Map files to ASMS with label *L*.
- `int get_privilege(pthread_t thread, void *ptr);`
  Query the permission of a thread to accessing memory on ASMS.

Figure 8: List of Arbiter API

```
1  ablib_malloc(sz, L)
2      if sz > threshold then
3          for every member thread do
4              Compute permission
5          Allocate a large block
6          return
7      Search free chunks in blocks with label L
8      if there is an available free chunk then
9          return
10     for every member thread do
11         Compute permission
12         Allocate a regular block
13     return

1  ablib_free ( ptr )
2      if it is a large block then
3          Free the block
4          return
5      Free the chunk pointed by ptr
6      if the whole block is free now then
7          Free the block
8      return
```

Figure 9: ASMS memory allocation algorithm

GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning

Xiaowei Zhu, Wentao Han, and Wenguang Chen

Department of Computer Science and Technology, Tsinghua University

{zhuxw13,hwt04}@mails.tsinghua.edu.cn, cwg@tsinghua.edu.cn

Abstract

In this paper, we present GridGraph, a system for processing large-scale graphs on a single machine. GridGraph breaks graphs into 1D-partitioned vertex chunks and 2D-partitioned edge blocks using a first fine-grained level partitioning in preprocessing. A second coarse-grained level partitioning is applied in runtime. Through a novel dual sliding windows method, GridGraph can stream the edges and apply on-the-fly vertex updates, thus reduce the I/O amount required for computation. The partitioning of edges also enable selective scheduling so that some of the blocks can be skipped to reduce unnecessary I/O. This is very effective when the active vertex set shrinks with convergence.

Our evaluation results show that GridGraph scales seamlessly with memory capacity and disk bandwidth, and outperforms state-of-the-art out-of-core systems, including GraphChi and X-Stream. Furthermore, we show that the performance of GridGraph is even competitive with distributed systems, and it also provides significant cost efficiency in cloud environment.

1 Introduction

There has been increasing interests to process large-scale graphs efficiently in both academic and industrial communities. Many real-world problems, such as online social networks, web graphs, user-item matrices, and more, can be represented as graph computing problems.

Many distributed graph processing systems like Pregel [17], GraphLab [16], PowerGraph [8], GraphX [28], and others [1, 22] have been proposed in the past few years. They are able to handle graphs of very large scale by exploiting the powerful computation resources of clusters. However, load imbalance [11, 20], synchronization overhead [33] and fault tolerance overhead [27] are still challenges for graph processing in distributed environment. Moreover, users need to be skillful since tuning a cluster and optimizing graph algorithms in distributed systems are non-trivial tasks.

GraphChi [13], X-Stream [21] and other out-of-core systems [9, 15, 31, 34] provide alternative solutions. They enable users to process large-scale graphs on a single machine by using disks efficiently. GraphChi partitions the vertices into disjoint intervals and breaks the large edge list into smaller shards containing edges with destinations in corresponding intervals. It uses a vertex-centric processing model, which gathers data from neighbors by reading edge values, computes and applies new values to vertices, and scatters new data to neighbors by writing values on edges. By using a novel parallel sliding windows method to reduce random I/O accesses, GraphChi is able to process large-scale graphs in reasonable time. However, its sharding process requires the edges in every shard to be sorted by sources. This is a very time consuming process since several passes over edges are needed. Fragmented accesses over several shards are often inevitable, decreasing the usage of disk bandwidth. X-Stream introduces an edge-centric scatter-gather processing model. In the scatter phase, X-Stream streams every edge and generates updates to propagate vertex states. In the gather phase, X-Stream streams every update, and applies it to the corresponding vertex state. Accesses to vertices are random and happen on a high level of storage hierarchy which is small but fast. And accesses to edges and updates fall into a low level of storage hierarchy which is large but slow. However, these accesses are sequential so that maximum throughput can be achieved. Although X-Stream can leverage high disk bandwidth by sequential accessing, it needs to generate updates which could be in the same magnitude as edges, and its lack of support on selective scheduling could also be a critical problem when dealing with graphs of large diameters.

We propose GridGraph, which groups edges into a "grid" representation. In GridGraph, vertices are partitioned into 1D chunks and edges are partitioned into 2D

blocks according to the source and destination vertices. We apply a logical higher level of partitioning in runtime. Chunks and blocks are grouped into larger ones for I/O efficiency. Different from current vertex-centric or edge-centric processing model, GridGraph combines the scatter and gather phases into one "streaming-apply" phase, which streams every edge and applies the generated update instantly onto the source or destination vertex. By aggregating the updates, only one pass over the edge blocks is needed. This is nearly optimal for iterative global computation, and is suited to both in-memory and out-of-core situations. Moreover, GridGraph offers selective scheduling, so that streaming on unnecessary edges can be reduced. This significantly improves performance for many iterative algorithms.

We evaluate GridGraph on real-world graphs and show that GridGraph outperforms state-of-the-art out-of-core graph engines by up to an order of magnitude. We also show that GridGraph has competitive performance to distributed solutions, and is far more cost effective and convenient to use. In summary, the contributions of this paper are:

- A novel grid representation for graphs, which can be generated from the original edge list using a fast range-based partitioning method. Different from the index plus adjacency list or shard representation that requires sorting, edge blocks of the grid can be transformed from an unsorted edge list with small preprocessing overhead, and can be utilized for different algorithms and on different machines.
- A 2-level hierarchical partitioning schema, which is effective for not only out-of-core but also in-memory scenarios.
- A fast streaming-apply graph processing model, which optimizes I/O amount. The locality of vertex accesses is guaranteed by dual sliding windows. Moreover, only one sequential read of edges is needed and the write amount is optimized to the order of vertices instead of edges.
- A flexible programming interface consisting of two streaming functions for vertices and edges respectively. Programmers can specify an optional user-defined filter function to skip computation on inactive vertices or edges. This improves performance significantly for iterative algorithms that active vertex set shrinks with convergence.

The remaining part of this paper is organized as follows. Section 2 describes the grid representation, which is at the core of GridGraph. Section 3 presents the computation model, and the 2-level hierarchical partitioning schema. Section 4 evaluates the system, and compares it with state-of-the-art systems. Section 5 discusses related works, and finally Section 6 concludes the paper.

2 Graph Representation

The grid representation plays an important role in GridGraph. We introduce the details of the grid format, as well as how the fast partitioning process works. We also make a comparison with other out-of-core graph engines and discuss the trade-offs in preprocessing.

2.1 The Grid Format

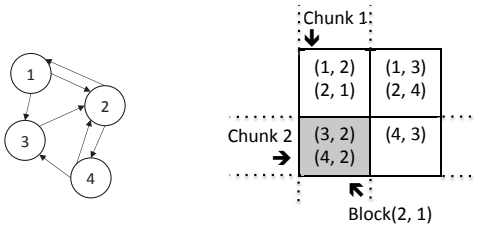
Partitioning is employed to process a graph larger than the memory capacity of a single machine. GraphChi designs the shard data format, and stores the adjacency list in several shards so that each shard can be fit into memory. Vertices are divided into disjoint intervals. Each interval associates a shard, which stores all the edges with destination vertex in the interval. Inside each shard, edges are sorted by source vertex and combined into the compact adjacency format. X-Stream also divides the vertices into disjoint subsets. A streaming partition consists of a vertex set, an edge list and an update list, so that data of each vertex set can be fit into memory. The edge list of a streaming partition (in the scatter phase) consists of all edges whose source vertex is in the partition's vertex set. The update list of a streaming partition (in the gather phase) consists of all updates whose destination vertex is in the partition's vertex set.

GridGraph partitions the vertices into P equalized vertex chunks. Each chunk contains vertices within a contiguous range. The whole $P \times P$ blocks can be viewed as a grid, and each edge is put into a corresponding block using the following rule: the source vertex determines the row of the block and the destination vertex determines the column of the block. Figure 1(b) illustrates how GridGraph partitions the example graph in Figure 1(a). There are 4 vertices in this graph and we choose $P = 2$ for this example. $\{1, 2\}$ and $\{3, 4\}$ are the 2 vertex chunks. For example, Edge (3, 2) is partitioned to Block (2, 1) since Vertex 3 belongs to Chunk 2 and Vertex 1 belongs to Chunk 1.

In addition to the edge blocks, GridGraph creates a metadata file which contains global information of the represented graph, including the number of vertices V and edges E, the partition count P, and the edge type T (to indicate whether the edges are weighted or not). Each edge block corresponds to a file on disks.

GridGraph does preprocessing in the following way:

1. The main thread sequentially reads edges from original unordered edge list and divides them into batches of edges and pushes each batch to the task



(a) An example graph (b) Grid representation

Figure 1: Organization of the edge blocks

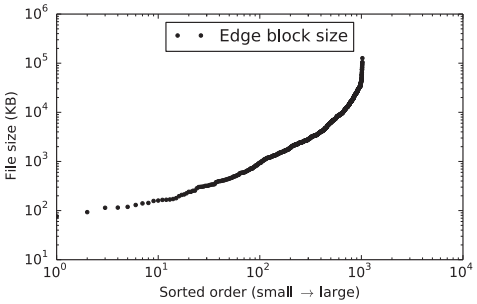


Figure 2: Edge block size distribution of Twitter graph using a 32×32 partitioning.

queue (to achieve substantial sequential disk bandwidth, we choose 24MB to be the size of each edge batch).

2. Each worker thread fetches a task from the queue, calculates the block that each edge in this batch belongs to, and appends the edge to the corresponding edge block file. To improve I/O throughput, each worker thread maintains a local buffer of each block, and flushes to files once the buffer is full.

After the partitioning process, GridGraph is ready to do computation. However, due to the irregular structure of real world graphs, some edge blocks might be too small to achieve substantial sequential bandwidth on HDDs. Figure 2 shows the distribution of edge block sizes in Twitter [12] graph using a 32×32 partitioning, which conforms to the power-law [7], with a large number of small files and a few big ones. Thus full sequential bandwidth can not be achieved sometimes due to potentially frequent disk seeks. To avoid such performance loss, an extra merge phase is required for GridGraph to perform better on HDD based systems, in which the edge block files are appended into a large file one by one and the start offset of each block is recorded in metadata. The time taken by each phase is shown in Section 4.

2.2 Discussion

Different from the shard representation used in GraphChi, GridGraph does not require the edges in each block to be sorted. This hence reduces both I/O and computation overhead in preprocessing. We only need to read and write the edges from and to disks once, rather than several passes over the edges in GraphChi. This lightweight preprocessing procedure can be finished very quickly (see Table 2), which is much faster than the preprocessing of GraphChi.

X-Stream, on the other hand, does not require explicit preprocessing. Edges are shuffled to several files according to the streaming partition. No sorting is required and the number of partitions is quite small. For many graphs that all the vertex data can be fit into memory, only one streaming partitions is needed. However, this partitioning strategy makes it inefficient for selective scheduling, which can largely affect its performance on many iterative algorithms that only a portion of the vertices are used in some iterations.

It takes very short time for GridGraph to complete the preprocessing. Moreover, the generated grid format can be utilized in all algorithms running on the same graph. By partitioning, GridGraph is able to conduct selective scheduling and reduce uncessary accesses to edge blocks without active edges[1]. We can see that this contributes a lot in many iterative algorithms like BFS and WCC (see Section 4), which a large portion of vertices are inactive in many iterations.

The selection of the number of partitions P is very important. With a more fine-grained partitioning (which means a larger value of P), while the preprocessing time becomes longer, better access locality of vertex data and more potential in selective scheduling can be achieved. Thus a larger P is preferred in partitioning. Currently, we choose P in such a way that the vertex data can be fit into last level cache. We choose P to be the minimum integer such that

$$\frac{V}{P} \times U \leq C,$$

where C is the size of last level cache and U is the data size of each vertex. This partitioning shows not only good performance (especially for in-memory situations) but also reasonable preprocessing cost. In Section 4, we evaluate the impact of P and discuss the trade-offs inside.

3 The Streaming-Apply Processing Model

GridGraph uses a streaming-apply processing model in which only one (read-only) pass over the edges is required and the write I/O amount is optimized to one pass over the vertices.

[1] An edge is active if its source vertex is active.

3.1 Programming Abstraction

GridGraph provides 2 functions to stream over vertices (Algorithm 1) and edges (Algorithm 2).

Algorithm 1 Vertex Streaming Interface

```
function StreamVertices(F_v, F)
    Sum = 0
    for each vertex do
        if F(vertex) then
            Sum += F_v(edge)
        end if
    end for
    return Sum
end function
```

Algorithm 2 Edge Streaming Interface

```
function StreamEdges(F_e, F)
    Sum = 0
    for each active block do          ▷ block with active edges
        for each edge ∈ block do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
        end for
    end for
    return Sum
end function
```

F is an optional user defined function which accepts a vertex as input and should returns a boolean value to indicate whether the vertex is needed in streaming. It is used when the algorithm needs selective scheduling to skip some useless streaming and is often used together with a bitmap, which can express the active vertex set compactly. F_e and F_v are user defined functions which describe the behavior of streaming. They accept an edge (for F_e), or a vertex (for F_v) as input, and should return a value of type R. The return values are accumulated and as the final reduced result to user. This value is often used to get the number of activated vertices, but is not restricted to this usage, e.g. users can use this function to get the sum of differences between iterations in PageRank to decide whether to stop computation.

GridGraph stores vertex data on disks. Each vertex data file corresponds to a vertex data vector. We use the memory mapping mechanism to reference vertex data backed in files. It provides convenient and transparent access to vectors, and simplifies the programming model: developers can treat it as normal arrays just as if they are in memory.

We use PageRank [19] as an example to show how to implement algorithms using GridGraph (shown in Algorithm 3[2]). PageRank is a link analysis algorithm that calculates a numerical weighting to each vertex in the graph to measure its relative importance among the vertices. The PR value of each vertex is initialized to 1. In each iteration, each vertex sends out their contributions to neighbors, which is the current PR value divided by its out degree. Each vertex sums up the contributions collected from neighbors and sets it as the new PR value. It converges when the mean difference reaches some threshold[3].

Algorithm 3 PageRank

```
function Contribute(e)
    Accum(&NewPR[e.dest], PR[e.source] / Deg[e.source])
end function
function Compute(v)
    NewPR[v] = 1 − d + d × NewPR[v]
    return |NewPR[v] − PR[v]|
end function
d = 0.85
PR = {1, ..., 1}
Converged = 0
while ¬Converged do
    NewPR = {0, ..., 0}
    StreamEdges(Contribute)
    Diff = StreamVertices(Compute)
    Swap(PR, NewPR)
    Converged = Diff / V ≤ Threshold
end while
```

3.2 Dual Sliding Windows

GridGraph streams edges block by block. When streaming a specific block, say, the block in the i-th row and j-th column, vertex data associated with this block falls into the i-th and j-th chunks. By selecting P such that each chunk is small enough to fit into the fast storage (i.e. memory for out-of-core situations or last level cache for in-memory situations), we can ensure good locality when accessing vertex data associated with the block being streamed.

The access sequence of blocks can be row-oriented or column-oriented, based on the update pattern. Assume that a vertex state is propagated from the source vertex to the destination vertex (which is the typical pattern in a lot of applications), i.e. source vertex data is read and destination vertex data is written. Since the column of each edge block corresponds to the destination vertex chunk, column oriented access order is preferred in this case. The destination vertex chunk is cached in memory when GridGraph streams blocks in the same column from top to bottom, so that expensive disk write operations are aggregated and minimized. This property is very impor-

[2] Accum(&s, a) is an atomic operation which adds a to s.

[3] Sometimes we run PageRank for certain number of iterations to analyze performance.

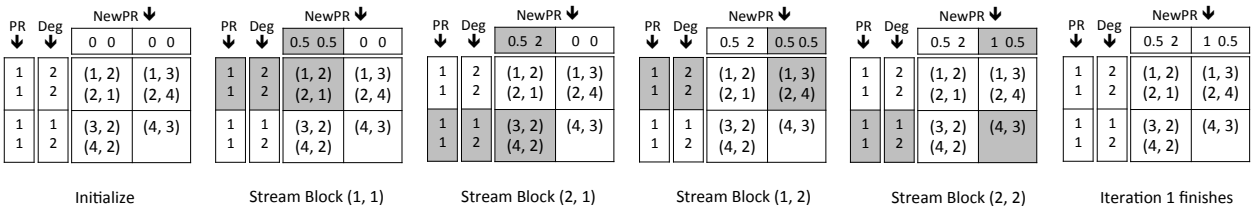


Figure 3: Illustration of dual sliding windows. It shows the first iteration of PageRank on the example graph. Shaded block and chunks are active (loaded into memory).

tant in practical use, especially for SSDs, since the write performance might degrade after writing large volume of data due to the write amplification phenomenon. On the other hand, since SSDs have upper limits of write cycles, it is thus important to reduce disk writes as much as possible to achieve ideal endurance.

Figure 3 visualizes how GridGraph uses dual sliding windows to apply updates onto vertices. PageRank is used as the example. The read window (in which we read current PageRank value and out degree from the source vertex) and the write window (in which we accumulate contributions to the new PageRank value of the destination vertex) slide as GridGraph streams edge blocks in the column-oriented order.

Since GridGraph applies in-place updates onto the active vertex chunk, there might exists data race, i.e. data of one vertex is concurrently modified by multiple worker threads. Thus, inside the process function F_e, users need to use atomic operations to apply thread-safe updates to vertices, so as to ensure the correctness of algorithms.

Utilizing the fact that the bandwidth of parallel randomized access to fast level storage is still orders of magnitude bigger than the sequential bandwidth of slow level storage (such as main memory vs. disks, and cache vs. main memory), the time of applying updates is overlapped with edge streaming. GridGraph only requires one read pass over the edges, which is advantageous to the solutions of GraphChi and X-Stream that need to mutate the edges (GraphChi) or first generating and then streaming through the updates (X-Stream).

Through read-only access to the edges, the memory required by GridGraph is very compact. In fact, it only needs a small buffer to hold the edge data being streamed so that other free memory can be used by page cache to hold edge data, which is very useful when active edge data becomes small enough to fit into memory.

Another advantage of this streaming-apply model is that it not only supports classical BSP [25] model, but also allows asynchronous [3] updates. Since vertex updates are in-place and instant, the effect of an update can be seen by following vertex accesses, which makes lots of iterative algorithms to converge faster.

3.3 2-Level Hierarchical Partitioning

We first give the I/O analysis of GridGraph in a complete streaming-apply iteration, which all the edges and vertices are accessed. Assume edge blocks are accessed in the column-oriented order. Edges are accessed once and source vertex data is read P times while destination vertex data is read and written once. Thus I/O amount is

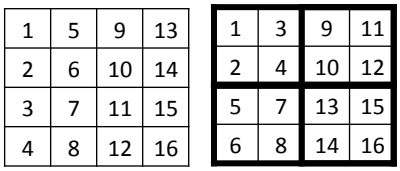
$$E + P \times V + 2 \times V$$

for each iteration. Thus a smaller P should be preferred to minimize I/O amount which seems opposite to the grid partitioning principle discussed in Section 2 that a larger P can ensure better locality and selective scheduling effect.

To deal with this dilemma, we apply a second level partitioning above the edge grid to reduce I/O accesses of vertices. The higher level partitioning consists of a $Q \times Q$ edge grid. Given a specified amount of memory M, and the size of each vertex data U (including source and destination vertex), Q is selected to be the minimum integer which satisfies the condition

$$\frac{V}{Q} \times U \leq M.$$

As we mentioned in Section 2, P is selected to fit vertex data into last level cache of which the capacity is much smaller than memory. Hence P is much bigger than Q, i.e. the $Q \times Q$ partitioning is more coarse-grained than the $P \times P$ one, so that we can just virtually group the edge blocks by adjusting the accessing orders of blocks. Figure 4 illustrates this concept. The preprocessed grid consists of 4×4 blocks, and a virtual 2×2 grid partitioning is applied over it. The whole grid is thus divided into 4 big blocks, with each big block containing 4 small blocks. The number inside each block indicates the access sequence. An exact column-oriented access order is used in the original 4×4 partitioning. After the second level 2×2 over 4×4 partitioning is applied, we access the coarse-grained (big) blocks in column-oriented order, and within each big block, we access the fine-grained (small) blocks in column-oriented order as well.



(a) 4x4 grid (b) 2x2 virtual grid

Figure 4: A 2-Level Hierarchical Partitioning Example. The number inside each block indicates the access sequence.

This 2-level hierarchical partitioning provides not only flexibility but also efficiency since the higher level partitioning is virtual and GridGraph is able to utilize the outcome of lower level partitioning thus no more actual overhead is added. At the same time, good properties of the original fine-grained edge grid such as more selective scheduling chances can still be leveraged.

3.4 Execution Implementation

Thanks to the good locality ensured by the property of dual sliding windows, the execution engine of GridGraph mainly concentrates on streaming edge blocks.

GridGraph streams each block sequentially. Before streaming, GridGraph first checks the activeness of each vertex chunk. Edge blocks are streamed one by one in the sequence that dual sliding windows needs, and if the corresponding source vertex chunk of the block is active, it is added to the task list.

GridGraph does computation as follows:

1. The main thread pushes tasks to the queue, containing the file, offset, and length to issue each read request. (Length is set to 24MB like in preprocessing to achieve full disk bandwidth.)

2. Worker thread fetches tasks from the queue until empty, read data from specified location and process each edge.

Each edge is first checked by a user defined filter function F, and if the source vertex is active, F_e is called on this edge to apply updates onto the source or destination vertex (note that we do not encourage users to apply updates onto both the source and destination vertex, which might make the memory mapped vector suffer from unexpected write backs onto the slow level storage).

4 Evaluation

We evaluate GridGraph on several real world social graphs and web graphs, and shows significant performance improvement compared with current out-of-core graph engines. GridGraph is even competitive with distributed systems when more powerful hardware can be utilized.

Dataset	V	E	Data size	P
LiveJournal	4.85M	69.0M	527 MB	4
Twitter	61.6M	1.47B	11 GB	32
UK	106M	3.74B	28 GB	64
Yahoo	1.41B	6.64B	50 GB	512

Table 2: Graph datasets used in evaluation.

4.1 Test Environment

Experiments are conducted on AWS EC2 storage optimized instances, including d2.xlarge instance, which contains 4 hyperthread vCPU cores, 30.5GB memory (30MB L3 Cache), and 3 HDDs of 2TB, and i2.xlarge instance, which has the same configuration with d2.xlarge except that it contains 1 800GB SSD instead of 3 2TB HDDs (and the L3 cache is 24MB). I2 instances are able to provide high IOPS while D2 instances can provide high-density storage. Both i2.xlarge and d2.xlarge instances can achieve more than 450GB/s sequential disk bandwidth.

For the I/O scalability evaluation, we also use more powerful i2.2xlarge, i2.4xlarge, and i2.8xlarge instances, which contain multiple (2, 4, 8) 800GB SSDs, as well as more (8, 16, 32) cores and (61GB, 122GB, 244GB) memory.

4.2 System Comparison

We evaluate the processing performance of GridGraph through comparison with the latest version of GraphChi and X-Stream on d2.xlarge[4] and i2.xlarge instances.

For each system, we run BFS, WCC, SpMV and Pagerank on 4 datasets: LiveJournal [2], Twitter [12], UK [4] and Yahoo [29]. All the graphs are real-world graphs with power-law degree distributions. LiveJournal and Twitter are social graphs, showing the following relationship between users within each online social network. UK and Yahoo are web graphs that consist of hyperlink relationships between web pages, with larger diameters than social graphs. Table 2 shows the magnitude, as well as our selection of P for each graph. For BFS and WCC, we run them until convergence, i.e. no more vertices can be found or updated; for SpMV, we run one iteration to calculate the multiplication result; and for PageRank, we run 20 iterations on each graph.

[4] A software RAID-0 array consisting of 3 HDDs is set up for evaluation on d2.xlarge instances.

	i2.xlarge (SSD)				d2.xlarge (HDD)			
	BFS	**WCC**	**SpMV**	**PageR.**	**BFS**	**WCC**	**SpMV**	**PageR.**
LiveJournal								
GraphChi	22.81	17.60	10.12	53.97	21.22	14.93	10.69	45.97
X-Stream	6.54	14.65	6.63	18.22	6.29	13.47	6.10	18.45
GridGraph	2.97	4.39	2.21	12.86	3.36	4.67	2.30	14.21
Twitter								
GraphChi	437.7	469.8	273.1	1263	443.3	406.1	220.7	1064
X-Stream	435.9	1199	143.9	1779	408.8	1089	128.3	1634
GridGraph	204.8	286.5	50.13	538.1	196.3	276.3	42.33	482.1
UK								
GraphChi	2768	1779	412.3	2083	3203	1709	401.2	2191
X-Stream	8081	12057	383.7	4374	7301	11066	319.4	4015
GridGraph	1843	1709	116.8	1347	1730	1609	97.38	1359
Yahoo								
GraphChi	-	114162	2676	13076	-	106735	3110	18361
X-Stream	-	-	1076	9957	-	-	1007	10575
GridGraph	16815	3602	263.1	4719	30178	4077	277.6	5118

Table 1: Execution time (in seconds) with 8 GB memory. "-" indicates that the corresponding system failed to finish execution in 48 hours.

GraphChi runs all algorithms in asynchronous mode, and an in-memory optimization is used when the number of vertices are small enough, so that vertex data can be allocated and hold in memory and thus edges are not modified during computation, which contributes a lot to performance on Twitter and UK graph.

Table 1 presents the performance of chosen algorithms on different graphs and systems with memory limited to 8GB to illustrate the applicability. Under this configuration, only the LiveJournal graph can be fit into memory, while other graphs require access to disks. We can see that GridGraph outperforms GraphChi and X-Stream on both HDD based d2.xlarge and SSD based i2.xlarge instances, and the performance does not vary much except for BFS on Yahoo, which lots of seek is experienced during the computation, thus making SSDs more advantageous than HDDs. In fact, sometimes better results can be achieved on d2.xlarge instance due to the fact that the peak sequential bandwidth of 3 HDDs on d2.xlarge is slightly greater than that of 1 SSD on i2.xlarge.

Figure 5 shows the disk bandwidth usage of 3 systems, which records the I/O throughput of a 10-minute interval running PageRank on Yahoo graph, using a d2.xlarge instance. X-Stream and GridGraph are available to exploit high sequential disk bandwidth while GraphChi is not so ideal due to more fragmented reads and writes across many shards. GridGraph try to minimize write amount thus more I/O is spent on read while X-Stream has to write a lot more data.

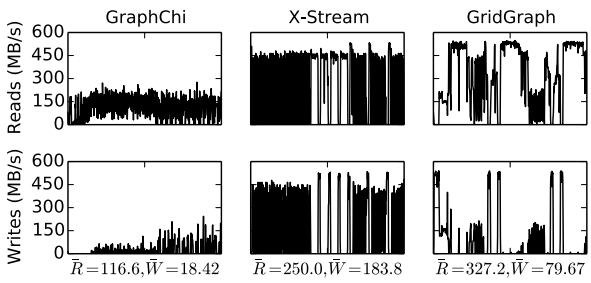
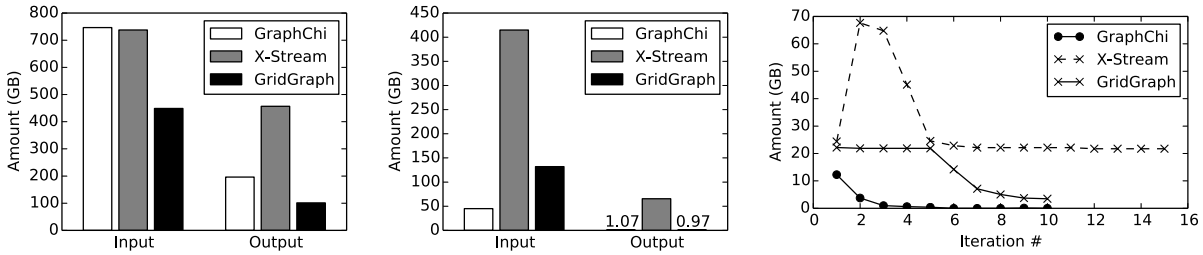


Figure 5: Disk bandwidth usage chart of a 10-minute interval on GraphChi, X-Stream and GridGraph. $\bar{R}$ = average read bandwidth; $\bar{W}$ = average write bandwidth.

For algorithms that all the vertices are active in computation, like SpMV and PageRank (thus every edge is streamed), GridGraph has significant reduction in I/O amount that is needed to complete computation. GraphChi needs to read from and write to edges to propagate vertex states, thus $2 \times E$ I/O amount to edges are required. X-Stream needs to read edges and generate updates in scatter phase, and read updates in gather phase, thus a total I/O amount of $3 \times E$ is required (note that the size of updates is in the same magnitude as edges). On the other hand, GridGraph only requires one read pass over the edges and several passes over the vertices. The write amount is also optimized in GridGraph, which



(a) PageRank on Yahoo graph (b) WCC on Twitter graph (c) WCC on Twitter graph

Figure 6: I/O amount comparison.

only one pass over the vertex data is needed. Figure 6(a) shows the I/O amount that each system needs to complete 5 iterations of PageRank on Yahoo graph. We can see that the input amount of GridGraph is about 60 percent of X-Stream and GraphChi, while the output amount of GridGraph is about 4.5 times less. Thanks to the nice vertex access pattern of dual sliding windows, vertex updates are aggregated within a chunk, which can be a very useful feature in practical use. For graphs that all the vertex data can be cached in memory (like in Figure 6(b)), only one disk read in the first iteration and one disk write after the final iteration is required no matter how many iterations are needed.

For iterative algorithms that only a portion of the whole vertex set participates in the computation of some iterations, such as BFS and WCC, GridGraph can benefit from another good property of grid partitioning that we can skip lots of useless reads with selective scheduling. We compare the I/O amount versus X-Stream and GraphChi using WCC on Twitter graph in Figure 6(b), and provide a per iteration I/O amount in Figure 6(c). We can see that I/O amount decreases with the convergence of the algorithm. In fact, when the volume of active edges reaches a certain level such that the data size is smaller than memory capacity, the page cache could buffer almost all of the edges that might be needed in latter iterations, thus complete the following computations very fast. This phenomenon is more obvious in web graphs than in social graphs, since the graph diameter is much bigger. GraphChi also supports selective scheduling, and its shard representation can have even better effect than GridGraph on I/O reduction. Though I/O amount required by GraphChi is rather small, it has to issue many fragmented reads across shards. Thus the performance is not ideal enough due to limited bandwidth usage.

Another interesting observation from Figure 6(c) is that GridGraph converges faster than X-Stream. This is due to the availability to support asynchronous update in GridGraph that applied updates can be used directly in the same iteration. In WCC, we always push the latest label through edges, which can speed up the label propagation process.

We conclude that GridGraph can perform well on large-scale real world graphs with limited resource. The reduction in I/O amount is the key to the performance gain.

4.3 Preprocessing Cost

Table 3 shows the preprocessing cost of GraphChi and GridGraph on i2.xlarge (SSD) and d2.xlarge (HDD) instances[5]. For SSDs, we only need to partition the edges and append them to different files. For HDDs, a merging phase that combines all the edge block files is required after partitioning. It is known that HDDs do not perform well on random access workloads due to high seek time and low I/O concurrency while SSDs do not have such severe performance penalty. As P increases, edge blocks become smaller and the number of blocks increases, thus making it hard for HDDs to achieve full sequential bandwidth since more time will be spent on seeking to potentially different positions. By merging all the blocks together and use offsets to indicate the region of each block, we can achieve full sequential throughput and benefit from selective scheduling at the same time.

We can see that GridGraph outperforms GraphChi in preprocessing time. GridGraph uses a lightweight range based partitioning, thus only one sequential read over the input edge list and append-only sequential writes to $P \times P$ edge block files are required, which can be handled very well by operating systems.

Figure 7 shows the preprocessing cost on a d2.xlarge instance using Twitter graph with different selections of P. We can see that as P becomes larger, the partitioning time and especially the merging time becomes longer, due to the fact that more small edge blocks are generated. Yet we can see the necessity of this merging phase on

[5]X-Stream does not require explicit preprocessing. Its preprocessing is covered in the first iteration before computation.

	C (S)	G (S)	C (H)	G (H) P	G (H) M	G (H) A
LiveJournal	14.73	1.99	13.06	1.64	1.02	2.66
Twitter	516.3	56.59	425.9	76.03	117.9	193.9
UK	1297	153.8	1084	167.6	329.7	497.3
Yahoo	2702	277.4	2913	352.5	2235.6	2588.1

Table 3: Preprocessing time (in seconds) for GraphChi and GridGraph on 4 datasets. C = GraphChi, G = GridGraph; S = SSD, H = HDD; P = time for partitioning phase, M = time for merging phase, A = overall time.

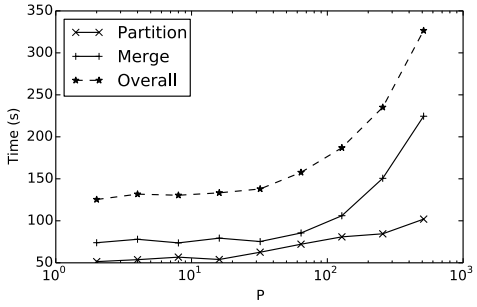


Figure 7: Preprocessing time on Twitter graph with different selections of P (from 2 to 512).

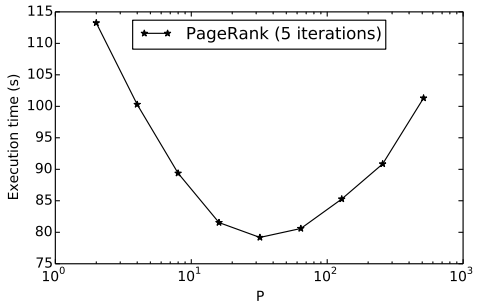


Figure 8: Execution time of PageRank on Twitter graph with different selections of P (from 2 to 512).

HDD based systems since several passes over the edge blocks are often required in multi-iteration computations, which can benefit a lot from this extra preprocessing cost.

We conclude that the preprocessing procedure in GridGraph is very efficient and essential.

4.4 Granularity of Partitioning

First, we evaluate the impact on performance with different selections of P for in-memory situations. Figure 8 shows the execution time of PageRank (5 iterations) on Twitter graph. We use all the available memory (30.5GB) of an i2.xlarge instance, so that the whole graph (including vertex and edge data) can be fit into memory. We can see that P should be neither too small nor too large to achieve a good performance. When P is too small, i.e. each vertex chunk can not be put into last level cache, the poor locality significantly affects the efficiency of vertex access. When P is too large, more data race between atomic operations in each vertex chunk also slows down the performance. Thus we should choose P by considering the size of last level cache as we mentioned in Section 2 to achieve good in-memory performance either when we have a big server with large memory or when we are trying to process a not so big graph that can be fit into memory.

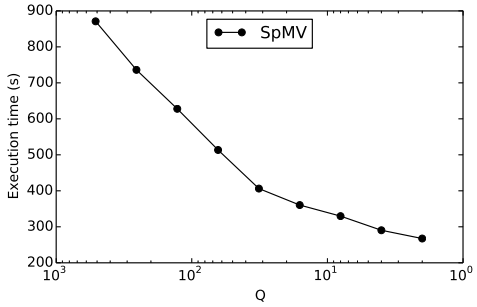


Figure 9: Execution time of SpMV on Yahoo graph with different selections of Q (from 512 to 2).

Next, we evaluate the impact of second level partitioning on performance for out-of-core scenarios. Figure 9 shows the execution time of SpMV on Yahoo graph with different selections of Q. Memory is limited to 8GB on an i2.xlarge instance so that the whole vertex data can not be cached in memory (via memory mapping). As Q decreases, we can see that the execution time descends dramatically due to the fact that a smaller Q can reduce the passes over source vertex data. Thus we should try to minimize Q when data of $\frac{V}{Q}$ vertices can be fit into memory, according to the analysis in Section 3.

We conclude that the 2-level hierarchical partitioning used in GridGraph is very essential to achieve good performance for both in-memory and out-of-core scenes.

Memory	Twitter WCC	Yahoo PageRank
8GB	286.5	1285
30.5GB	120.6	943.1

Table 4: Scalability with Memory. Execution time is in seconds.

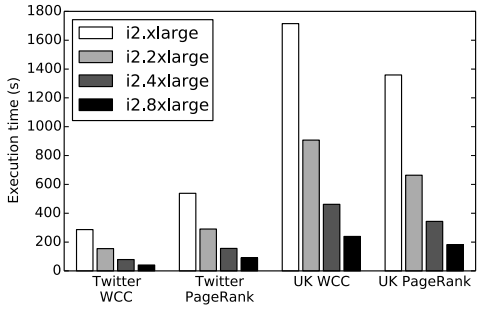


Figure 10: Scalability with I/O.

4.5 Scalability

We evaluate the scalability of GridGraph by observing the improvement when more hardware resource is added.

Table 4 shows the performance variance of WCC on Twitter graph and PageRank (5 iterations) on Yahoo graph when usable memory increases. With the memory increased from 8GB to 30.5GB, the whole undirected Twitter graph (22GB) can be fit into memory, so that edge data is read from disks only once. Meanwhile, with 30.5GB memory, the whole vertex data of Yahoo graph (16GB) can be cached in memory via memory mapping, thus only one pass of read (when program initializes) and write (when program exits) of vertex data is required.

We also evaluate the performance improvement with disk bandwidth. Figure 10 shows the performance when using other I2 instances. Each i2.[n]xlarge instance contains n 800GB SSDs, along with $4 \times n$ hyperthread vCPU cores, and $30.5 \times n$ RAM. Disks are set up as a software RAID-0 array. We do not limit the memory that GridGraph can use, but force direct I/O to the edges to bypass the effect of page cache. We can see that GridGraph scales almost linearly with disk bandwidth.

We conclude that GridGraph scales well when more powerful hardware resource can be utilized.

4.6 Comparison with Distributed Systems

From the result in Figure 10, we find that the performance of GridGraph is even competitive with distributed graph systems. Figure 11 shows the performance comparison between GridGraph using an i2.4xlarge instance (containing 16 hyperthread cores, 122GB RAM, 4 800GB SSDs, $3.41/h), versus PowerGraph and GraphX on a cluster with 16 m2.4xlarge instances (each with 8 cores, 68.4GB RAM, 2 840GB HDDs, $0.98/h), using the result from [28]. We can find that even single-node disk based solutions can provide good enough performance (note that we do edge streaming via direct I/O in Figure 10) and significant reduction in cost ($3.41/h vs. $15.68/h). Moreover, GridGraph is a very convenient single-node solution to use and deploy, thus reducing the effort that cluster-based solutions concerns about. In fact, limited scaling is observed in distributed graph engines ([28]) due to high communication overhead relative to computation in many graph algorithms while GridGraph can scale smoothly as the memory and I/O bandwidth being increased.

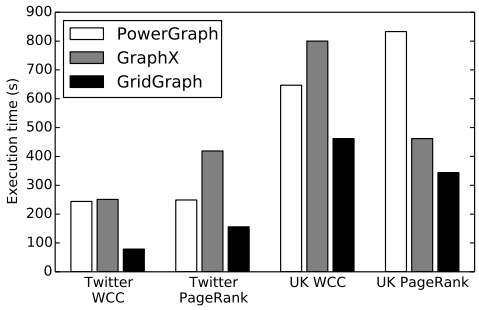


Figure 11: Performance comparison with PowerGraph and GraphX.

We conclude that GridGraph is competitive even with distributed systems when more powerful hardware is available.

5 Related Work

While we have discussed GraphChi and X-Stream in detail, there are other out-of-core graph engines using alternative approaches. TurboGraph [9] manages adjacency lists in pages, issues on-demand I/O requests to disks, and employs a cache manager to maintain frequently used pages in memory to reduce disk I/O. It is efficient for targeted queries like BFS while for applications that require global updates, the performance might degrade due to frequent accesses to the large vector backed on disks. FlashGraph [34] implements a semi-external memory graph engine which stores vertex states in memory and adjacency lists on SSDs, and presents impressive performance. Yet it lacks the ability to process extremely large graphs of which vertices can not be fit into memory. MMap [15] presents a simple approach by leveraging memory mapping mechanism in operating

systems by mapping edge and vertex data files in memory. It provides good programmability and is efficient when memory is adequate, while may suffer from drastic random writes when memory is not enough due to the random vertex access pattern. These solutions require a sorted adjacency list representation of graph, which needs time-consuming preprocessing, and SSDs to efficiently process random I/O requests. GraphQ [26] divides graphs into partitions and uses user programmable heuristics to merge partitions. It aims to answer queries by analyzing subgraphs. PathGraph [31] uses a path-centric method to model a large graph as a collection of tree-based partitions. Its compact design in storage allows efficient data access and achieves good performance on machines with sufficient memory. Galois [18] provides a machine-topology-aware scheduler, a priority scheduler and a library of scalable data structures, and uses a CSR format of graphs in its out-of-core implementation. GridGraph is inspired by these works on out-of-core graph processing, such as partitioning, locality and scheduling optimization, but is unique in its wide applicability and hardware friendliness that only limited resource is required and writes to disks are optimized. This not only provides good performance, but also protects disks from worn-out, especially for SSDs.

There are many graph engines using shared memory configurations. X-Stream [21] has its in-memory streaming engine and uses a parallel multistage shuffler to fit vertex data of each partition into cache. Ligra [23] is a shared-memory graph processing framework which provides two very simple routines for mapping over vertices and edges, inspiring GridGraph for the streaming interface. It adaptively switches between two modes based on the density of active vertex subsets when mapping over edges, and is especially efficient for applications like BFS. Polymer [32] uses graph-aware data allocation, layout and access strategy that reduces remote memory accesses and turns inevitable random remote accesses into sequential ones. While GridGraph concentrates on out-of-core graph processing, some of the techniques in these works can be integrated to improve in-memory performance further.

The 2D grid partitioning used in GridGraph is also utilized similarly in distributed graph systems and applications [10, 28, 30] to reduce communication overhead. PowerLyra [6] provides an efficient hybrid-cut graph partitioning algorithm which combines edge-cut and vertex-cut with heuristics that differentiate the computation and partitioning on high-degree and low-degree vertices. GridGraph uses grid partitioning to optimize the locality of vertex accesses when streaming edges and applies a novel 2-level hierarchical partitioning to adapt to both in-memory and out-of-core situations.

6 Conclusion

In this paper, we propose GridGraph, an out-of-core graph engine using a grid representation for large-scale graphs by partitioning vertices and edges to 1D chunks and 2D blocks respectively, which can be produced efficiently through a lightweight range-based shuffling. A second logical level partitioning is applied over this grid partitioning and is adaptive to both in-memory and out-of-core scenarios.

GridGraph uses a new streaming-apply model that streams edges sequentially and applies updates onto vertices instantly. By streaming edge blocks in a locality-friendly manner for vertices, GridGraph is able to access the vertex data in memory without involving I/O accesses. Furthermore, GridGraph could skip over unnecessary edge blocks. As a result, GridGraph achieves significantly better performance than state-of-the-art out-of-core graph systems, such as GraphChi and X-Stream, and works on both HDDs and SSDs. It is particularly interesting to observe that in some cases, GridGraph is even faster than mainstream distributed graph processing systems that require much more resources.

The performance of GridGraph is mainly restricted by I/O bandwidth. We plan to employ compression techniques[5, 14, 24] on the edge grid to further reduce the I/O bandwidth required and improve efficiency.

Acknowledgments

We sincerely thank our shepherd Haibo Chen and the anonymous reviewers for their insightful comments and suggestions. This work is partially supported by the National Grand Fundamental Research 973 Program of China (under Grant No. 2014CB340402) and the National Natural Science Foundation of China (No. 61232008).

References

[1] AVERY, C. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* (2011).

[2] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, pp. 44–54.

[3] BERTSEKAS, D. P., AND TSITSIKLIS, J. N. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[4] BOLDI, P., SANTINI, M., AND VIGNA, S. A large time-aware web graph. In *ACM SIGIR Forum* (2008), vol. 42, ACM, pp. 33–38.

[5] BOLDI, P., AND VIGNA, S. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 595–602.

[6] CHEN, R., SHI, J., CHEN, Y., AND CHEN, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 1.

[7] FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review* (1999), vol. 29, ACM, pp. 251–262.

[8] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), vol. 12, p. 2.

[9] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (2013), ACM, pp. 77–85.

[10] JAIN, N., LIAO, G., AND WILLKE, T. L. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 4.

[11] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[12] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[13] KYROLA, A., BLELLOCH, G. E., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *OSDI* (2012), vol. 12, pp. 31–46.

[14] LENHARDT, R., AND ALAKUIJALA, J. Gipfeli-high speed compression algorithm. In *Data Compression Conference (DCC), 2012* (2012), IEEE, pp. 109–118.

[15] LIN, Z., KAHNG, M., SABRIN, K. M., CHAU, D. H. P., LEE, H., AND KANG, U. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *Big Data (Big Data), 2014 IEEE International Conference on* (2014), IEEE, pp. 159–164.

[16] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment 5*, 8 (2012), 716–727.

[17] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[18] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 456–471.

[19] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web.

[20] RANDLES, M., LAMB, D., AND TALEB-BENDIAB, A. A comparative study into distributed load balancing algorithms for cloud computing. In *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on* (2010), IEEE, pp. 551–556.

[21] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[22] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 505–516.

[23] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 135–146.

[24] SHUN, J., DHULIPALA, L., AND BLELLOCH, G. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Proceedings of the IEEE Data Compression Conference (DCC)* (2015).

[25] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[26] WANG, K., XU, G., SU, Z., AND LIU, Y. D. Graphq: Graph query processing with abstraction refinement. In *USENIX ATC* (2015).

[27] WANG, P., ZHANG, K., CHEN, R., CHEN, H., AND GUAN, H. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (2014), IEEE, pp. 562–573.

[28] XIN, R. S., GONZALEZ, J. E., FRANKLIN, M. J., AND STOICA, I. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (2013), ACM, p. 2.

[29] YAHOO. Yahoo! altavista web page hyperlink connectivity graph, circa 2002. `http://webscope.sandbox.yahoo.com/`.

[30] YOO, A., CHOW, E., HENDERSON, K., MCLENDON, W., HENDRICKSON, B., AND CATALYUREK, U. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (2005), IEEE Computer Society, p. 25.

[31] YUAN, P., ZHANG, W., XIE, C., JIN, H., LIU, L., AND LEE, K. Fast iterative graph computation: a path centric approach. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for* (2014), IEEE, pp. 401–412.

[32] ZHANG, K., CHEN, R., AND CHEN, H. Numa-aware graph-structured analytics. In *Proc. PPoPP* (2015).

[33] ZHAO, Y., YOSHIGOE, K., XIE, M., ZHOU, S., SEKER, R., AND BIAN, J. Lightgraph: Lighten communication in distributed graph-parallel processing. In *Big Data (BigData Congress), 2014 IEEE International Congress on* (2014), IEEE, pp. 717–724.

[34] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 45–58.

GraphQ: Graph Query Processing with Abstraction Refinement

Scalable and Programmable Analytics over Very Large Graphs on a Single PC

Kai Wang Guoqing Xu
University of California, Irvine

Zhendong Su
University of California, Davis

Yu David Liu
SUNY at Binghamton

Abstract

This paper introduces *GraphQ*, a scalable querying framework for very large graphs. GraphQ is built on a key insight that many interesting graph properties — such as finding cliques of a certain size, or finding vertices with a certain page rank — can be effectively computed by exploring only a small fraction of the graph, and traversing the complete graph is an overkill. The centerpiece of our framework is the novel idea of *abstraction refinement*, where the very large graph is represented as multiple levels of abstractions, and a query is processed through iterative refinement across graph abstraction levels. As a result, GraphQ enjoys several distinctive traits unseen in existing graph processing systems: query processing is naturally *budget-aware*, friendly for *out-of-core processing* when "Big Graphs" cannot entirely fit into memory, and endowed with strong correctness properties on query answers. With GraphQ, a wide range of complex *analytical queries* over very large graphs can be answered with resources affordable to *a single PC*, which complies with the recent trend advocating single-machine-based Big Data processing.

Experiments show GraphQ can answer queries in graphs 4-6 times bigger than the memory capacity, only in several seconds to minutes. In contrast, GraphChi, a state-of-the-art graph processing system, takes hours to days to compute a whole-graph solution. An additional comparison with a modified version of GraphChi that terminates immediately when a query is answered shows that GraphQ is on average 1.6–13.4× faster due to its ability to process partial graphs.

1 Introduction

Developing scalable systems for efficient processing of very large graphs is a key challenge faced by Big Data developers and researchers. Given a graph analytical task expressed as a set of user-defined functions (UDF), existing processing systems compute a *complete solution* over the input graph. Despite much progress, computing a complete solution is still time-consuming. For example, using a 32-node cluster, it takes Preglix [5], a state-of-the-art graph processing system, more than 2,500 seconds to compute a complete solution (*i.e.*, all communities in the input graph) over a 70GB webgraph for a simple community detection algorithm.

While necessary in many cases, the computation of complete solutions — and the overhead of maintaining them — seems an overkill for many real-world applications. For example, queries such as "find one path between LA and NYC whose length is $\leq$ 3,000 miles" or "find 10 programmer communities in Southern California whose sizes are $\geq$ 1000" have many real-world usage scenarios *e.g.*, any path whose length is smaller than a threshold between two cities is acceptable for a navigation system. Unlike database queries that can be answered by filtering records, these queries need (iterative) computations over graph vertices and edges. In this paper, we refer to such queries as *analytical queries*. Furthermore, it appears that many of them can be answered by exploring only a *small fraction* of the input graph — if a solution can be found in a subgraph of the input graph, why do we have to exhaustively traverse the entire graph?

This paper is a quest driven by two simple questions: given the great number of real-world applications that need analytical queries, can we have a ground-up redesign of graph processing systems — from the programming model to the runtime engine — that can facilitate query answering over *partial graphs*, so that a client application can quickly obtain satisfactory results? If partial graphs are sufficient, can we answer analytical queries on one single PC so that the client can be satisfied without resorting to clusters?

GraphQ This paper presents *GraphQ*, a novel graph processing framework for analytical queries. In GraphQ, an analytical query has the form *"find n entities from the graph with a given quantitative property"*, which is general enough to express a large class of queries, such as page rank, single source shortest path, community detection, connected components, *etc.* A detailed discussion of GraphQ's answerability can be found in §3. At its core, GraphQ features two interconnected innovations:

- A simple yet expressive *partition-check-refine* programming model that naturally supports programmable analytical queries processed through *incremental* accesses to graph data
- A novel *abstraction refinement* algorithm to support efficient query processing, fundamentally decoupling the resource usage for graph processing from the (potentially massive) size of the graph

From the perspective of a GraphQ user, the very large input graph can be divided into *partitions*. How partitions are defined is programmable, and each partition on the high level can be viewed as a subgraph that GraphQ queries operate on. Query answering in GraphQ follows a repeated lock-step *check-refine* procedure, until either the query is answered or the budget is exhausted.

In particular, (1) the *check* phase aims to answer the query over each individual partition without considering inter-partition edges connecting these partitions. A query is successfully answered if a *check* predicate returns true; (2) if not, a *refine* process is triggered to identify a set of inter-partition edges to add back to the graph. These recovered edges will lead to a broader scope of partitions to assist query answering, and the execution loops back to step (1). Both the *check* procedure (determining whether the query is answered) and the *refine* procedure (determining what new inter-partition edges to include) are programmable, leading to a programming model suitable for defining complex analytical queries with significant in-graph computations.

Key to finding the most profitable inter-partition edges to add in each step is a novel *abstraction refinement* algorithm at the core of its query processing engine. Conceptually, the "Big Graph" under GraphQ is summarized into an *abstraction graph*, which can be intuitively viewed as a "summarization overlay" on top of the complete concrete graph (CG). The abstraction graph serves as a compact "navigation map" to guide the query processing algorithm to find profitable partitions for refinement.

Usage Scenarios We envision that GraphQ can be used in a variety of real-world data analytical applications. Example applications include:

- *Target marketing:* GraphQ can help a business quickly find a target group of customers with given properties;
- *Navigation:* GraphQ can help navigation systems quickly find paths with acceptable lengths
- *Memory-constrained data analytics:* GraphQ can provide good-enough answers for analytical applications with memory constraints

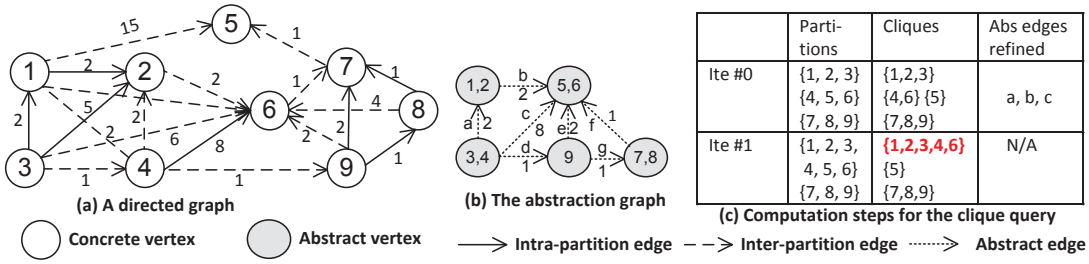
Contributions To the best of our knowledge, our technique is the first to borrow the idea of abstraction refinement from program analysis and verification [8] to process graphs, resulting in a query system that can quickly find correct answers in partial graphs. While there exists a body of work on graph query systems and graph databases (such as GraphChi-DB [17], Neo4j[1], and Titan[2]), the refinement-based query answering in GraphQ provides several unique features unseen in existing systems.

First, GraphQ reflects a ground-up redesign of graph processing systems in the era of "Big Data": unlike the predominant approach of graph querying where only simple graph analytics—those often involving SQL-like semantics where graph vertices/edges are filtered by meeting certain conditions or patterns [13, 14, 7], GraphQ has a strong and general notion of "answerability" which allows for a much wider range of analytical queries to be performed with flexible in-graph computation (*cf.* §3). Furthermore, the abstraction-guided search process makes it possible to answer a query by exploring the most relevant parts of the graph, while a graph database treats all vertices and edges uniformly and thus can be much less efficient.

Second, the idea of abstraction refinement in GraphQ provides a natural data organization and data movement strategy for designing efficient *out-of-core* Big Data systems. In particular, ignoring inter-partition edges (that are abstracted) allows GraphQ to load one partition at a time and perform vertex-centric computation on it independently of other partitions. The ability of exploring only a small fraction of the graph at a time enables GraphQ to answer queries over very large graphs *on one single PC*, thus in compliance with the recent trend that advocates single-machine-based Big Data processing [16, 23, 29, 17]. While our partitions are conceptually similar to GraphChi's shards (*cf.* §4), GraphQ does not need data from multiple partitions simultaneously, leading to significantly reduced random disk accesses compared to GraphChi's parallel sliding window (PSW) algorithm.

Third, GraphQ enjoys a strong notion of *budget awareness*: its query answering capability grows proportionally with the budget used to answer queries. As the refinement progresses, small partitions are merged into larger ones and it is getting increasingly difficult to load a partition into memory. Allowing a big partition to span between memory and disk is a natural choice (which is similar to GraphChi's PSW algorithm). However, continuing the search after the physical memory is exhausted will involve frequent disk I/O and significantly slow down query processing, rendering GraphQ's benefit less obvious compared to whole-graph computation. Hence, we treat the capacity of the main memory as a *budget* and terminate GraphQ with an out-of-budget failure when a merged partition is too big to fit into memory. There are various trade-offs that can be explored by the user to tune GraphQ; a more detailed discussion can be found at the end of §2.

It is important to note that GraphQ is fundamentally different from approximate computing [3, 30, 6], which terminates the computation early to produce *approximate* answers that may contain errors. GraphQ always produces correct answers for the user-specified query goals,



	Partitions	Cliques	Abs edges refined
Ite #0	{1, 2, 3} {4, 5, 6} {7, 8, 9}	{1,2,3} {4,6} {5} {7,8,9}	a, b, c
Ite #1	{1, 2, 3, 4, 5, 6} {7, 8, 9}	**{1,2,3,4,6}** {5} {7,8,9}	N/A

(c) Computation steps for the clique query

Figure 1: An example graph, its abstraction graph, and the computation steps for finding a clique whose size is ≥5. The answer of the query is highlighted.

but improves the computation scalability and efficiency by finding a scope on the input graph that is sufficient to answer a query.

Summary of Experimental Results Our experimental results show that GraphQ can answer queries in the magnitude of seconds to hundreds of seconds for graphs with several billions of edges. For example, over the twitter-2010 graph, GraphQ quickly identified 64 large communities in *49 seconds* while it took a state-of-the-art system, GraphChi [16], 6.4 hours to compute a whole-graph solution over the same graph. We have also compared GraphQ with a modified version of GraphChi that attempts to answer the same queries by terminating immediately when a satisfiable solution is found. The results demonstrate that GraphQ is, on average, 1.6–13.4× faster than this modified version due to the reduced I/O and computation from the processing of partial graphs. GraphQ is publicly available at https://bitbucket.org/wangk7/graphq.

2 Overview and Programming Model

Background Common to graph processing systems, the graph operated by GraphQ can be mathematically viewed as a directed (sparse) graph, $G = (V, E)$. A value is associated with each vertex $v \in V$, indicating an application-specific property of the vertex. For simplicity, we assume vertex values are labeled from 1 to $|V|$. Given an edge e of the form $u \rightarrow v$ in the graph, e is referred to as vertex v's *in-edge* and as vertex u's *out-edge*. The developer specifies an `update`(v) function, which can access the values of a vertex and its neighboring vertices. These values are fed into a function f that computes a new value for the vertex. The goal of the computation is to "iterate around" vertices to update their values until a global "fixed-point" is reached. This vertex-centric model is widely used in graph processing systems, such as Pregel [20], Pregelix [5], and GraphLab [18].

Figure 1 shows a simple directed graph that we will use as a running example throughout the paper. For each GraphQ query, the user first needs to find a related *base application* that performs whole-graph vertex-centric computation. This is not difficult, since many of these algorithms are readily available. In our example, the base application is Maximal Clique, and the query aims to find a clique whose size is no less than 5 (*i.e.* goal) over the input graph.

GraphQ first divides the concrete graph in Figure 1 (a) into three *partitions* — {1, 2, 3}, {4, 5, 6}, and {7, 8, 9} — a "pre-processing" step that only needs to be performed once for each graph. When the query is submitted, the goal of GraphQ is to use an *abstraction graph* to guide the selection of partitions to be merged, hoping that the query can be answered by merging only a very small number of partitions. Initially, inter-partition edges (shown as arrows with dashed lines) are disabled; they will be gradually recovered.

Programming Model A sample program for answering the clique query can be found in Figure 2. Overall, GraphQ is endowed with an expressive 2-tier programming model to balance simplicity and programmability:

- First, GraphQ *end users* only need to write 2-3 lines of code to submit a query. For example, the end user writes lines 2-5, submitting a `CliqueQuery` to look for `Clique` instances whose size is no fewer than 5 over the `ExampleGraph`.
- Second, GraphQ *library programmers* define how a query can be answered through a flexible programming model that fully supports in-graph computation. In the example, the clique query is defined between lines 7-40, by extending the `Query` and `QueryResult` classes in our library.

We expect regular GraphQ users — those who only care about *what* to query but not *how* to query it — to program only the first tier (between lines 2-5). The appeal of the GraphQ programming model lies in its flexibility. On one hand, the simplicity of the GraphQ first-tier interface is on par with query languages for similar purposes (such as SQL). On the other hand, for programmers concerned with graph processing efficiency, GraphQ provides opportunities for full-fledged programming "under the hood" at the second tier.

```
1
2  // end-user
3  Graph g = new ExampleGraph(); // A partitioned graph
4  CliqueQuery cq = new CliqueQuery(g, 5);
5  List<Clique> qr = cq.submit();
6
7  // library programmer
8  class CliqueQuery extends Query {
9    final Graph G; // graph
10   final int N; // goal
11   final int M; // max # of results to refine with
12   final int K; // max # of partitions to merge
13   final int delta; // the inc over K at each refinement
14
15   List<Partition> initPartitions()
16      { return g.partitions; }
17
18   boolean check(Clique c) {
19      if (c.size()>=N) { report(c); return true; }
20   }
21
22   List<AbstractEdge> refine(Clique c1, Clique c2) {
23       List<AbstractEdge> list;
24       foreach(Vertex v in c1.vertices())
25         foreach(Vertex u in c2.vertices())
26           AbstractEdge ae = g.abstractEdge(u, v);
27           if (ae != null) { list.add(ae); }
28       return list;
29   }
30
31   int resultThreshold() { return M; }
32   int partitionThreshold() { return K; }
33
34   CliqueQuery(Graph g, int n) {
35      this.G = g; this.N = n;
36   }
37 }
38 class Clique extends QueryResult {
39   int refinePriority()  { return size(); }
40   int size() {...}
41 }
```

Figure 2: Programming for answering clique Queries.

Partitions Given a very large graph, one can specify how it is partitioned using GraphQ parameters. A partition is both a logical and a physical concept. Logically, a partition is a subgraph (connected component) of the concrete graph. Physically, it is often aligned with the physical storage unit of data, such as a disk file. In our formulation where the graph vertices are labeled with numbers from 1 to $|V|$, we select partitions as containing vertices with continuous label numbers, and edges connecting those vertices in the concrete graph. Beyond this mathematical formulation is an intuitive goal: if we use labels 1 to $|V|$ to mimic the physical sequence of vertex storage, the partitions should be created to be as aligned with physical storage as possible. Thanks to this design, loading a partition is very efficient due to sequential disk accesses with strong locality.

When a query is defined — such as `CliqueQuery` — the programmer first decides what partitions should be *initially* considered to compute local solutions (*e.g.* cliques). This is supported by overriding the `initPartitions` method of the `Query` class, as in line 16. In our example, this method selects all partitions because we have no knowledge of whether and what cliques exist in each partition initially. GraphQ loads one partition into memory at a time and performs vertex-centric computation on the partition to compute local cliques independently of other partitions.

Observe that this does not contradict with our early discussion of incremental graph data processing: at the local computation phase, all partition-based computations are independent of each other. Therefore, when the data for one partition is loaded, the data for previously loaded partitions can be written back to disk, and at this phase GraphQ does not need to hold data in memory for more than one partition. Overall, this phase is very efficient because all inter-partition edges are ignored and there are only a very small number of random disk accesses.

Abstraction Graph The abstraction graph (AG) summarizes the concrete graph. Each *abstract vertex* in the AG abstracts a set of concrete vertices and each *abstract edge* connects two abstract vertices. An abstract edge can have an *abstract weight* that abstracts the weights of the actual edges it represents, which we have omitted in this short example.

To see the motivation behind the design of AG, observe that inter-partition edges can scatter across the partitions (*i.e.*, disk files) they connect, and knowing whether a concrete edge exists between two partitions requires loading both partitions into memory and a linear scan of them, a potentially costly step with a large number of disk accesses. As a "summarization" of the concrete graph, the AG is much smaller in size and can be *always* held in memory.

GraphQ first checks the existence of an abstract edge on the AG: the non-existence of an abstract edge between two abstract vertices $\bar{u}$ and $\bar{v}$ guarantees the non-existence of a concrete edge between any pair of concrete vertices (u, v) abstracted by $\bar{u}$ and $\bar{v}$; hence, we can safely skip the check of concrete edges. On the other hand, the existence of an abstract edge does not necessarily imply the existence of a concrete edge, and hence, the abstract edge needs to be *refined* to recover the concrete edges it represents.

The granularity of the AG is a design issue to be determined by the user. At one extreme, each partition can be an abstract vertex in the AG. This very coarse-grained abstraction may not be precise enough for GraphQ to quickly eliminate infeasible concrete edges. At the other extreme, a very fine-grained AG may take much space and the computation over the AG (such as a lookup) may take time. Since the AG is always in memory to provide quick guidance, a rule of thumb is to allow the abstraction granularity (*i.e.*, the number of concrete vertices represented by one abstract vertex) to be proportional to the memory capacity.

Using parameters, the user can specify the ratio between the size of the AG and the main memory — the

more memory a system has, the larger AG will be constructed by GraphQ to provide more precise guidance. Figure 1 (b) shows the AG for the concrete graph in Figure 1 (a). The GraphQ runtime uses the simple *interval domain* [9] to abstract concrete vertices — each abstract vertex represents two concrete vertices that have consecutive labels. This simple design turns out to be friendly for performance as well: each abstract edge represents a set of concrete edges stored together in the partition file; since refining an abstract edge needs to load its corresponding concrete edges, storing/loading these edges together maximizes sequential disk accesses and data locality. A detailed explanation of the storage structure can be found in §4.

An alternative route we decide not to pursue is to provide the user full programmability to construct their own AGs. The issue at concern is *correctness*. Our design of the abstraction graph is built upon the principled idea of abstraction refinement, with correctness guarantees (§3). The correctness is hinged upon that the AG is indeed a "sound" abstraction of the concrete graph. We rely on the GraphQ runtime to maintain this notion of sound abstraction.

Abstraction Refinement At the end of each local computation (*i.e.*, over a partition), GraphQ invokes the `check` method of the `Query` object. The method returns **true** if the query can be answered, and the result is reported through the `report` method (see line 19). Query processing terminates. If all local computations are complete and all `check` invocations return **false**, GraphQ tries to merge partitions to provide a larger scope for query answering. Recall that in our initial partition definition, all inter-partition edges have been ignored. The crucial challenge of partition merging thus becomes recovering the inter-partition edges, a process we call *abstraction refinement*.

In GraphQ, the refinement process is guided by the `QueryResult` — `Clique` in our example — from local computations. The key insight is that the results so far should offer clues on which partitions should be merged at a higher priority. The "priority" here can be customized by programmers through overriding the `refinePriority` method of class `QueryResult`. In the clique query example here, the programmer uses the size of the clique as the metric for priority (see line 39). Intuitively, merging partitions where larger cliques have been discovered is more likely to reach the goal of finding a clique of a certain size.

GraphQ next selects M (returned by `resultThreshold` in line 31) results with the highest priorities (*i.e.* largest cliques) for pairwise inspection. For each pair of cliques resulting from different partitions, the `refine` method (line 22) is invoked to verify if there is any potential for the two input cliques to combine into a larger clique. `refine` returns a list of abstract edges that should be refined. The implementation of `refine` is provided by programmers, typically involving the consultation of the AG. In our example, the method returns a list of candidate abstract edges whose corresponding concrete edges may potentially connect vertices from the two existing cliques (in two partitions) in order to form a larger clique.

Based on the returned abstract edges, GraphQ consults the AG to find the concrete edges these abstract edges represent. GraphQ then merges the partitions in which these concrete edges are located. To avoid a large number of partitions to be merged at a time — that would require the data associated all partitions to be loaded into memory at the same time — programmers can set a threshold specified by `partitionThresold`, in line 32. GraphQ adopts an iterative merging process: in each pass, merging only happens when the refinement leads to the merging of no more than K (returned by `partitionThresold`) partitions. If the merged partitions cannot answer the queries, GraphQ increases K by δ (line 13) at each subsequent pass to explore more partitions. This design enables GraphQ to gradually use more memory as the query processing progresses.

GraphQ terminates query processing in one of the 3 scenarios: (1) the `check` method returns **true**, in which case the query is answered; (2) all partitions are merged in one, and the `check` method still returns **false** — a situation in which this query is impossible to answer; and (3) a (memory) budget runs out, in which case GraphQ returns the best `QueryResults` that have been found *so far*. We will rigorously define this notion in §3.

Example Figure 1 (c) shows the GraphQ computational steps for answering the clique query. The three columns in the table show the partitions considered in the beginning of each iteration, the local maximal cliques identified, and the abstract edges selected by GraphQ to refine at the end of the iteration, respectively. Before iteration #0, the user selects all the three partitions via `initPartitions`. The vertex-centric computation of these partitions identifies four local cliques $\{1, 2, 3\}$, $\{4, 6\}$, $\{5\}$, and $\{7, 8, 9\}$.

Since the `check` function cannot find a clique whose size is ≥ 5, GraphQ ranks the four local cliques based on their sizes (by calling `refinePriority`) and invokes `refine` five times with the following clique pairs: ($\{1, 2, 3\}$, $\{7, 8, 9\}$), ($\{1, 2, 3\}$, $\{4, 6\}$), ($\{4, 6\}$, and $\{7, 8, 9\}$), ($\{5\}$,$\{1, 2, 3\}$), ($\{5\}$,$\{7, 8, 9\}$). For instance, for input ($\{1, 2, 3\}$, $\{7, 8, 9\}$), no abstract edge exists on the AG that connects any vertex in the first clique with any vertex in the second. Hence, `refine` returns an empty list.

For input ($\{1, 2, 3\}$, $\{4, 6\}$), however, GraphQ detects that there is an abstract edge between every abstract

vertex that represents {1, 2, 3} and every abstract vertex that represents {4, 6}. The abstract edges connecting these two cliques (*i.e.*, *a*, *b*, and *c*) are then added into list `list` and returned.

After checking all pairs of cliques, GraphQ obtains 6 lists of abstract edges, among which five span two partitions and one spans three. Suppose K is 2 at this moment. The one spanning three partitions is discarded. For the remaining five lists, (*a*, *b*, *c*) is the first list returned by `refine` (on input ({1, 2, 3}, {4, 6})). These three abstract edges are selected and their refinement adds the following four concrete (inter-partition) edges back to the graph: 4→2, 3→4, 1→5, and 2→6. The second iteration repeats vertex-centric computation by considering a merged partition {1, 2, 4, 5, 6}. When the partition is processed, a new clique {1, 2, 3, 4, 6} is found. Function `check` finds that the clique answers the query; so it reports the clique and terminates the process.

Programmability Discussions In addition to answering queries with user-specified goals, our programming model can also support aggregation queries (min, max, average, *etc*.). For example, to find the largest clique under a memory budget, only minor changes are needed to the `CliqueQuery` example. First, we can define a private field called `max` to the class. Second, we need to update the `check` method as follows:

```
if(c.size()>max)
  {max=c.size(); return false;}
```

The observation here is that `check` should always return **false**. GraphQ will continue the refinement until the (memory) budget runs out, and the result `c` aligns with our intuition of being "the largest `Clique` under the budget based on the user-specified refinement heuristics", a flavor of the budget-aware query processing.

GraphQ can also support *multiplicity* of results, such as the top 30 largest cliques. This is just a variation of the example above. Instead of reporting a clique `c`, the `CliqueQuery` should maintain a "top 30" list, and use it as the argument for `report`.

Trade-off Discussions It is clear that GraphQ provides several trade-offs that the user can explore to tune its performance. First, the memory size determines GraphQ's answerability. A higher budget (*i.e.* more memory) will lead to (1) finding more entities with higher goals, or (2) finding the same number of entities with the same goals more quickly. Since GraphQ can be embedded in a data analytical application running on a PC, imposing a memory budget allows the application to perform intelligent resource management between GraphQ an other parts of the system, obtaining satisfiable query answers while preventing GraphQ from draining the memory.

Another tradeoff is defined by abstraction granularity, that is, the ratio between the size of the AG and the memory size. The larger this ratio is, the more precise guidance the AG provides. On the other hand, holding a very large AG in memory could hurt performance by eclipsing the memory that could have been allocated for data loading and processing. Hence, achieving good performance dictates finding the sweetspot.

3 Abstraction-Guided Query Answering

This section formally presents our core idea of applying abstracting refinement to graph processing. In particular, we rigorously define GraphQ's answerability.

Definition 3.1 (Graph Query). *A user query is a 5-tuple* $(\Delta, \phi, \pi, \diamond, g)$ *that requests to find, in a directed graph* $G = (V_G, E_G)$, Δ *entities satisfying a pair of predicates* $\langle \phi, \pi \diamond g \rangle$. *Definition predicate* $\phi \in \Phi$ *is a logical formula* ($\mathbb{P}(G) \to \mathbb{B}$) *over the set of all G's subgraphs that defines an entity,* $\pi \in \Pi$ *is a quantitative function* ($\mathbb{P}(G) \to \mathbb{R}$) *over the set of subgraphs satisfying* ϕ*, measuring the entity's* size*, and* $\diamond$ *is a numerical comparison operator* (e.g., $\geq$ *or* $=$) *that compares the output of* π *with a user-specified* goal of the query $g \in \mathbb{R}$.

This definition is applicable to a wide variety of user queries. For example, for the clique query discussed in §2, ϕ is the following predicate on the vertices and edges of a subgraph $S \subseteq G$, defining a clique:
$\forall\, v_1, v_2 \in V_S$: $\exists e \in E_S$: $e = (v_1, v_2) \vee e = (v_2, v_1)$,
while π is a simple function that returns the number of vertices $|V_S|$ in the subgraph. $\diamond$ and g are $\geq$ and 5, respectively. From this point on, we will refer to ϕ and π as the *definition predicate* and the *size function*, respectively.

Definition 3.2 (Monotonicity of the Size Function). *A query* $(\Delta, \phi, \pi, \diamond, g)$ *is GraphQ-answerable if* $\pi \in \Pi$ *is a* monotone *function with respect to operator* $\diamond$*:* $\forall S_1 \in \mathbb{P}(G), S_2 \in \mathbb{P}(G) : S_2 \subseteq S_1 \wedge \phi(S_1) \wedge \phi(S_2) \implies \pi(S_1) \diamond \pi(S_2)$.

While the user can specify an arbitrary size function π or goal g, π has to be *monotone* in order for GraphQ to answer the query. More precisely, for any subgraphs S_1 and S_2 of the input graph G, if S_2 is a subgraph of S_1 and they both satisfy the definition predicate ϕ, the relationship between their sizes $\pi(S_1)$ and $\pi(S_2)$ is $\pi(S_1) \diamond \pi(S_2)$. For example, if S_2 is a clique with N vertices, and S_1 is a supergraph of S_2 and also a clique, S_1's size must be $\geq N$. Monotonicity of the size function implies that once GraphQ finds a solution that satisfies a query at a certain point, the solution will *always* satisfy the query because GraphQ will only find better solutions in the forward execution. It also matches well with the underlying vertex-centric computation model that gradually propagates the information of a vertex to distant vertices (*i.e.*, which has the same effect as considering increasingly large subgraphs).

Definition 3.3 (Partition). *A partition P of graph G is a subgraph (V_P, E_P) of G such that vertices in V_P have contiguous labels [i, i + $|V_P|$], where $i \in I$ is the minimum integer label a vertex in V_P has and $|V_P|$ is the number of vertices of P. A* partitioning *of G produces a set of partitions $P_1, P_2, \ldots P_k$ such that $\forall j \in [1, k-1]$: $max_{v \in V_{P_j}} label(v) + 1 = min_{v \in V_{P_{j+1}}} label(v)$. An edge $e = (v_1, v_2)$ is an* intra-partition *edge if v_1 and v_2 are in the same partition; otherwise, e is an* inter-partition *edge.*

Logically, each partition is defined by a label range, and physically, it is a disk file containing the edges whose targets fall into the range. The physical structure of a partition will be discussed in §4.

Definition 3.4 (Abstraction Graph). *An abstraction graph $(\bar{V}, \bar{E}, \alpha, \gamma)$ summarizes a concrete graph (V, E) using abstraction relation α: $V \rightarrow \bar{V}$. The AG is a* sound *abstraction of the concrete graph if $\forall e = (v_1, v_2) \in E$: $\exists \bar{e} = (\bar{v_1}, \bar{v_2}) \in \bar{E} : \bar{v_1}, \bar{v_2} \in \bar{V} \wedge (v_1, \bar{v_1}) \in \alpha \wedge (v_2, \bar{v_2}) \in \alpha$. γ: $\bar{V} \rightarrow V$ is a* concretization relation *such that $(\bar{v}, v) \in \gamma$ iff $(v, \bar{v}) \in \alpha$.*

α and γ form a *monotone Galois connection* [9] between G and AG (which are both posets). There are multiple ways to define the abstraction function α. In GraphQ, α is defined based on an interval domain [9]. Specifically, each abstract vertex $\bar{v}$ has an associated interval $[i, j]$; $(v, \bar{v}) \in \alpha$ iff $label(v) \in [i, j]$. The primary goal is to make concrete edges whose target vertices have contiguous labels stay together in a partition file. To concretize an abstract edge, GraphQ will only need *sequential accesses* to a partition file, thereby maximizing locality and refinement performance. Different abstract vertices have disjoint intervals. The length of the interval is determined by a user-specified percentage r and the maximum heap size M—the size of the AG cannot be greater than $r \times M$. The implementation details of the partitioning and the AG construction can be found in §4. Clearly, the AG constructed by the interval domain is a sound abstraction of the input graph.

Lemma 3.5 (Edge Feasibility). *If no abstract edge exists from $\bar{v_1}$ to $\bar{v_2}$ on the AG, there must not exist a concrete edge from v_1 to v_2 on the concrete graph such that $(v_1, \bar{v_1}) \in \alpha$ and $(v_2, \bar{v_2}) \in \alpha$.*

The lemma can be easily proved by contradiction. It enables GraphQ to inspect the AG first to quickly skip over infeasible solutions.

Definition 3.6 (Abstraction Refinement). *Given a subgraph S = (V_s, E_s) of a concrete graph G = (V, E) and its AG = ($\bar{V}$, $\bar{E}$) of G, an abstraction refinement $\sqsubseteq$ on S selects a set of abstract edges $\bar{e} \in \bar{E}$ and adds into E_s all such concrete edges e that $e \in E \setminus E_s : (\bar{e}, e) \in \alpha$. An abstraction refinement of the form $S \sqsubseteq S'$ produces a new subgraph $S' = (V'_s, E'_s)$, such that $V_s = V'_s$ and $E_s \subseteq E'_s$. A refinement is an* effective *refinement if $E_s \subset E'_s$.*

The concretization function is used to obtain concrete edges for a selected abstract edge. After an effective refinement, the resulting graph S' becomes a (strict) supergraph of S, providing a larger scope for query answering.

Lemma 3.7 (Refinement Soundness). *An entity satisfying the predicates (ϕ, $\pi \diamond g$) found in a subgraph S is preserved by an abstraction refinement on S.*

The lemma shows an important property of our analysis. Since our goal is to find Δ entities, this property guarantees that the entities we find in previous iterations will stay as we enlarge the scope. The lemma can be easily proved by considering Definition 3.2: since the size function π is monotone, if the predicate $\pi(S) \diamond g$ holds in subgraph S, the predicate $\pi(S') \diamond g$ must also hold in subgraph S' that is a strict supergraph of S. Because S' contains all vertices and edges of S, the fact the definition predicate ϕ holds on S implies that ϕ also holds on S' (*i.e.*, $\phi(S) \implies \phi(S')$).

Definition 3.8 (Essence of Query Answering). *Given an initial subgraph S = (V, E_s) composed of a set $\mathbb{P}$ of disjoint partitions ((V1, E1), ..., (V_j, E_j)) such that V = $V_1 \cup \ldots \cup V_j$ and $E_s = E_1 \cup \ldots \cup E_j$, as well as an AG = ($\bar{V}$, $\bar{E}$), answering a query (Δ, ϕ, π, $\diamond$, g) aims to find a refinement chain $S \sqsubseteq^* S''$ such that there exist at least Δ distinct entities in S'', each of which satisfies both ϕ and $\pi \diamond g$.*

In the worst case, S'' becomes G and graph answering has (at least) the same cost as computing a whole-graph solution. Each refinement step bridges multiple partitions. Suppose we have a partition graph (PG) for G where each partition is a vertex. The refinement chain starts with a *PG* without edges (*i.e.*, each partition is a connected component), and gradually adds edges and reduces the number of connected components. Suppose PG_S is the PG for a subgraph S, ρ is a function that takes a *PG* as input and returns the maximum number of partitions in a connected component of the *PG*, and each initial partition has the (same) size η. We have the following definition:

Definition 3.9 (Budget-Aware Query Answering). *Answering a query under a memory budget M aims to find a refinement chain $S \sqsubseteq^* S''$ such that $\forall (S_1 \sqsubseteq S_2) \in \sqsubseteq^*$: $\eta \times \rho(PG_{S_2}) \leq M$.*

In other words, the number of (initial) partitions connected by each refinement step must not exceed a threshold t such that $t \times \eta \geq M$. Otherwise, the next iteration

would not have enough memory to load and process these t partitions.

Theorem 3.10 (Soundness of Query Answering). *GraphQ either returns correct solutions or does not return any solution if the vertex-centric computation is correctly implemented.*

Limitations Despite its practical usefulness, GraphQ can only answer queries whose vertex update functions are monotonic, while many real-world problems may not conform to this property. For example, for machine learning algorithms that perform probability propagation on edges (*e.g.*, belief propagation and the coupled EM (CoEM)), the probability in a vertex may fluctuate during the computation, preventing the user from formulating a probability problem as GraphQ queries.

4 Design and Implementation

We have implemented GraphQ based on GraphChi [16], a high-performance single-machine graph processing system. GraphChi has both C++ and Java versions; GraphQ is implemented on top of its Java version to provide an easy way for the user to write UDFs. Our implementation has an approximate of 5K lines of code and is available for download on BitBucket. The pre-processing step splits the graph file into a set of small files with the same format, each representing a partition (*i.e.*, a vertex interval). We modify the *shard* construction algorithm in GraphChi to partition the graph. Similarly to a shard in [16], each partition file contains all in-edges of the vertices that logically belong to the partition; hence, edges stored in a partition file whose sources do not belong to the partition are inter-partition edges.

The AG is constructed when the graph is partitioned. To allow concrete edges (*i.e.*, lines in each text file) represented by the same abstract edge to be physically located together, we first sort all edges in a partition based on the labels of their source vertices — it moves together edges from contiguous vertices. Next, for each abstract vertex (*i.e.*, an interval), we sort edges that come from this interval based on the labels of their target vertices — now the concrete edges represented by the same abstract edge are restructured to stay in a contiguous block of the file. This is a very important handling and will allow efficient disk accesses, provided that large graph processing is often I/O dominated.

For example, for an abstract edge [40, 80] → [1024, 1268], its concrete edges are located physically in the partition file containing the vertex range [1024, 1268]. The first sort moves all edges coming from [40, 80] together. However, among these edges, those going to [1024, 1268] and those not are mixed. The second sort moves them around based on their target vertices, and thus, edges going to contiguous vertices are stored contiguously. Although the interval length used in the abstraction is statically fixed (*i.e.*, defined as a user parameter), we do not allow an abstract vertex to represent concrete vertices spanning two partitions — we adjust the abstraction interval if the number of the last set of vertices in a partition is smaller than the fixed interval size.

Each abstract edge consists of the starting and ending positions of the concrete edges it represents (including the partition ID and the line offsets), as well as various summaries of these edges, such as the number of edges, and the minimum and maximum of their weights. The AG is saved as a disk file after the construction. It will be loaded into memory upon query answering. When an (initial or merged) partition is processed, we modify the parallel sliding window algorithm in GraphChi to load the entire partition into memory. In GraphChi, a *memory shard* is a partition being processed while *sliding shards* are partitions containing out-edges for the vertices in the memory shard. Since inter-partition edges are ignored, GraphQ eliminates sliding shards and treats each partition p as a memory shard. The number of random disk accesses at each step thus equals the number of initial partitions contained in p.

The loaded data may include both enabled and disabled edges; the disabled edges are ignored during processing. Initially, all inter-partition edges are disabled. Refining an abstract edge loads the partitions to be merged and enables the inter-partition edges it represents before starting the computation. We treat the refinement process as an *evolving graph*, and modify the incremental algorithm in GraphChi to only compute and propagate values from the newly added edges.

A user-specified ratio r is used to control the size of the AG. Ideally, we do not want the size of the AG to exceed $r\times$ the memory size. However, this makes it very difficult to select the interval size (*i.e.* abstraction granularity) before doing partitioning, because the size of the AG is related to its number of edges and it is unclear how this number is related to the interval size before scanning the whole graph. To solve the problem, we use the following formula to calculate the interval size i: $i = \frac{size(G)}{M \times r}$, under a rough estimation that if the number of vertices is reduced by i times, the number of edges (and thus the size of the graph) is also reduced by i times. In practice, the size of the AG built using i is always close to $M \times r$, although it often exceeds the threshold.

5 Queries and Methodology

We have implemented UDFs for five common graph algorithms shown in Table 1. The pre-processing time is a one-time cost, which does not contribute to the actual query answering time. For PageRank and Path, GraphQ does not need to compute local results; what par-

Name	*Query GraphQ to Find*	*Init*	*RefinePriority*	*GraphChi Time*	*GraphQ Pre-proc. Time*
PageRank	Δ vertices whose pageranks are $\geq N$	none	X-percentages ($\uparrow$)	1754, 2880 secs.	120+0, 200+0 secs.
Clique	Δ cliques whose sizes are $\geq N$	all	clique sizes ($\uparrow$)	5.5, 50.2 hrs.	400+500, 800+1060 secs.
Community	Δ communities whose sizes are $\geq N$	all	community sizes ($\uparrow$)	3.4, 6.4 hrs.	150+200, 300+400 sec.
Path	Δ paths whose lengths are $\leq N$	none	path lengths ($\downarrow$)	?, ?	200+0, 400+0 secs.
Triangle	Δ vertices whose edge triangles are $\geq N$	all	triangle counts ($\uparrow$)	1990, 3194 secs.	200+300, 400+600 secs.

Table 1: A summary of queries performed in the evaluation: reported are the names and forms of the queries, initial partition selection, priority of partition merging, whole-graph computation times in GraphChi for the uk-2005 [4] and the twitter-2010 [15] graphs, and the time for pre-processing them; $\uparrow$ ($\downarrow$) means the higher (lower) the better; each pre-processing time has two components $a+b$, where a represents the time for partitioning and AG construction, and b represents the time for initial (local) computation; "?" means the whole-graph computation cannot finish in 48 hours.

Name	*Type*	$\|V\|$	$\|E\|$	*#IP*	*#MP*	δ
uk-2005 [4]	webgraph	39M	0.75B	50	30	10
twitter-2010 [15]	social network	42M	1.5B	100	50	10

Table 2: Our graph inputs: reported in each section are their names, types, numbers of vertices, numbers of edges, numbers of initial partitions (*IP*), numbers of maximum partitions allowed to be merged before out of budget (*MP*), and numbers of partitions increased at each step (δ, *cf.* line 13 in Figure 2).

titions to be merged can be determined simply based on the structure of each partition. We experimented GraphQ with a variety of graphs. Due to space limitations, this section reports our results with the two largest graphs, shown in Table 2. Since the focus of this work is *not* to improve the whole-graph computation, we have not run other distributed platforms.

PageRank Answering PageRank queries is based on the whole-graph PageRank algorithm used widely to rank pages in a webgraph. The algorithm is not strictly monotone, because vertices with few incoming edges would give out more than they gain in the beginning and thus their pageranks values would drop in the first few iterations. However, after a short "warm-up" phase, popular pages would soon get their values back and their pageranks would continue to grow until the convergence is reached. To get meaningful pagerank values to query upon, we focus on the top 100 vertices reported by GraphChi (among many million vertices in a graph). Their pageranks are very high and these vertices represent the pages that a user is interested in and wants to find from the graph.

Focusing on the most popular vertices also allows us to bypass the non-monotonic computation problem—since the goals are very high, it is only possible to answer a query during monotonic phase (after the non-monotonic warm-up finishes). The refinement logic we implemented favors the merging of partitions that can lead to a larger *X-percentage*. The X-percentage of a partition is defined as the percentage of the outgoing edges of the vertex with the largest degree that stay in the partition. It is a metric that measures the completeness of the edges for the most popular vertex in the partition. The higher the X-percentage is, the quicker it is for the pagerank computation to reach a high value and thus the easier for GraphQ to find popular vertices. PageRank does not need a local phase—from the AG, we directly identify a list of partitions whose merging may yield a large X-percentage.

Clique is based on the Maximal Clique algorithm that computes a maximal clique for each vertex in the graph. Since the input is a directed graph, a set of vertices forms a clique if for each pair of vertices u and v, there are two edges between them going both directions. GraphChi does not support variable-size edge and vertex data, and hence, we used 10 as the upper-bound for the size of a clique we can find. In other words, we associated with each edge and vertex a 44-byte buffer (*i.e.*, 10 vertices take 40 bytes and used an additional 4-byte space in the beginning to save the actual length). Due to the large amount of data swapped between memory and disk, the whole-graph computation over twitter-2010 took more than 2 days.

Path is based on the SSSP algorithm and aims to find paths with acceptable length between a given source and destination. Similarly to Clique, we associated a (fixed-length) buffer with each edge/vertex to store the shortest path for the edge/vertex. Since none of our input graphs have edge weights, we assigned each edge a random weight between 1 and 5. However, the whole-graph computation could not finish processing these graphs in 2 days. To generate reasonable queries for GraphQ, we sampled each graph to get a smaller graph (that is 1/5 of the original graph) and ran the whole-graph SSSP algorithm to obtain the shortest paths between a specified vertex S (randomly chosen) and all other vertices in the sample graph. If there exists a path between S and another vertex v in the small graph, a path must also exist in the original graph. The SSSP computation over even the small graphs took a few hours.

Community is based on a community detection algorithm in which a vertex chooses the most frequent label

of its neighbors as its own label. Triangle uses a triangle counting algorithm that counts the number of edge triangles incident to each vertex. This problem is used in social network analysis for analyzing the graph connectivity properties [27]. For both applications, we obtained their whole-graph solutions and focus on the 100 largest entities (*i.e.*, communities and vertices with most triangles). Community and Triangle favor the merging of partitions that can yield large communities and triangle counts, respectively.

6 Evaluation

Test Setup All experiments were performed on a normal PC with one Intel Core i5-3470 CPU (3.2GHz) and 10GB memory, running Ubuntu 12.04. The JVM used was the HotSpot Client VM (build 24.65-b04, mixed mode, sharing). Some of our results for GraphChi may look different from those reported in [16] due to different versions of GraphChi used as well as different hardware configurations. We have conducted three sets of experiments. First, we performed queries with various goals and Δ to understand the query processing capability of GraphQ. Second, we compared the query answering performance between GraphQ and *GraphChi-ET* (i.e., acronym for "GraphChi with early termination") — a modified version of GraphChi that terminates immediately when a query is answered. Third, we varied the abstraction granularity to understand the impact of abstraction refinement. The first and third sets of experiments ran GraphQ on the PC's embedded 500GB HDD to understand the query performance on a normal PC while a Samsung 850 250GB SSD was used for the second set to minimize the I/O costs, enabling a fair comparison with GraphChi-ET.

6.1 Query Efficiency

In this experiment, the numbers of initial partitions for the two graphs are shown in Table 2. The maximum heap size is 8GB, and the ratio between the AG size and the heap size is 25%. For the two graphs, the maximum number of partitions that can be merged before out of budget is 30 and 50. For each algorithm, GraphQ first performed local computation on initial partitions (as specified by the UDF `initPartitions`). Next, we generated queries whose goals were randomly chosen from different value intervals. Queries with easy goals/small Δ were asked earlier than those with more difficult goals/larger Δ, so that the computation results for earlier queries could serve a basis for later queries (*i.e.*, incremental computation). This explains why answering a difficult query is sometimes faster than answering an easy query (as shown later in this section).

PageRank To better understand the performance, we divided the top 100 vertices (with the highest pagerank values from the whole-graph computation) into several intervals based on their pagerank values. Each interval is thus defined by a pair of lower- and upper-bound pageranks. We generated 20 queries for each interval, each requesting to find Δ vertices with the goal being a randomly generated value that falls into the interval. For each interval reported in Table 3, all 20 queries were successfully answered. The average running time for answering these queries over uk-2005 is shown in the *Time* sections.

Δ	(a) Top20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	56.1	20	5.6	10	3.0	10	4.3	10
2	32.2	20	5.0	10	5.1	10	6.6	10
4	120.0	20	27.0	10	19.2	10	21.6	10
8	350.1	30	182.9	30	54.3	20	41.9	20

Table 3: GraphQ performance for answering PageRank queries over uk-2005; each section shows the performance of answering queries on pagerank values that belong to an interval in the top 100 vertex list; reported in each section are the number of entities requested to find (Δ), the average query answering time in seconds (*Time*), and the number of partitions merged when a query is answered (*Par*).

The largest Δ we have tried is 8—GraphQ ran out of budget for most of the queries when a larger Δ was used. When $\Delta \leq 4$, GraphQ could successfully answer all queries even including those from the top 10 category. For twitter-2010, GraphQ always failed on queries whose goals were selected from the top 10 category. Otherwise, it successfully answered all queries. For example, the average time for answering 8 queries whose goals are from the top 10-20 category is 754.7 seconds.

Clique The biggest clique found in twitter-2010 (by the 52-hour whole-graph computation) has 6 vertices and there are totally 66 of them. The (relatively small) size of the maximum clique is expected because a clique in a directed graph has a stronger requirement: bi-directional edges must exist between each pair of vertices. The largest Δ we have tried is 64. Table 4 shows GraphQ's performance as we changed Δ; the running time reported is the average time across answering 20 queries in each interval. GraphQ could easily find 8 of the 66 6-clique (in 823 seconds), but the time increased significantly when we asked for 16 of them. GraphQ could find no more than 26 6-cliques before running out of budget. If a user is willing to sacrifice her goal and look for smaller cliques (say 5-cliques), GraphQ can find 64 of them in 460 seconds (by merging only 10 partitions).

Community The whole-graph computation of community detection took 1.5 hours on uk-2005 and 6.4 hours on twitter-2010. Similarly to PageRank, we

Δ	(a) *Size = 6*		(b) *Size = 5*		(c) *Size = 4*		(d) *Size = 3*	
	Time	*Par*	*Time*	*Par*	*Time*	Par	*Time*	*Par*
1	98.3	10	2.0	10	2.0	10	2.0	10
2	248.1	10	2.0	10	2.3	10	2.0	10
4	489.5	20	2.1	10	2.0	10	8.3	10
8	823.9	20	51	10	2.1	10	8.2	10
16	5960.3	30	49.1	10	2.1	10	9.6	10
32	-	50	144.1	10	2.8	10	16.4	10
64	-	50	460.0	10	128.3	10	20.0	10

Table 4: GraphQ's performance for answering Clique queries over twitter-2010; a "-" sign means some queries in the group could not be answered.

focused on the top 100 largest communities and asked GraphQ for communities of different sizes (that belong to different intervals on the top 100 list). For each interval, we picked 20 random sizes to run GraphQ and the average running time over uk-2005 is reported in Table 5. The whole-graph result shows that there are a few (less than 10) communities that are much larger than the other communities on the list. These communities have many millions of vertices and none of them could be found by GraphQ before the budget ran out. Hence, Table 5 does not include any measurement for queries with a size that belongs to the top 10 interval.

Δ	(a) Top10-20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	8.2	10	4.9	10	4.3	10	4.5	10
2	51.8	10	34.5	10	20.1	10	14.2	10
4	142.1	20	63.3	10	27.1	10	25.4	10
8	292.3	20	160.6	20	56.9	10	35.5	10
16	563.4	30	236.7	30	196.7	20	97.7	20
32	-	30	-	30	-	30	332.8	30

Table 5: GraphQ's performance for answering Community queries over uk-2005; each section reports the average time for finding communities whose sizes belong to different intervals in the top 100 community list.

Interestingly, we found that GraphQ performed much better over twitter-2010 than uk-2005: for twitter-2010, GraphQ could easily find (in 162.1 seconds) 256 communities from the top 10-20 range by merging only 20 partitions as well as 1024 communities (in 188.2 seconds) from the top 20-40 range by merging 20 partitions. This is primarily because twitter-2010 is a social network graph in which communities are much "better defined" than a webgraph such as uk-2005.

Path We inspected the whole-graph solution for each sample graph (*cf.* §5) and found a set t of vertices v such that the shortest path on the small graph between S (the source) and each v is between 10 and 25 and contains at least 5 vertices. We randomly selected 20 vertices u from t and queried GraphQ for paths between S and u over the original graph. Based on the length of their shortest paths on the small graph, we used 10, 15, 20, and 25 as the goals to perform queries (recall that each edge has an artificial length between 1 and 5). The average time to answer these queries on twitter-2010 is reported in Figure 6.

Δ	(a) 10		(b) 15		(c) 20		(d) 25	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	59.5	10	57.6	10	58.1	10	45.2	10
2	55.5	20	53.2	20	49.1	20	65.0	10
4	230.1	50	111.8	20	110.7	20	115.6	20

Table 6: GraphQ's performance for answering Path queries over twitter-2010.

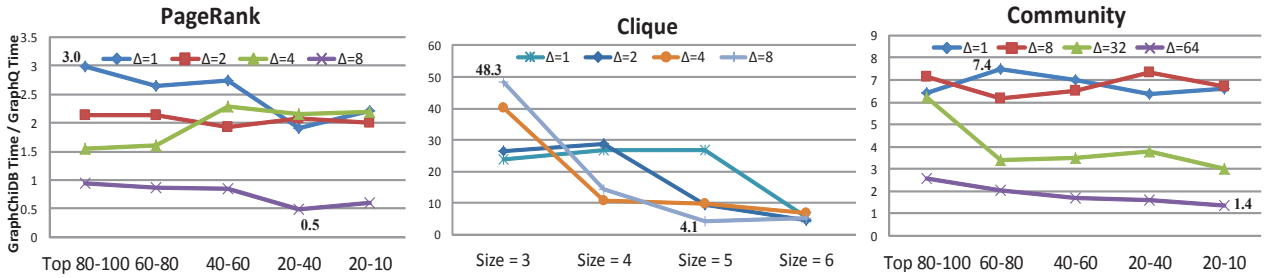
Our results for Path clearly demonstrate the benefit of GraphQ: it took the whole-graph computation 6.2 hours to process a graph only 1/5 as big as twitter-2010, while GraphQ can quickly find many paths of reasonable length in the original twitter graph.

Triangle A similar experiment was performed for Triangle (as shown in Figure 7): we focused on the top 100 vertices with the largest numbers of edge triangles. GraphQ could find only two vertices when a value from the top 10 triangle count list was used as a query goal. However, if the goal is chosen from the top 10-20 interval, GraphQ can easily find 16 vertices (which obviously include some top 10 vertices). It is worth noting that GraphQ found these vertices by combining only 10 partitions. This is easy to understand—edge triangles are local to vertices; computing them does not need to propagate any value on the graph. Hence, vertices with large triangle counts can be easily found as long as (most of) their own edges are recovered.

Δ	(a) Top10-20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	Par	*Time*	*Par*
1	3.3	10	3.0	10	2.9	10	4.5	10
2	3.2	10	3.6	10	3.9	10	7.6	10
4	3.4	10	3.2	10	3.1	10	8.7	10
8	2.8	10	3.3	10	3.0	10	19.6	10
16	2.9	10	2.9	10	3.2	10	313.3	10

Table 7: GraphQ's performance for answering Triangle queries over uk-2005.

The measurements in Table 3–7 also demonstrate the impact of the budget. For twitter-2010, merging 50, 30, 20, and 10 partitions requires, roughly, 6GB, 3.6GB, 2.4GB, and 1.2GB of memory, while, for uk-2005, the amounts of memory needed to merge 30, 20, and 10 partitions are 5.5GB, 4GB, and 2GB, respectively. From these measurements, it is easy to see what queries can and cannot be answered given a memory budget.



(a) PageRank (b) Clique (c) Community

Figure 3: Ratios between the running times of GraphChi-ET and GraphQ over twitter-2010: (a) PageRank: Max = 3.0, Min = 0.5, GeoMean = 1.6; (b) Clique: Max = 48.3, Min = 4.1, GeoMean = 13.4; and (c) Community: Max = 7.5, Min = 1.4, GeoMean = 4.2.

System	Time(s)	Comp.	Comp.Perc.	I/O	IO.Perc.
Q:PR	520.0	147.6	28.4%	372.4	71.6%
ET:PR	301.0	69.0	22.9%	232.0	77.1%
Q:Clique	637.0	548.5	86.1%	88.5	13.9%
ET:Clique	3208.0	2857.1	89.1%	351.0	10.9%
Q:Comm	81.5	25.6	31.4%	55.9	68.6%
ET:Comm	112.0	45.0	40.2%	68.0	60.7%

Table 8: A breakdown of time on computation and I/O for GraphQ and GraphChi-ET for PageRank, Clique, and Comm; measurements were obtained by running the most difficult queries from Figure 3.

6.2 Comparison to GraphChi-ET

GraphChi-ET is a modified version of GraphChi in which we developed a simple interface that allows the user to specify the Δ and goal for a query and then run GraphChi's whole-graph computation to answer the query – the computation is terminated immediately when the query is answered. Figures 3 shows performance comparisons between GraphQ and GraphChi-ET over twitter-2010 on three algorithms using SSD. A similar trend can also be observed on the other two algorithms; their results are omitted due to space limitations.

Note that for PageRank, GraphQ outperforms GraphChi-ET in all cases except when $\Delta = 8$. In this case, GraphQ is about 2× slower than GraphChi-ET because GraphQ needs to merge 50 partitions and is always close to running out of budget. The memory pressure is constantly high, making in-memory computation less efficient than GraphChi-ET's PSW algorithm. For all the other benchmarks, GraphQ runs much faster than GraphChi-ET. An extreme case is when $\Delta = 1$ for Clique, as shown in Figure 3 (b), GraphChi-ET found a 3-clique in 159.5 seconds, while GraphQ successfully answered the query only in 3.3 seconds. This improvement stems primarily from GraphQ's ability of prioritizing partitions and intelligently enlarging the processing scope.

Table 8 shows a detailed breakdown of running time on I/O and computation for answering the most difficult queries from Figure 3 (i.e., those represented by points at the bottom right corner of each plot). These queries have the longest running time, which enables an easier comparison. Clearly, GraphQ reduces both computation and I/O because it loads and processes fewer partitions. However, the percentages of I/O and computation in the total time of each query are roughly the same for GraphQ and GraphChi-ET.

6.3 Impact of Abstraction Refinement

To understand the impact of abstraction refinement, we varied the abstraction granularity by using 0.5GB, 1GB, and 2GB of the heap to store the AG. The numbers of abstract vertices for each partition corresponding to these sizes are $a = 25$, 50, and 100, respectively, for twitter-2010. We fixed the budget at 50 partitions (which consume 6GB memory), so that we could focus on how performance changes with the abstraction granularity. We have also tested GraphQ without abstraction refinement: partitions are randomly selected to merge. These experiments were conducted for all the algorithms; due to space limitations, we show our results only for PageRank.

Figure 4 compares performance under different abstraction granularity for $\Delta = 1$, 4, and 8. While configuration $a = 100$ always yields the best performance, its running time is very close to that of $a = 50$. It is interesting to see that, in many cases (especially when $\Delta = 4$), $a = 25$ yields worse performance than random selection. We carefully inspected this AG and found that the abstraction level is so high that different abstract vertices have similar degrees. The X-percentages for different partitions computed based on the AG are also very similar, and hence, partitions are merged almost in a sequential manner (*e.g.*, partitions 1–10 are first merged, followed by 10–20, *etc.*). In this case, the random selection has a higher probability of finding the appropriate partitions to merge.

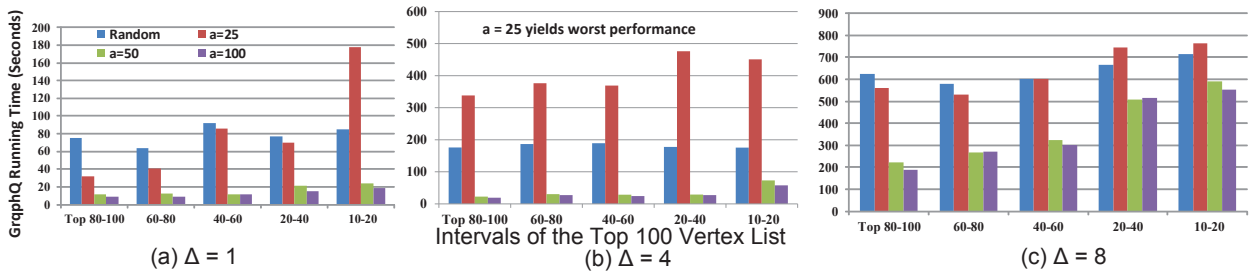


Figure 4: GraphQ's running time (in seconds) for answering PageRank queries over twitter-2010 using different abstraction graphs: *Random* means no refinement is used and partitions are merged randomly; $a = i$ means a partition is represented by i abstract vertices.

Despite its slow running time, random selection found all vertices requested by the queries. This is because, in the twitter graph, the edges of high-degree vertices are reasonably evenly distributed in different partitions of the graph. A similar observation was made for Triangle. But for the other three algorithms, their dependence on the AG is much stronger. For example, GraphQ could not answer any path query without the AG. As another example, no cliques larger than 3 could be found by using random selection.

7 Related Work

Graph Analytics A large body of work [16, 24, 19, 18, 5, 20, 26, 12] exists on efficient processing of large graphs. Existing work focuses on developing either new programming models [5, 18, 20] or algorithms that can reduce systems cost of graph processing such as communication overhead, random disk accesses, or GC effort. GraphChi [16], X-Stream [23], and GridGraph [29] are systems that perform out-of-core graph processing on a single machine. GraphQ differs from all these systems in its way to analyze partial graphs.

Work from [22] proposes *Galois*, a lightweight infrastructure that uses a rich programming model with coordinated and autonomous scheduling to support more efficient whole-graph computation. Unlike the existing systems that compute whole-graph solutions, GraphQ employs abstraction refinement to answer various kinds of analytical queries, facilitating applications that only concern small portions of the graph. There also exists work on graph databases such as Neo4j [1] and GraphChiDB [17]. They focus primiarily on enabling quick lookups on edge and vertex properties, while GraphQ focuses on quickly answering analytical queries.

Approximate Queries There is a vast body of work [13, 14, 7] on providing approximate answers to relational queries. These techniques use synopses like samples [13], histograms [14], and wavelets [7] to efficiently answer database queries. However, they have limited applicability to graph queries. Graph compression/-clustering/summarization [21, 28, 10, 25, 11] has been extensively studied in the database community. These techniques focus on (lossy and lossless) algorithms to summarize the input graph so that graph queries can be answered efficiently on the summary graph. Unlike the graph compression techniques that trade off graph accuracy for efficiency, GraphQ never answers queries over a summary graph, but instead, it only uses the summary graph to rule out infeasible solutions and always resorts to the concrete graph to find a solution. In addition, the graphs used to evaluate the aforementioned systems are relatively small—they only have a few hundreds of vertices and edges, which can be easily loaded into memory. In comparison, the graphs GraphQ analyzes are at the scale of several billions of edges and cannot be entirely loaded into memory.

8 Conclusion

This paper presents GraphQ, a graph query system based on abstraction refinement. GraphQ divides a graph into partitions and merges them with the guidance from a flexible programming model. An abstraction graph is used to quickly rule out infeasible solutions and identify mergeable partitions. GraphQ uses the memory capacity as a budget and tries its best to find solutions before exhausting the memory. GraphQ is the *first* graph processing system that can answer analytical queries over partial graphs, opening up new possibilities to scale up Big Graph processing with small amounts of resources.

Acknowledgements

We would like to thank our shepherd Emil Sit and the anonymous reviewers for their valuable and thorough comments. We also thank Rajiv Gupta, Xiaoning Ding, and Feng Qin for their helpful comments on an early draft of the paper. This material is based upon work supported by the National Science Foundation under the grants CCF-1054515, CCF-1117603, CNS-1321179, CNS-1319187, CCF-1349528, and CCF-1409829, and by the Office of Naval Research under grant N00014-14-1-0549.

References

[1] The neo4j graph database. http://neo4j.com/, 2014.

[2] The Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/, 2014.

[3] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[4] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[5] Y. Bu, V. Borkar, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 7, 2015.

[6] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, pages 169–180, 2012.

[7] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3):199–223, 2001.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[10] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[11] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *SIGMOD*, pages 301–312, 2014.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[14] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

[15] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[16] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[17] A. Kyrola and C. Guestrin. GraphChi-DB: Simple design for a scalable graph database system – on just a PC. http://arxiv.org/pdf/1403.0701v1.pdf.

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[20] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[21] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[22] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[23] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[24] S. Salihoglu and J. Widom. GPS: A graph processing system. In *Scientific and Statistical Database Management*, July 2013.

[25] H. Toivonen, F. Zhou, A. Hartikainen, and A. Hinkka. Compression of weighted graphs. In *KDD*, pages 965–973, 2011.

[26] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *OOPSLA*, pages 861–878, 2014.

[27] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[28] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1):718–729, 2009.

[29] X. Zhu, W. Han, and W. Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, 2015.

[30] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.

Accurate Latency-based Congestion Feedback for Datacenters

Changhyun Lee Chunjong Park Keon Jang† Sue Moon Dongsu Han
KAIST †Intel Labs

Abstract

The nature of congestion feedback largely governs the behavior of congestion control. In datacenter networks, where RTTs are in hundreds of microseconds, accurate feedback is crucial to achieve both high utilization and low queueing delay. Proposals for datacenter congestion control predominantly leverage ECN or even explicit in-network feedback (e.g., RCP-type feedback) to minimize the queuing delay. In this work we explore latency-based feedback as an alternative and show its advantages over ECN. Against the common belief that such implicit feedback is noisy and inaccurate, we demonstrate that latency-based implicit feedback is accurate enough to signal a single packet's queuing delay in 10 Gbps networks.

DX enables accurate queuing delay measurements whose error falls within 1.98 and 0.53 microseconds using software-based and hardware-based latency measurements, respectively. This enables us to design a new congestion control algorithm that performs fine-grained control to adjust the congestion window just enough to achieve very low queuing delay while attaining full utilization. Our extensive evaluation shows that 1) the latency measurement accurately reflects the one-way queuing delay in single packet level; 2) the latency feedback can be used to perform practical and fine-grained congestion control in high-speed datacenter networks; and 3) DX outperforms DCTCP with 5.33x smaller median queueing delay at 1 Gbps and 1.57x at 10 Gbps.

1 Introduction

The quality of network congestion control fundamentally depends on the accuracy and granularity of congestion feedback. For the most part, the history of congestion control has largely been about identifying the "right" form of congestion feedback. From packet loss and explicit congestion notification (ECN) to explicit in-network feedback [1, 2], the pursuit for accurate and fine-grained feedback has been central tenet in designing new congestion control algorithms. Novel forms of congestion feedback have enabled innovative congestion control behaviors that formed the basis of a number of flexible and efficient congestion control algorithms [3, 4], as the requirements for congestion control diversified [5].

With the advent of datacenter networking, identifying and leveraging more accurate and fine-grained feedback mechanisms have become even more crucial [6]. Round trip times (RTTs), which represent the interval of the control loop, are few hundreds of microseconds, where as TCP is designed to work in the wide area network (WAN) with hundreds of milliseconds of RTTs. Prevalence of latency-sensitive flows in datacenters (e.g., Partition/Aggregate workloads) requires low latency while the end-to-end latency is dominated by in-network queuing delay [6]. As a result, proposals for datacenter congestion control predominantly leverage ECN (e.g., DCTCP [6] and HULL [7]) or explicit in-network feedback (e.g., RCP-type feedback [2]), to minimize the queuing delay and the flow completion times.

This paper takes a relatively unexplored path of identifying a better form of feedback for datacenter networks. In particular, this paper explores the prospect of using network latency as congestion feedback in the datacenter environment. We believe latency can be a good form of congestion feedback in datacenters for a number of reasons: (i) by definition, it includes all queuing delay throughout the network, and hence is a good indicator for congestion; (ii) a datacenter is typically owned by a single entity who can enforce all end hosts to use the same latency-based protocol, effectively removing potential source of errors originating from uncontrolled traffic; and (iii) finally, latency-based feedback does not require any switch support.

Although latency-based feedback has been previously explored in WAN [8, 9], the datacenter environment is very different, posing unique requirements that are difficult to address. Datacenters have much higher bandwidth (10 Gbps to even 40 Gbps) at the end host and very low latency (few hundreds of microseconds) in the network.

This makes it difficult to measure the queuing delay of individual packets for a number of reasons: (i) I/O batching at the end host, which is essential for high throughput, introduces large measurement error (§2). (ii) Measuring queuing delay requires high precision because a single MSS packet introduces only 0.3 (1.2) microseconds of queuing delay in 40*GbE* (10*GbE*) networks. As a result, the common belief is that latency measurement might be too noisy to serve as reliable congestion feedback [6, 10].

On the contrary, we argue that it is possible to accurately measure the queuing delay at the end-host, so that even a single packet queuing delay is detectable. Realizing this requires solving several design and implementation challenges. First, even with very accurate hardware measurement, bursty I/O (e.g., DMA bursts) leads to inaccurate delay measurements. Second, ACK packets on the reverse path may be queued behind data packets and add noise to the latency measurement. To address these issues, we leverage a combination of recent advances in software low latency packet processing [11, 12] and hardware technology [13] that allows us to measure queuing delay accurately.

Such accurate delay measurements enable a more fine-grained control loop for datacenter congestion control. In particular, we envision a fine-grained feedback control loop achieves near zero-queuing with high utilization. Translating latency into feedback control to achieve high-utilization and low queuing is non-trivial. We present DX, a latency based congestion control that addresses these challenges. DX performs window adaptation to achieve low queuing delay (as low as that of HULL [7] and 6.6 times smaller than DCTCP), while achieving 99.9% utilization. Moreover it provides advantages over recent works in that it does not require any switch modifications.

To summarize, our contributions in this paper are the followings: (i) novel techniques to accurately measure in-network queuing delay based on end-to-end latency measurements; (ii) a congestion control logic that exploits latency-based feedback to achieve just a few packets of queuing delay and high utilization without any form of in-network support; and (iii) a prototype that demonstrates the feasibility and its benefits in our testbed.

2 Accurate queuing delay measurement

Latency measurement can be inaccurate for many reasons including variability in end-host stack latency, NIC queuing delay, and I/O batching. In this section, we describe several techniques to eliminate such sources of errors. Our goal is to achieve a level of accuracy that can distinguish even a single MSS packet queuing at 10 Gbps, which is 1.2 μs. This is necessary to target near zero queuing as congestion control should be able to back off even when a single packet is queued.

Before we introduce our solutions to each source of error, we first show how noisy the latency measurement is without any care. Figure 1 shows the round trip time measured by the sender's kernel when saturating a 10 Gbps link; we generate TCP traffic using *iperf* [14] on Linux kernel. the sender and the receiver are connected back to back, so no queueing is expected in the network. Our measurement shows that the round-trip time varies from 23 μs to 733 μs, which potentially gives up to 591 packets of error. The middle 50% of RTT samples still exhibit wide range of errors of 111 μs that corresponds to 93 packets. These errors are an order of magnitude larger than our target latency error, 1.2 μs.

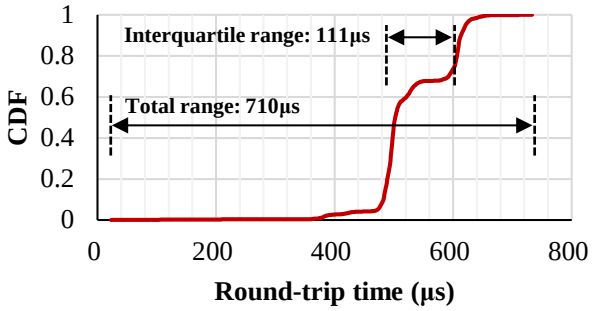


Figure 1: Round-trip time measured in kernel

Source of error	Elimination technique
End-host network stack ($\sim 100\mu s$)	Exclude host stack delay
I/O batching & DMA bursts (tens of μs)	Burst reduction & error calibration
Reverse path queuing ($\sim 100\mu s$)	Use difference in one-way latency
Clock drift (long term effect)	Frequent base delay update

Table 1: Sources of errors in latency measurement and our techniques for mitigation.

Table 1 shows four sources of measurement errors and their magnitude. We eliminate each of them to achieve our target accuracy ($\sim$1μsec).

Removing host stack delay: End-host network stack latency variation is over an order of magnitude larger than our target accuracy. Our measurement shows about 80μs standard deviation, when the RTT is measured in the Linux kernel's TCP stack. Thus, it is crucial to eliminate the host processing delay in both a sender and a receiver.

For software timestamping, our implementation choice eliminates the end host stack delay at the sender as we timestamp packets right before the TX, and right after

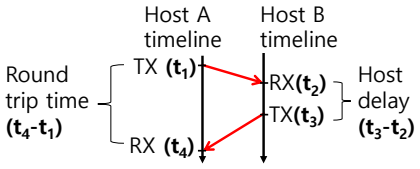


Figure 2: Timeline of timestamp measurement points

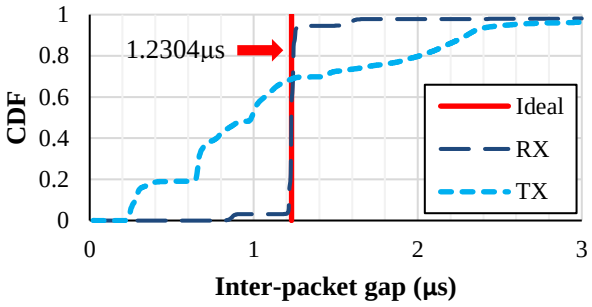


Figure 3: H/W timestamped inter-packet gap at 10 Gbps

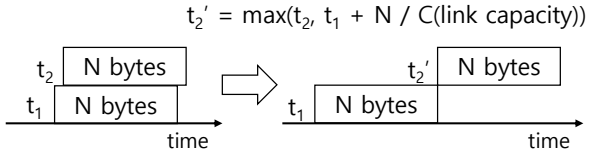


Figure 4: Example delay calibration for bursty packet reception

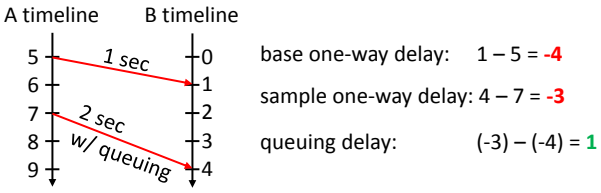


Figure 5: One-way queuing delay without time synchronization

the RX on top of DPDK [12]. Hardware timestamping innately removes such delay.

Now, we need to deal with the end-host stack delay at the receiver. Figure 2 shows how DX timestamps packets when a host sends one data packet and receives back an ACK packet. To remove the end host stack delay from the receiver, we simply subtract the $t_3 - t_2$ from $t_4 - t_1$. The timestamp values are stored and delivered in the option fields of the TCP header.

Burst reduction: TCP stack is known to transmit packets in a burst. The amount of burst is affected by the window size and TCP Segmentation Offloading (TSO), and ranges up to 64 KB. Burst packets affect timestamping because all packets in a TX burst get the almost the same timestamp, and yet they are received by one by one at the receiver. This results in an error as large as $50\mu s$.

To eliminate packet bursts, we use a software token bucket to pace the traffic at the link capacity. The token bucket is a packet queue and drained by polling in SoftNIC [15].

At each poll, the number of packets drained is calculated based on the link rate and the elapsed time from the last poll. The upper bound is 10 packets, which is enough to saturate 99.99% of the link capacity even in 10 Gbps networks. We note that our token bucket is different from TCP pacing or the pacer in HULL [7] where each and every packet is paced at the target rate; our token bucket is simply implemented with very small overhead. In addition, we keep a separate queue for each flow to prevent the latency increase from other flows' queue build-ups.

Error calibration: Even after the burst reduction, packets can be still batched for TX as well as RX. Interestingly, we find that even hardware timestamping is subject to the noise introduced by packet bursts due to its implementation. We run a simple experiment where sending a traffic near line rate 9.5 Gbps from a sender to a receiver connected back to back. We measure the inter packet gap using hardware timestamps, and plot the results in Figure 3. Ideally, all packets should be spaced at $1.23\mu s$. As shown in the figure, a large portion of the packet gaps of TX and RX falls below $1.23\mu s$. The packet gaps of TX are more variable than that of RX, as it is directly affected by I/O batching, while RX DMA is triggered when a packet is received by the NIC. The noise in the H/W is caused by the fact that the NIC timestamps packets when it completes the DMA, rather than timestamping them when the packets are sent or received on the wire. We believe this is not a fundamental problem, and H/W timestamping accuracy can be further improved by minor changes in implementation.

In this paper, we employ simple heuristics to reduce the noise by accounting for burst transmission in software. Suppose two packets are received or transmitted in the same batch as in Figure 4. If the packets are spaced with timestamps whose interval is smaller than what the link capacity allows, we correct the timestamp of the latter packet to be at least transmission delay away from the former packet's timestamp. In our measurement at $10Gbps$, 68% of the TX timestamp gaps need such calibration.

One-way queuing delay: So far, we have described techniques to accurately measure RTT. However, RTT includes the delay on the reverse path, which is another source of noise for determining queuing on the forward path. A simple solution to this is measuring one-way delay which requires clock synchronization between two hosts. PTP (Precision Time Protocol) enables clock synchronization with sub-microseconds [16]. However it requires hardware support and possibly switch support to

remove errors from queuing delay. It also requires periodic synchronization to compensate clock drifts. Since we are targeting a microsecond level of accuracy, even a short term drift could affect the queuing delay measurement. For these reasons, we choose not to rely on clock synchronization.

Our intuition is that unlike one-way delay, queuing delay can be measured simply by subtracting the baseline delay (skewed one-way delay with zero queuing) from the sample one-way delay even if the clocks are not synchronized. For example, suppose a clock difference of 5 seconds, as depicted in Figure 5. When we measure one-way delay from A to B, which takes one second propagation delay (no queuing), the one-way delay measured would be -4 seconds instead of one second. When we measure another sample where it takes 2 seconds due to queuing delay, it would result in -3 seconds. By subtracting -4 from -3, we get one second queuing delay.

Now, there are two remaining issues. First is obtaining accurate baseline delay, and second is dealing with clock drifts. The base line can be obtained by picking the minimum one-way delay amongst many samples. The frequency of zero queuing being measured depends on the congestion control algorithm behavior. Since we target near zero-queuing, we observe this every few RTTs.

Handling clock drift: A standard clock drifts only 40 nsecs per msec [17]. This means that the relative error between two measurements (e.g., base one-way delay and sample one-way delay) taken from two clocks during a millisecond can only contain tens of nanoseconds of error. Thus, we make sure that base one-way delay is updated frequently (every few round trip times). One last caveat with updating base one-way delay is that clock drift differences can cause one-way delay measurements to continuously increase or decrease. If we simply take minimum base one-way delay, it causes one side to update its base one-way delay continuously, while the other side never updates the base delay because its measurement continuously increases. As a workaround, we update the base one-way delay when the RTT measurement hits the new minimum or re-observe the current minimum; RTT measurements are not affected by clock drift, and minimum RTT implies no queueing in the network. This event happens frequently enough in DX, and it ensures that clock drifts do not cause problems.

3 DX: Latency-based Congestion Control

The ability to accurately measure the switch queue length from end-hosts enables new opportunities. In particular, DX leverages its power for finer-grained congestion control.

We present a congestion control algorithm for datacenters that targets near zero queueing delay based on implicit feedback, without any form of in-network support. Because latency feedback signals the amount of excessive packets in the network, it allows senders to calculate the maximum number of packets to drain from the network while achieving full utilization. This section presents the basic mechanisms and design of our new congestion control algorithm, DX. Our target deployment environment is datacenters, and we assume that all traffic congestion is controlled by DX, similar to the previous work [3, 5–7, 10].

DX is a window-based congestion control algorithm. DX's congestion avoidance follows the popular Additive Increase Multiplicative Decrease (AIMD) rule. The key difference from TCP (e.g., TCP Reno) is its congestion avoidance algorithm. DX uses the queueing delay to make a decision on whether to increase or decrease congestion window in the next round at every RTT. Zero queueing delay indicates that there is still more room for packets in the network, so the window size is increased by one at a time. On the other hand, any positive queueing delay means that a sender must decrease the window.

DX updates the window size using the formula below:

$$new\ CWND = \begin{cases} CWND + 1, & \text{if } Q = 0 \\ CWND \times (1 - \frac{Q}{V}), & \text{if } Q > 0, \end{cases} \quad (1)$$

where Q represents the latency feedback, that is, the average queueing delay in the current window, and V is a self-updated coefficient of which role is critical in our congestion control.

When $Q > 0$, DX decreases the window proportional to the current queueing delay. The amount to decrease should be just enough to drain the currently queued packets not to affect utilization. An aggressive decrease in the congestion window will cause the network utilization to drop below 100%. For DX, the exact amount depends on the number of flows sharing the bottleneck because the aggregate sending rate of these flows should decrease to drain the queue. V is the coefficient that accounts for the number of competing flows. We drive the value of V using the analysis below.

We denote the link capacity (packets / sec) as C, the base RTT as R, single-packet transmission delay as D, the number of flows as N, and the window size and the queueing delay of flow k at time t as $W_k{}^{(t)}$ and $Q_k{}^{(t)}$, respectively. Without loss of generality, we assume at time t the bottleneck link fully utilized and the queue size is zero. We also assume that their behaviors are synchronized to derive a closed-form analysis and verify the results using simulations and testbed experiments. At time t, because the link is fully utilized and the queuing delay is zero, the sum of the window size equals to the bandwidth delay product $C \cdot R$:

$$\sum_{k=1}^{N} {W_k}^{(t)} = C \cdot R \tag{2}$$

Since none of the N flows experiences congestion, they all increase their window size by one at time $t+1$:

$$\sum_{k=1}^{N} {W_k}^{(t+1)} = C \cdot R + N \tag{3}$$

Now all the senders observe a positive queueing delay, and they respond by decreasing the window size using the multiplicative factor, $1 - Q/V$, as in (1). As a result, at time $t+2$, we expect fewer packets in the network; we want just enough packets to fully saturate the link and achieve zero queuing delay in the next round. We calculate the total number of packets in the network (in both the link and the queues) at time $t+2$ from the sum of window size of all the flows.

$$\sum_{k=1}^{N} {W_k}^{(t+2)} = \sum_{k=1}^{N} {W_k}^{(t+1)} (1 - \frac{{Q_k}^{(t+1)}}{V}) \tag{4}$$

Assuming every flow experiences maximum queueing delay $N \cdot D$ in the worst case, we get:

$$\begin{aligned} \sum_{k=1}^{N} {W_k}^{(t+2)} &= \sum_{k=1}^{N} {W_k}^{(t+1)} (1 - \frac{N \cdot D}{V}) \\ &= (C \cdot R + N)(1 - \frac{N \cdot D}{V}) \end{aligned} \tag{5}$$

We want total number of in-flight packets at time $t+2$ to equal to the bandwidth delay product:

$$(C \cdot R + N)(1 - \frac{N \cdot D}{V}) = C \cdot R \tag{6}$$

Solving for V results in:

$$V = \frac{N \cdot D}{(1 - \frac{C \cdot R}{C \cdot R + N})} \tag{7}$$

Among the variables required to calculate V, the only unknown is N, which is the number of concurrent flows. The number of flows can be estimated from the sender's own window size because DX achieves fair-share throughput at steady state. For notational convenience, we denote ${W_k}^{(t+1)}$ as W^* and rewrite (3) as:

$$\sum_{k=1}^{N} {W_k}^{(t+1)} = N \times W^* = C \cdot R + N \Leftrightarrow N = \frac{C \cdot R}{W^* - 1}$$

Using (5) and replacing D, single-packet transmission delay, with $(1/C)$, we get:

$$V = \frac{R \cdot W^*}{W^* - 1} \tag{8}$$

In calculating V, the sender only needs to know the based RTT, R, and the previous window size W^*. No additional measurement is required. We do not need to rely on external configuration or parameter settings either, unlike the ECN-based approaches. Even if the link capacity in the network varies across links, it does not affect our calculation of V.

4 Implementation

We have implemented DX in two parts: latency measurement in DPDK-based NIC driver and latency-based congestion control in the Linux's TCP stack. This separation provides a few advantages: (i) it measures latency more accurately than doing so in the Linux Kernel; (ii) legacy applications can take advantage of DX without modification; and (iii) it separates the latency measurement from the TCP stack, and hides the differences between hardware implementations, such as timestamp clock frequencies or timestamping mechanisms. We present the implementation of software- and hardware-based latency measurements and modifications to the kernel TCP stack to support latency feedback.

4.1 Timestamping and delay calculation

We measure four timestamp values as shown in section 2 Figure 2: t_1 and t_2 are the transmission and reception time of a data packet, and t_3 and t_4 are the transmission and reception time of a corresponding ACK packet.

Software timestamping: To eliminate host processing delay, we perform TX timestamping right before pushing packets to the NIC, and RX timestamping right after the packets are received, at the DPDK-based device driver. We use `rdtsc` to get CPU cycles and transform this into nanoseconds timescales. We correct timestamps using techniques described in §2. All four timestamps must be delivered to the sender to calculate the one-way delay and the base RTT. We use TCP's option fields to relay t_1, t_2, and t_3 (§4.2).

To calculate one-way delay, the DX receiver stores a mapping from expected ACK number to t_1 and t_2 when it receives a data packet. It then puts them in the corresponding ACK along with the ACK's transmission time (t_3). The memory overhead is proportional to the arrived data of which the corresponding ACK has not been sent yet. The memory overhead is negligible as it requires store 8 bytes per in-flight packet. In the presence of delayed ACK, not all timestamps are delivered back to the sender, and some of them are discarded.

Hardware timestamping: We have implemented hardware-based timestamping on Mellanox ConnectX-3 using a DPDK-ported driver. Although the hardware supports RX/TX timestamping for all packets, its driver did not support TX timestaming. We have modified the driver to timestamp all RX/TX packets.

The NIC hardware delivers timestamps to the driver by putting the timestamps in the ring descriptor when it completes DMA. This causes an issue with the previous logic to carry t_1 in the original data packet. To resolve this, we store mapping of expected ACK number to the t_1 at the sender, and retrieve this when ACK is received.

LRO handling: Large Receive Offload (LRO) is a widely used technique for reducing CPU overhead on the receiver side. It aggregates received TCP data packets into a large single TCP packet and passes to the kernel. It is crucial to achieve 10 Gbps or beyond in today's Linux TCP stack. This affects DX in two ways. First, it makes the TCP receiver generate fewer number of ACKs, which in turn reduces the number of t_3 and t_4 samples. Second, even though t_1 and t_2 are acquired before LRO bundling at the driver, we cannot deliver all of them back to the kernel TCP stack due to limited space in the TCP option header. To work around the problem, for each ACK that is processed, we scan through the previous t_1 and t_2 samples, and deliver average one-way delay with the sample count. In fact, instead of passing all timestamps to the TCP layer, we only passes one-way delay t_2 - t_1 and RTT $((t_4 - t_1) - (t_3 - t_2))$

Burst mitigation: As shown in § 2, burstiness from I/O batching incurs timestamping errors. To control burstiness, we implement a simple token bucket with burst size of MTU and rate set to link capacity. SoftNIC [15] does polling on the token bucket to draw packets and passes them to the timestamping module or the NIC. If the polling loop takes longer than the transmission time of a packet, the token bucket emits more than one packet, but limits the number of packets to keep up with link capacity.

4.2 Congestion control

We implement DX congestion control algorithm in the Linux 3.13.11 kernel. We add DX as a new TCP option that consumes 14 bytes of additional TCP header. The first 2 bytes are for the option number and the option length required by the TCP option parser. The remaining 12 bytes are divided into three 4 byte spaces and used for storing timestamps and/or an ACK number.

Most of modifications are made in the *tcp_ack()* function in TCP stack. This is triggered when an ACK packet is received. An ACK packet carries one-way delay and RTT in the header that are pre-calculated by the DPDK-based device driver. For each round trip time, the received delay samples are averaged and used for new CWND calculation. The current implementation takes the average one-way delay observed during the last round trip.

Practical considerations: In real-world networks, a transient increase in queueing delay Q does not always mean network congestion. Reacting to wrong congestion signals results in low link utilization. There are two sources of error: measurement noise and instant queueing due to packet bursts. Although we have shown that our latency measurement has a low standard deviation up to about a microsecond, it can still trigger undesirable window reduction as DX reacts to a positive queueing delay whether large or small. On the other hand, instant queueing can happen with even very small number of packets. For example, if two packets arrive at the switch at the exactly same moment, one of them will be served after the first packet's transmission delay, hence positive queueing delay.

To tackle such practical issues, we come up with two simple techniques. First, we use headroom when determining congestion; DX does not decrease window size when $Q < headroom$.

Second, to be robust against transient increase in delay measurements, we use the average queueing delay during an RTT period. In an ideal network without packet bursts, the maximum queueing delay is a good indication of excess packets. In real networks, however, taking the maximum is easily affected by instant queueing. Taking the minimum removes the burstiness most effectively, but it detects congestion only when all the packets in the window experience positive queueing delay. Hence we choose the average to balance them out.

Note that DCTCP, a previous ECN-based solution, also suffers from bursty instant queueing and requires higher ECN threshold in practice than theoretic calculation [6].

5 Evaluation

Throughout the evaluation, We answer three main questions:

- Can DX obtain the accuracy of a single packet's queuing delay in high-speed networks?
- Can DX achieve minimal queuing delay while achieving high utilization?
- How does DX perform in large scale networks with realistic workloads?

By using testbed experiments, we show that our noise reduction techniques are effective and queuing delay can be measured with an accuracy of a single MSS packet at 10 Gbps. We evaluate DX against DCTCP and verify that it reduces queuing in the switch up to five times.

Next, we use *ns-2* packet level simulation to conduct more detailed analysis and evaluate DX in large-scale with realistic workload. First, we verify the DX's effectiveness by looking at queuing delay, utilization and fairness. We then quantify the impact of measurement errors on DX to evaluate its robustness. Finally, we perform large-scale evaluation to compare DX's overall performance against the state of the art: DCTCP [6] and HULL [7].

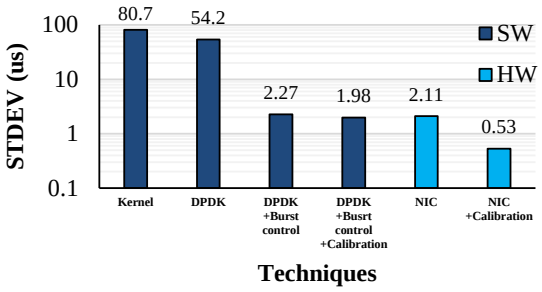
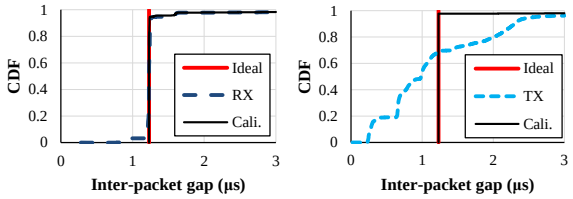


Figure 6: Improvements with noise reduction techniques



(a) RX Calibration (b) TX Calibration

Figure 7: Effect of calibration in H/W timestamped inter-packet gap at 10 Gbps

5.1 Accuracy of queuing delay in testbed

For testbed experiments, we use Intel 1 GbE/10 GbE NICs for software timestamping and Mellanox ConnectX-3 40 GbE NIC for hardware timestamping; the Mellanox NIC is used in 10 Gbps mode due to the lack of 40 GbE switches.

Effectiveness of noise reduction techniques: To quantify the benefit of each technique, we apply the techniques one by one and measure RTT using both software and hardware. Two machines are connected back to back, and we conduct RTT measurement at 10 Gbps link. We plot the standard deviation in Figure 6. Ideally, the RTT should remain unchanged since there is no network queueing delay. In software-based solution, we reduce the measurement error (presented as standard deviation) down to 1.98 μs by timestamping at DPDK and applying burst control and calibration. Among the techniques, burst control is the most effective, cutting down the error by 23.8 times. In hardware solution, simply timestamping at NIC achieves comparable noise with all techniques applied in the software solution. After inter-packet interval calibration, the noise drops further down to 0.53 μs, less than half of a single packet's queueing delay at 10 Gbps, which is within our target accuracy.

Calibration of H/W timestamping: We look further into how calibration affects the accuracy of hardware timestamping. Figure 7 shows the CDF of inter packet gap measurements before and after calibration for both RX and TX. The calibration effectively removes the inter packet gap samples smaller than link transmission delay

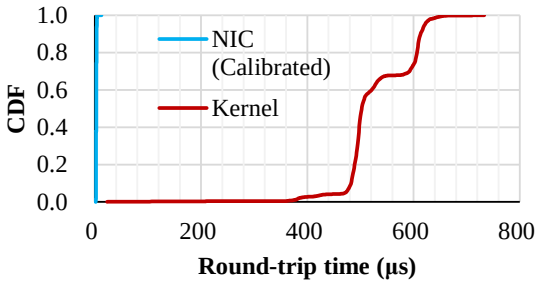
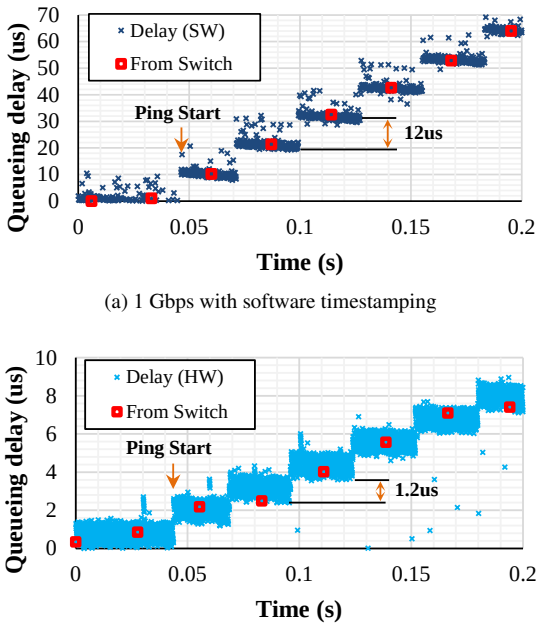


Figure 8: Improvement on RTT measurement error compared to kernel's



(a) 1 Gbps with software timestamping

(b) 10 Gbps with hardware timestamping

Figure 9: Accuracy of queuing delay measurement

which originally took up 68% for TX and 32% for RX.

Overall RTT measurement accuracy improvement: Now, we look at how much overall improvements we made on the accuracy of RTT measurement. We plot the CDF of RTT measurement for our technique using hardware and RTT measured in the Kernel in Figure 8. The total range of RTT has decreased by 62 times, from 710 μs to 11.38 μs. The standard deviation is improved from 80.7 μs to 0.53 μs by two orders of magnitude, and falls below a single packet queuing at 10 Gbps.

Verification of queuing delay: Now that we can measure RTT accurately, the remaining question is whether it leads to accurate queuing delay estimation. We conduct a controlled experiment where we have a full control over the queuing level. To create such scenario, we saturate a port in a switch by generating full throttle traffic from one host, and inject a MTU-sized ICMP packet to the same port at fixed interval from another host. This way, we

increase the queuing by a packet at fixed interval, and we measure the queuing statistics from the switch to verify our queuing delay measurement.

Figure 9 shows the time series of queuing delay measured by DX along with the ground truth queue occupancy measured at the switch (marked as red squares). We use software and hardware timestamping for 1 Gbps and 10 Gbps, respectively. Every time a new ping packet enters the network, the queueing delay increases by one MTU packet transmission delay: 12 μs at 1 Gbps and 1.2 μs at 10 Gbps. The queue length retrieved from the switch also matches our measurement result. The result at 10 Gbps seems noisier than at 1 Gbps due to the smaller transmission delay; note that the scale of Y-axis is different in two graphs.

Overall, we observe that our noise reduction techniques can effectively eliminate the sources of errors and result in accurate queuing delay measurement.

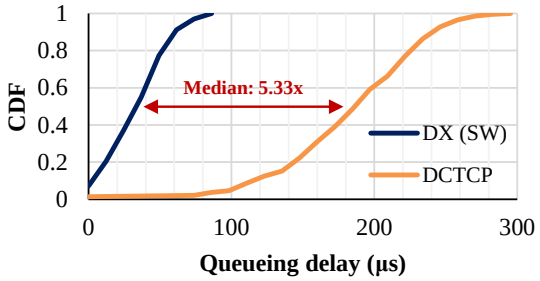
5.2 DX congestion control in testbed

Using the accurate queueing delay measurements, we run our DX prototype with three servers in our testbed; two nodes are senders and the other is a receiver. We use *iperf* [14] to generate TCP flows for 15 seconds. For comparison, we run DCTCP in the same topology. The ECN marking threshold for DCTCP is set to the recommended value of 20 at 1 Gbps and 65 at 10 Gbps [6]. During the experiment, the switch queue length is measured every 20 ms by reading the register values from the switch chipset. We first present the result at 1 Gbps bottleneck link in Figure 10a. In both protocols, two senders saturate the bottleneck link with fair-share throughput. The queue length is measured in bytes and converted into time.

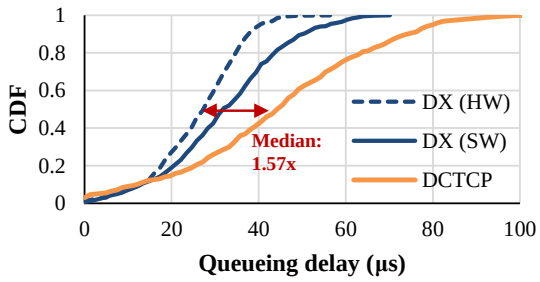
We observe that DX consistently reduces the switch queue length compared to that of DCTCP. The average queueing delay of DX, 37.8 μs, is 4.85 times smaller than that of DCTCP, 183.4 μs. DX shows 5.33x improvement in median queue length over DCTCP (3 packets for DX and 16 packets for DCTCP). DCTCP's maximum queue length goes up to 24 packets, while DX peaks at 8 packets.

We run the same experiment with 10 Gbps bottleneck. For 10 Gbps, we additionally run DX with hardware timestamp using Mellanox ConnectX-3 NIC. Figure 10b shows the result. DX (HW) denotes hardware timestamping, and DX (SW) denotes software timestamping. DX (HW) decreases the average queue length by 1.67 times compared to DCTCP, from 43.4 μs to 26.0 μs. DX (SW) achieves 31.8 μs of average queuing delay. The result also shows that DX effectively reduces the 99th-percentile queue length by a factor of 2 with hardware timestamping; DX (HW) and DX (SW) achieve 52 packets and 38 packets respectively while DCTCP achieves 78 packets.

To summarize, latency feedback is effective in maintaining low queue occupancy than ECN feedback, while



(a) 1Gbps bottleneck



(b) 10Gbps bottleneck

Figure 10: Queue length comparison of DX against DCTCP in Testbed

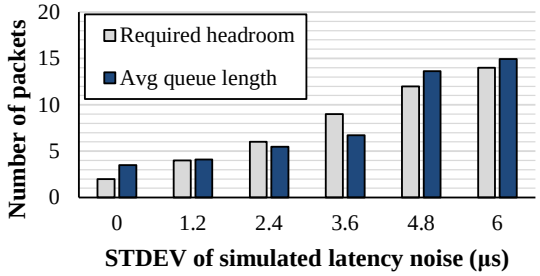


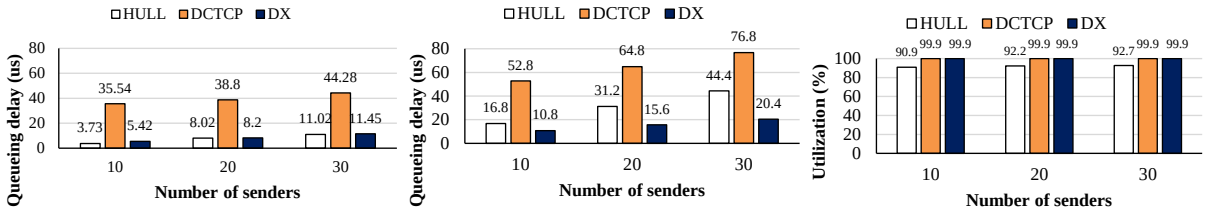
Figure 11: Impact of latency noise to headroom and queue length

saturating the link. DX achieves 4.85 times smaller average queue size at 1 Gbps and 1.67 times at 10 Gbps compared to DCTCP. DX reacts to congestion much earlier than DCTCP and reduces the congestion window to the right amount to minimize the queue length while achieving full utilization. DX achieves the lowest queueing delay among existing end-to-end congestion controls with implicit feedback that do not require any switch modifications,

In the next section, we also show that DX is even comparable to HULL, a solution that requires in-network support and switch modification.

5.3 Large-scale simulation

In this section, we run DX, DCTCP, and HULL in simulation to observe the performance in larger-scale environment.



(a) Queueing delay (average) (b) Queueing delay (99th percentile) (c) Utilization

Figure 12: Queueing delay and utilization of HULL, DCTCP, and DX

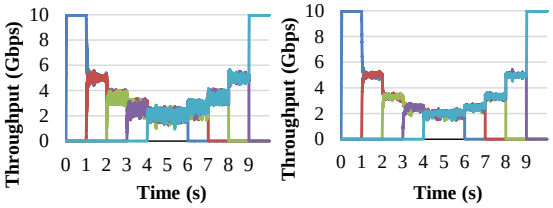
First, we run *ns-2* simulation using a dumbbell topology with 10 Gbps link capacity. Before the main simulation, we evaluate the impact of latency noise to the headroom size and average queue length. We generate latency noise using normal distribution with varying standard deviation. The noise level is multiples of 1.2 μs, single packet's transmission delay. As the simulated noise level increases, we need more headroom for full link utilization. Figure 11 shows the required headroom for full utilization and the resulting queue length in average. We observe that even if the noise becomes as large as 6 μs, DX can sustain noise error by simply increasing headroom size followed by the same amount of increase in queue length. Note that the standard deviation of our hardware timestamping is only 0.53 μs.

For scalability test, we now vary the number of simultaneous flows from 10 to 30 as queuing delay and utilization are correlated with it; the number of senders has a direct impact on queueing delay as shown in DCTCP [6]. We measure the queuing delay and utilization, and plot them in Figure 12.

Queueing delay: Many distributed applications with short flows are sensitive to the tail latency as the slowest flow that belongs to a task determines the completion time of the task [18]. Hence, we look at the 99th percentile queuing delay as well as the average queueing delay. On average, DX achieves 6.6x smaller queueing delay than DCTCP with ten senders, and slightly higher queuing delay than HULL. At 99th percentile, DX even outperforms HULL by 1.6x to 2.2x. The reason that DX achieves such low queuing is because of the immediate reaction to the queuing whereas both DCTCP and HULL uses weighted averaging for reducing congestion window size that takes multiple round trip times.

Utilization: DX achieves 99.9% of utilization which is comparable to DCTCP, but with much smaller queuing. HULL sacrifices utilization to reduce the queuing delay achieving about 90% of the bottleneck link capacity. We note that low queueing delay of DX does not sacrifice the utilization.

Fairness and throughput stability: To evaluate the throughput fairness, we generate 5 identical flows in the



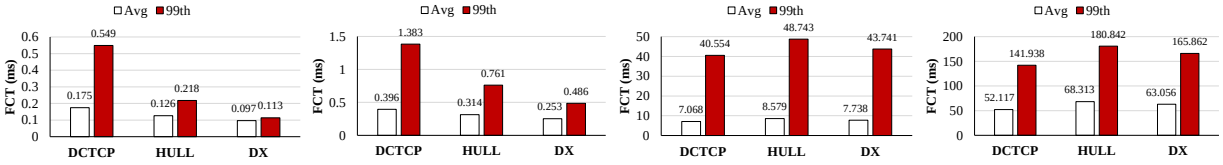
(a) DCTCP (b) DX

Figure 13: Fairness of five flows with DCTCP and DX

10 Gbps link one by one with 1 second interval and stop each flow after 5 seconds of transfer. In Figure 13, we see that both protocols offer fair throughput to exiting flows at each moment. One interesting observation is that DX flows have more stable throughput than DCTCP flows. This implies that DX provides higher fairness than DCTCP in small time scale. We compute the standard deviation of throughput to quantify the stability; 268 Mbps for DCTCP and 122 Mbps for DX.

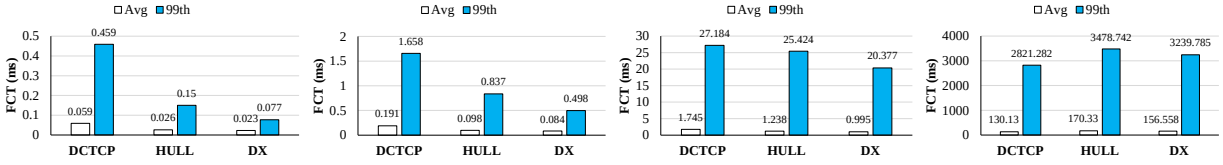
To understand the performance of DX in a large-scale data center environment, we perform simulations with realistic topology and traffic workload. The network consists of 192 servers and 56 switches that are connected as a 3-tier fat tree; there are 8 core switches, 16 aggregation switches, and 32 top-of-rack switches. All network links have 10 Gbps bandwidth, and the path selection is done by ECMP routing. The network topology we use is similar to that of HULL [7]. Once the simulation starts, the flow generator module selects a sender and a receiver randomly and starts a new flow. Each new flow is generated following Poisson process to produce 15% load at the edge. We run simulation until we have 100,000 flows started and finished. To test realistic workload, we choose flow size according to empirical workload reported from real-world data centers. We use two workload data: web search [6] and data mining [19].

Web search workload: The web search workload mostly contains small and medium-sized flows from a few KB to tens of MB; more than 95% of total bytes come from the flow smaller than 20MB, and the average flow size is 654KB [20]. In Figure 14, we present the flow comple-



(a) 0KB-10KB (b) 10KB-100KB (c) 100KB-10MB (d) 10MB-

Figure 14: Flow completion time of search workload



(a) 0KB-10KB (b) 10KB-100KB (c) 100KB-10MB (d) 10MB-

Figure 15: Flow completion time of data mining workload

tion time (FCT) in four flow-size groups: (0KB,10KB), [10KB,100KB), [100KB,10MB), and [10MB,∞).

For the flows smaller than 10KB, DX significantly reduces the 99th percentile FCT; it is 4.9x smaller than DCTCP and 1.9x smaller than HULL. DX also achieves minimal flow completion time in the 10KB-100KB group.

In larger flow size group, the performance of DX falls between DCTCP and HULL. DX achieves 7.7% lower average flow completion time compared to HULL and 20.9% higher than DCTCP for flows of size 10 MB and greater. This is because when ACK packets from other flows share the same bottleneck link, the queuing delay increases slightly. As a result, DX senders respond to the increased queuing delay. This is a side effect of targeting zero queueing. Because ACK packets are small and often piggy-backed on data packets we believe this is not a serious problem, but leave this as future work.

Data mining workload: The data mining workload is comprised of tiny and large-sized flows from hundreds of bytes to 1GB. The flow size is highly skewed that 80% of flows are smaller than 10KB [20] so 95% of bytes come from flows larger than 30MB; the average flow size is 7,452KB. The flow completion time of data mining workload is presented in Figure 15.

The performance improvement of DX is more outstanding for data minining workload than for search workload. In the three flow groups up to 10MB, DX flows finish early in every case. The biggest benefit comes from the smallest flow group as tail FCT is 6.0x smaller than DCTCP and 1.9x than HULL. For the largest flow group, DX suffers the same problem from the search workload but still shows shorter completion time than HULL's.

6 Discussion

NIC support for latency measurements: Current commodity NICs' support for timestamping is primarily for IEEE 1588 PTP, a hardware-based time synchronization protocol, designed to achieve sub-microsecond accuracy. While we leverage this functionality in DX, it is not perfectly suitable for our network latency measurements as explained in §2. In particular, it timestamps TX packets after completing DMA, and it does not support recording the TX time directly on the packets at the time of transmission. Although, our implementation works around these issues in software to reduce measurement errors, we believe changes in hardware will be more effective, especially for 10G/40G networks. If the hardware timestamps packets as it sends them out in the wire, the errors from NIC queueing and DMA bursts would be eliminated. Also, if it allows us to directly write timestamps on the packet header, this can shorten the feedback loop of DX by an RTT.

Deployment and co-existence with TCP: DX strictly targets datacenter networks for deployment. Datacenter environment favors DX deployment in that 1) it belongs to a single administration domain that can readily adopt a new protocol, and 2) network structure is more homogeneous and static than WAN, which helps latency measurement stability. As DX does not require any changes to the existing network switches, we can deploy DX with only end-host modification. Software-based solution can be deployed on existing machines, and hardware-based solution requires timestamping-enabled NICs. IEEE 1588 PTP-enabled NICs are already popular [21], and we envision timestamping-enabled NICs become more popular in the near future.

DX is specifically designed for handling only internal datacenter traffic, not external traffic to WAN. Separation between internal and external traffic is attainable by using load balancers and application proxies in existing datacenters [6]. We do not claim that DX can operate with conventional TCP sharing the same queue at network switches; a single TCP flow can cause a switch queue to overflow, which is directly against DX's goal. Our best resort to co-existing with TCP flows is to exploit priority queues at the switch and separate DX traffic from other TCP traffic. How to design such network efficiently is out of this paper's scope and we leave it as future work.

7 Related Work

Latency-based feedback in wide area network: There have been numerous proposals for network congestion control since the advent of the Internet. Although the majority of proposals use packet loss to detect network congestion, a large body of work has studied latency feedback. Latency-based TCP all agree on latency being more informative source of measuring congestion level, but the purpose and control mechanism is different in each protocol. TCP Vegas [8] is one of the earliest work and aims at achieving high throughput by avoiding loss. FAST TCP [9] is designed to quickly reach the fair-share throughput and uses latency for an equation parameter. TCP Nice [22] and TCP-LP [23] operate in low priority minimizing interference with other flows. So far, the latency-based approach has only been used in wide area network, and no protocol is known to target zero queueing delay.

ECN-based feedback in datacenter networks: Monitoring congestion level at the switch can help controlling the rate of TCP to minimize queuing. ECN marking in the TCP header has received much attention recently. DCTCP [6] uses a predefined threshold, and end-nodes then count the number of ECN marked packets to determine the degree of congestion and decrease the window size accordingly. HULL [7] is a similar to DCTCP, but sacrifices a small portion of the link capacity with phantom queue implemented at switches to detect congestion early and to achieve lower queueing delay than DCTCP. D^2TCP [24] also follows the same line of idea as DCTCP, and it uses gamma correction function to take into account each flow's deadline when adjusting the window size. As another variant of DCTCP, L^2DCT [25] considers flows' priority when reducing window size, and the priority is determined by the scheduling policy used in the network. ECN* [26] proposes dequeue marking for ECN to work effectively in datacenters. The aforementioned ECN marking approaches require modification of the TCP stack in end-node OS as well as minor parameter tunings at switches.

In-network feedback in datacenter networks: A few approaches have proposed to modify network switches in a way that TCP senders or middle switches can learn congestion status more quickly and accurately. D^3 [3] employs similar mechanism to RCP so that it can control flow rates to implement deadline based scheduling. DeTail [27] has implemented a new cross-layer network stack so that flows can avoid congested paths in the network, and PDQ [28] proposes distributed scheduling of flows that posses different priorities. These solutions are much harder to deploy than end-to-end solutions.

Flow scheduling in datacenter networks: Finally, we note that flow scheduling approaches, such as pFabric, PDQ, Varys, and PASE, also offer low flow completion times using prioritization and multiple queues. While some solutions intermix the congestion control and flow scheduling [29], we believe that congestion control and flow scheduling are largely orthogonal. For example, PASE adopts a DCTCP-like rate control scheme for lower priority queues [29] to ensure fairsharing and low queuing delay. Thus, in general, our latency-based feedback is orthogonal to flow scheduling approaches.

8 Conclusion

In this paper, we explore latency feedback for congestion control in data center networks. To acquire reliable latency measurements, we develop both software and hardware level solutions to measure only the network-side latency. Our measurement results show that we can achieve sub-microseconds level of accuracy. Based on the accurate latency feedback, we develop DX that achieves high utilization and low queueing delay in datacenter networks. DX outperforms DCTCP [6] with 5.33x smaller queueing delay at 1 Gbps and 1.57x at 10 Gbps in testbed experiment. The queueing delay reduction is comparable or better than HULL [7] in simulation. Our prototype implementation shows that DX has much potential to be a practical solution in the real-world datacenters.

Acknowledgements

We thank our shepherd Edouard Bugnion and anonymous reviewers for their valuable comments. We also thank Keunhong Lee for providing his own implementation of Mellanox NIC driver and Sangjin Han for the SoftNIC implementation. This research was supported in part by Cisco Research Center (No. 576768), Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Korean government (MSIP) (No. 2014007580), and an Institute for Information communications Technology Promotion (IITP) grant funded by the Korean government (MSIP) (No. B0126-15-1078).

References

[1] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the ACM SIGCOMM conference*, 2002.

[2] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-shen, and Nick Mckeown. Processor Sharing Flows in the Internet. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, 2005.

[3] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM conference*, 2011.

[4] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the Data Center Network. In *Proceedings of USENIX NSDI conference*, 2011.

[5] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. FCP: A Flexible Transport Framework for Accommodating Diversity. In *Proceedings of the ACM SIGCOMM conference*, 2013.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM conference*, 2010.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of USENIX NSDI conference*, 2012.

[8] Lawrence Brakmo and Larry Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13:1465–1480, 1995.

[9] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, December 2006.

[10] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proceedings of the ACM CoNEXT*, 2010.

[11] Mario Flajslik and Mendel Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proceedings of the USENIX ATC*, 2013.

[12] Intel DPDK: Data Plane Development Kit. `http://dpdk.org/`.

[13] Highly Accurate Time Synchronization with ConnectX-3 and TimeKeeper, Mellanox. `http://www.mellanox.com/related-docs/whitepapers/WP_Highly_Accurate_Time_Synchronization.pdf`.

[14] Iperf - The TCP/UDP Bandwidth Measurement Tool. `http://iperf.fr/`.

[15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[16] IEEE 1588: Precision Time Protocol (PTP).

[17] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal Clock Synchronization in Networks. In *Proceedings of ACM SenSys Conference*, 2009.

[18] Mosharaf Chowdhury and Ion Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *Proceedings of ACM HotNets*, 2012.

[19] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM conference*, 2009.

[20] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM conference*, 2013.

[21] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *Proceedings of the ACM SIGCOMM conference*, 2014.

[22] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proceedings of USENIX OSDI conference*, 2002.

[23] Aleksandar Kuzmanovic and Edward W Knightly. TCP-LP: A Distributed Algorithm for Low Priority Data Transfer. In *Proceedings of IEEE INFOCOM Conference*, 2003.

[24] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM conference*, 2012.

[25] Ali Munir, Ihsan Qazi, Zartash Uzmi, Aisha Mushtaq, Saad Ismail, M. Iqbal, and Basma Khan. Minimizing Flow Completion Times in Data Centers. In *Proceedings of IEEE INFOCOM Conference*, 2013.

[26] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ECN for Data Center Networks. In *Proceedings of the ACM CoNEXT*, 2012.

[27] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM conference*, 2012.

[28] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM conference*, 2012.

[29] Ali Munir, Ghufran Baig, S Irteza, I Qazi, I Liu, and F Dogar. Friends, not Foes Synthesizing Existing Transport Strategies for Data Center Networks. In *Proceedings of the ACM SIGCOMM conference*, 2014.

Mahimahi: Accurate Record-and-Replay for HTTP

Ravi Netravali*, Anirudh Sivaraman*, Somak Das*, Ameesh Goyal*, Keith Winstein†, James Mickens‡, Hari Balakrishnan*
*MIT CSAIL †Stanford University ‡Harvard University
{ravinet, anirudh, somakrdas, ameesh, hari}@csail.mit.edu, keithw@cs.stanford.edu, mickens@eecs.harvard.edu

Abstract

This paper presents Mahimahi, a framework to record traffic from HTTP-based applications, and later replay it under emulated network conditions. Mahimahi improves upon prior record-and-replay frameworks in three ways. First, it is more accurate because it carefully emulates the *multi-server* nature of Web applications, present in 98% of the Alexa US Top 500 Web pages. Second, it *isolates* its own network traffic, allowing multiple Mahimahi instances emulating different networks to run concurrently without mutual interference. And third, it is designed as a set of *composable shells*, providing ease-of-use and extensibility.

We evaluate Mahimahi by: (1) analyzing the performance of HTTP/1.1, SPDY, and QUIC on a corpus of 500 sites, (2) using Mahimahi to understand the reasons why these protocols are suboptimal, (3) developing Cumulus, a cloud-based browser designed to overcome these problems, using Mahimahi both to implement Cumulus by extending one of its shells, and to evaluate it, (4) using Mahimahi to evaluate HTTP multiplexing protocols on multiple performance metrics (page load time and speed index), and (5) describing how others have used Mahimahi.

1 Introduction

HTTP is the de facto communication protocol for client-server applications today [27]. Beyond its widespread use as an application-layer protocol for loading Web pages, HTTP is now used for mobile apps [22], video streaming [14], and instant messaging [19].

It is useful to evaluate the performance of these applications under controlled experimental conditions. For example, browser developers may wish to evaluate how changes to their document object model (DOM) and JavaScript parsers affect Web page load times, while network-protocol designers might want to understand the application-level impact of new multiplexing protocols like QUIC [30]. Similarly, a mobile app developer may wish to determine the user-perceived latency [28] for user interactions over different wireless networks.

Motivated by such questions, we developed Mahimahi[1], a framework to record traffic from applications that use HTTP, and later replay recorded traffic under emulated network conditions. Mahimahi works with any application that uses HTTP or HTTPS. Application clients (Web browsers, video players, and apps within mobile-phone emulators) can be run unmodified within Mahimahi. Additionally, Mahimahi's replay semantics can be extended to support the server-side logic of many applications, such as YouTube.

Mahimahi has three notable features that distinguish it from other record-and-replay tools such as Google's web-page-replay [11] and Fiddler [34]:

1. **Accuracy:** Mahimahi is careful about emulating the multi-server nature of Web applications. Instead of responding to all requests from a single server, Mahimahi creates a separate server for each distinct server contacted while recording. We find that emulating multiple servers is a key factor in accurately measuring Web page load times (§4.1).
2. **Isolation:** Using Linux's network namespaces [7], Mahimahi isolates its traffic from the rest of the host system, allowing multiple instances of its shells to run in parallel with no mutual interference (§4.2). Because other tools modify the network configuration of the entire host [11, 34], they cannot provide this feature.
3. **Composability and extensibility:** Mahimahi is structured as a set of UNIX shells, allowing the user to run unmodified client binaries within each shell. *RecordShell* allows a user to record all HTTP traffic for any process spawned within it. *ReplayShell* replays recorded content using local servers that emulate the application servers. To emulate network conditions, Mahimahi includes *DelayShell*, which emulates a fixed network propagation delay, and *LinkShell*, which emulates both fixed-capacity and variable-capacity links. These shells can be nested within one another, allowing the user to flexibly experiment with many different network configurations. Mahimahi makes it easy to modify these shells and add new ones; e.g., to record-and-replay YouTube videos, emulate packet losses, implement active queue management algorithms, etc. (§4.3).

We used Mahimahi to evaluate Web multiplexing protocols. We were able to easily extend Mahimahi to support QUIC, a new protocol in active development at Google. We compared HTTP/1.1, SPDY [3], and QUIC to a hypothetical optimal protocol and found that all three are suboptimal. We then used Mahimahi to understand the shortcomings of these multiplexing protocols. We found that each protocol is suboptimal because of the request serialization caused by source-level object dependencies present in today's Web pages. Resolving each dependency requires an RTT between the client and ori-

[1] Mahimahi was previously introduced in a demo [23].

Configuration	HTTP/1.1	SPDY	QUIC-toy	**Cumulus**	Optimal
1 Mbit/s, 120 ms	8.7, 15.0	8.6, 12.6	*7.6, 10.8*	**6.4, 9.8**	5.3, 8.8
14 Mbits/s, 120 ms	4.3, 6.0	3.9, 5.6	*3.8, 5.4*	**2.4, 3.6**	1.8, 2.9
25 Mbits/s, 120 ms	4.3, 6.0	3.9, 5.4	*3.6, 4.9*	**2.0, 3.2**	1.7, 2.7

Table 1: Median, 75%ile page load times, in seconds, for the Alexa US Top 500 sites for different link rates and the same minimum RTT (120 ms). Comparing the median page load times, Cumulus is between 18-33% of the hypothetical optimal, outperforming the *best of the other schemes* (shown in each row in italics) by between 19% to 80% in these configurations. Moreover, we show later that as RTT grows, the gap from optimal for HTTP/1.1, SPDY and QUIC grows quickly, whereas Cumulus is a lot closer to optimal.

gin Web servers; Mahimahi allowed us to pinpoint the problem because we were able to conduct a large number of emulation experiments under different network conditions quickly.

We used these findings to develop *Cumulus*, a new system to improve HTTP application performance, especially on long-delay paths. Cumulus has two components: the "Remote Proxy," a headless browser that the user runs on a well-provisioned cloud server, and the "Local Proxy," a transparent, caching HTTP proxy that runs on the user's computer. These two components cooperate to move the resolution of object dependencies closer to origin Web servers, reducing the effective RTT. Mahimahi's shell structure allowed us to implement Cumulus with ease by adapting RecordShell to implement the Local Proxy.

To evaluate Cumulus, we used Mahimahi yet again, this time on the same large number of network configurations used to understand HTTP/1.1, SPDY, and QUIC. Our key result is that page load times with Cumulus do not degrade dramatically with increasing round-trip times (RTTs), unlike the other multiplexing protocols. Some representative results are shown in Table 1. We have also evaluated Cumulus on AT&T's live cellular network in Boston, finding that it outperforms existing Web accelerators such as Opera Turbo [1] and Chrome Data Compression Proxy [15].

Mahimahi has been used in other projects, including an analysis of mobile app traffic patterns to compare single-path and multi-path TCP [13], and an evaluation of intelligent network selection schemes [12]. Mahimahi has also been used in Stanford's graduate networking course [41] and at Mozilla to understand and improve networking within browsers. Mahimahi and our experimental data are available under an open source license at `http://mahimahi.mit.edu`. Mahimahi has been queued for inclusion with the Debian distribution.

2 Related Work

This section describes prior work on Web record-and-replay tools and network emulation frameworks.

2.1 Record-and-replay tools

The most prominent Web page record-and-replay tools are Google's web-page-replay [11] and Telerik's Fiddler [34]. web-page-replay uses DNS indirection to intercept HTTP traffic during both record and replay, while Fiddler adjusts the system-wide proxy settings in the Windows networking stack. With both tools, all HTTP requests from a browser are sent to a proxy server that records the request and forwards it to the corresponding origin server. Responses also pass through the proxy server and are recorded and sent back to the browser.

Both tools suffer from two shortcomings. First, because they serve all HTTP responses from a single server, neither tool preserves the multi-server nature of Web applications. Consolidating HTTP resources onto a single server during replay allows browsers to use a single connection to fetch all resources, which is impossible when resources are on different servers. Mahimahi faithfully emulates the multi-server nature of Web applications, leading to more accurate measurements (§4.1).

Second, these tools do not provide isolation: the network conditions that web-page-replay and Fiddler emulate affect all other processes on the host machine. These include the link rate, link delay, and DNS indirection settings for web-page-replay, and the system proxy address, specified in the Windows networking stack, for Fiddler. During replay, this lack of isolation could lead to inaccurate measurements if cross traffic from other processes reaches the replaying proxy server. The lack of isolation also precludes multiple independent instances of web-page-replay or Fiddler from running concurrently—a useful feature for expediting experiments, or for experimenting with different applications concurrently. Mahimahi overcomes these problems by using Linux's network namespaces [7].

Other record-and-replay tools such as Timelapse/Dolos [8] and WaRR [6] target reproducible application debugging by capturing program executions (including user input and activity) and replaying them, while providing popular debugging abstractions including breakpoints. These systems are complementary to Mahimahi; they can be run within ReplayShell, which

ensures that served HTTP content, including dynamic content such as JavaScript, does not vary during replay.

2.2 Emulation Frameworks

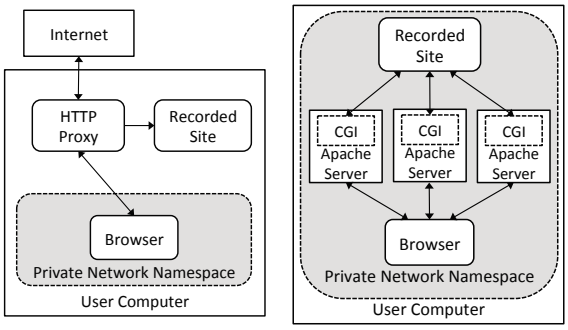
Tools like dummynet [10] and netem [20] emulate network conditions including link rate, one-way delay, and stochastic loss. Mahimahi uses its own network emulation shells, LinkShell and DelayShell. Unlike dummynet and netem, LinkShell can emulate variable-rate cellular links, in addition to static link rates, because it runs over packet-delivery traces. Mahimahi also allows users to evaluate new in-network algorithms (instead of Drop Tail FIFO) by modifying the source code of LinkShell. A similar evaluation using web-page-replay would require developing a new kernel module for dummynet, a more complicated task.

Mahimahi is general enough to record and replay any HTTP client-server application under emulated conditions. It is, however, limited in that it only emulates one physical client connected to an arbitrary number of servers. Mahimahi supports a single shared link from the client to all servers, as well as multi-homed clients (§5.5), allowing the evaluation of multipath-capable transport protocols such as MPTCP [25]. Mahimahi cannot emulate arbitrary network topologies such as transit-stub [9]; for emulating applications over such topologies, tools like Mininet [21] are more suitable.

3 MAHIMAHI

Mahimahi is structured as a set of four UNIX shells, allowing users to run unmodified client binaries within each shell. Each shell creates a new network namespace for itself prior to launching the shell. Quoting from the man page, "a network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices" [7]. A separate network namespace minimizes disruption to the host machine during recording, prevents accidental download of resources over the Internet during replay, and ensures that the host machine is isolated from all network configuration changes that are required to evaluate an application.

RecordShell (§3.1) records all HTTP traffic for subsequent replay. *ReplayShell* (§3.2) replays previously recorded HTTP content. *DelayShell* (§3.3) delays all packets originating from the shell by a user-specified amount and *LinkShell* (§3.4) emulates a network link by delivering packets according to a user-specified packet-delivery trace. All components of Mahimahi run on a single physical machine (which we call the host machine) and can be arbitrarily composed with each other. For example, to replay recorded content over a cellular network with a 10 ms minimum RTT, one would run a client application inside DelayShell inside LinkShell inside ReplayShell.



(a) RecordShell (b) ReplayShell

Figure 1: RecordShell has a transparent proxy for HTTP traffic. ReplayShell handles all HTTP traffic inside a private network namespace. Arrows indicate the direction of HTTP Request and Response traffic.

3.1 RecordShell

RecordShell (Figure 1a) records HTTP data and stores it on disk in a structured format for subsequent replay. On startup, RecordShell spawns a man-in-the-middle proxy on the host machine to store and forward all HTTP traffic both to and from an application running within RecordShell. To operate transparently, RecordShell adds an iptable rule that forwards all TCP traffic from within RecordShell to the man-in-the-middle proxy.

When an application inside RecordShell attempts to connect to a server, it connects to the proxy instead. The proxy then establishes a TCP connection with the application, uses the `SO_ORIGINAL_DST` socket option to determine the server's address for the connection, and connects to the server on the application's behalf. An HTTP parser running at the proxy captures traffic passing through it to parse HTTP requests and responses from TCP segments. Once an HTTP request and its corresponding response have both been parsed, the proxy writes them to disk, associating the request with the response. At the end of a record session, a recorded directory consists of a set of files, one for each HTTP request-response pair seen during that session.

SSL traffic is handled similarly by splitting the SSL connection and establishing two separate SSL connections: one between the proxy and the application and another between the proxy and the server. The proxy can establish a secure connection with the application in two ways. In the first approach, RecordShell's proxy uses a new Root CA, in the same way Fiddler does [35]. Clients must manually trust this CA once and individual certificates are signed by this Root CA.

Another approach is for RecordShell's proxy to use a self-signed certificate. This approach may trigger warn-

ings within applications that only accept certificates signed by any one of a list of trusted Certificate Authorities (CAs). Most modern browsers allow users to disable these warnings. Certain applications, such as mobile phone emulators, do not allow these warnings to be disabled; the first approach handles these applications [31].

3.2 ReplayShell

ReplayShell (Figure 1b) also runs on the test machine and mirrors the server side of Web applications using content recorded by RecordShell. ReplayShell accurately emulates the multi-server nature of most Web applications today by spawning an Apache 2.2.22 Web server for each distinct IP/port pair seen while recording. Each server handles HTTPS traffic using Apache's `mod_ssl` module and may be configured to speak HTTP/1.1 or SPDY (using `mod_spdy`).

To operate transparently, ReplayShell binds each Apache server to the same IP address and port number as its recorded counterpart. To do so, ReplayShell creates a separate dummy (virtual) interface for each distinct server IP. These interfaces can have arbitrary IPs because they are in a separate network namespace.

All client requests are handled by one of ReplayShell's servers, each of which can read all of the previously recorded content. Each server redirects all incoming requests to a CGI script using Apache's `mod_rewrite` module. The CGI script on each server compares each incoming HTTP request to the set of all recorded request-response pairs to locate a matching request and return the corresponding response. Incoming requests may be influenced by local state present in the client application (e.g. time-sensitive query string parameters) and may not exactly match any recorded request. We handle such requests using a matching heuristic that enforces that some parts of the request must match exactly, while tolerating some degree of imperfection in other parts.

We expect the Host and User-Agent header fields, along with the requested resource (without the query string), to exactly match the corresponding values in some stored request. If multiple stored requests match on these properties, the algorithm selects the request whose query string has the maximal common substring to the incoming query string.

3.3 DelayShell

DelayShell emulates a link with a fixed minimum one-way delay. All packets sent to and from an application running inside DelayShell are stored in a packet queue. A separate queue is maintained for packets traversing the link in each direction. When a packet arrives, it is assigned a delivery time, which is the sum of its arrival time and the user-specified one-way delay. Packets are released from the queue at their delivery time. This technique enforces a fixed delay on a per-packet basis.

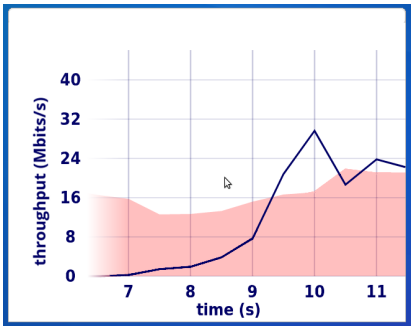


Figure 2: LinkShell supports live graphing of network usage, comparing the link capacity of the input traces (red shading) to the amount of data a client application attempts to transmit (blue line).

3.4 LinkShell

LinkShell emulates a link using packet-delivery traces. It emulates both time-varying links such as cellular links and links with a fixed link rate. When a packet arrives into the link, it is directly placed into either the uplink or downlink packet queue. LinkShell is trace-driven and releases packets from each queue based on the corresponding packet-delivery trace. Each line in the trace is a packet-delivery opportunity: the time at which an MTU-sized packet will be delivered in the emulation.[2] Accounting is done at the byte-level, and each delivery opportunity represents the ability to deliver 1500 bytes. Thus, a single line in the trace file can correspond to the delivery of several packets whose sizes sum to 1500 bytes. Delivery opportunities are wasted if bytes are unavailable at the instant of the opportunity.

LinkShell supports live graphing of network usage and per-packet queuing delay, giving near-instantaneous feedback on the performance of applications and network protocols. Uplink and downlink capacity are calculated using the input packet-delivery traces, while network usage, in each direction, is based on the amount of data that a client application attempts to transmit or receive. Per-packet queuing delay is computed as the time each packet remains in LinkShell's uplink or downlink queues.

Figure 2 illustrates the downlink network usage of a single Web page load of `http://www.cnn.com`, using Google Chrome over an emulated Verizon LTE cellular network with a minimum RTT of 100 ms. As shown, Web servers try to exceed the link capacity at around 9.3 seconds into the trace.

[2]For example, a link that can pass one MTU-sized packet per millisecond (12 Mbits/s) can be represented by a file that contains just "1" (LinkShell repeats the trace file when it reaches the end).

4 NOVELTY

Mahimahi introduces three new features in comparison to existing record-and-replay tools. We describe each of these in greater detail below.

4.1 Multi-server emulation for greater accuracy

A key component of ReplayShell is that it emulates the multi-server nature of Web applications. As discussed in §3, ReplayShell creates a network namespace containing an Apache server for each distinct server encountered in a recorded directory. We show through three experiments that emulating this multi-server nature is critical to the accurate measurement of Web page load times.

A large number of websites today are multi-server. We measure the number of physical servers used by each site in the the Alexa US Top 500 [5]. We find that the median number of servers is 20, the 95%ile is 51, and the 99%ile is 58. Only 9 of the 500 Web pages (1.8%) we consider use a single server.

Next, we illustrate the importance of preserving the multi-server nature of Web applications by comparing measurements collected using ReplayShell and web-page-replay to real page load times on the Internet. To obtain measurements on the Internet, we use Selenium to automate Google Chrome loading 20 Web pages from the Alexa US Top 500, 25 times each, inside a LinkShell of 5 Mbits/s and a DelayShell with a minimum RTT of 100 ms. We chose a minimum RTT of 100 ms to equalize delays to Web servers contacted while loading each Web page.[3] For a fair comparison, we record copies of each Web page with RecordShell and web-page-replay immediately following the completion of these Internet measurements; Web content can change frequently, which can significantly affect page load time. We then replay each recorded Web page 25 times using ReplayShell, a modified version of ReplayShell that serves all resources from a single server, and web-page-replay. With ReplayShell, we perform each page load inside LinkShell with a 5 Mbits/s trace and DelayShell with a minimum RTT of 100 ms, as described above. We emulate these same network conditions with web-page-replay.

We define the error, per site, as the absolute value of the percent difference between mean page load times (over 25 runs) within an emulation environment and on the Internet. As shown in Figure 3, ReplayShell with multi-server emulation yields page load times that most accurately resemble page load times collected on the Internet. The median error is 12.4%, compared to 36.7% and 20.5% with web-page-replay and single-server ReplayShell, respectively.[4]

[3]The 20 sites used here are all hosted by CDNs in close proximity with ping times of less than 5 ms.

[4]We are not certain why single-server ReplayShell is so much more accurate than web-page-replay.

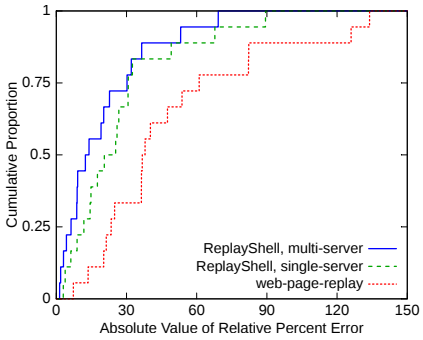


Figure 3: Preserving a Web page's multi-server nature yields measurements that more closely resembles measurements on the Internet.

	30 ms	120 ms	300 ms
1 Mbit/s	1.6%, 27.6%	1.7%, 10.8%	2.1%, 9.7%
14 Mbits/s	19.3%, 127.3%	6.2%, 42.4%	3.3%, 20.3%
25 Mbits/s	21.4%, 111.6%	6.3%, 51.8%	2.6%, 15.0%

Table 2: Median and 95%ile difference in page load time without multi-server emulation.

Finally, we run more exhaustive experiments to show the effect that multi-server emulation has on Web page load times across different network conditions. Using an Amazon EC2 m3.large instance located in the US-east-1a region and running Ubuntu 13.10, we measure page load times for each recorded page in the Alexa US Top 500 when loaded with Google Chrome. We consider 9 different configurations: link rates in $\{1, 14, 25\}$ Mbits/s and RTTs in $\{30, 120, 300\}$ ms. We load each page over each configuration using both ReplayShell and the modified version of ReplayShell used above that eliminates the multi-server nature altogether by setting up one Apache server to respond to all HTTP requests and resolving all DNS queries to that server alone.

Table 2 shows the median and 95%ile difference in page load time when multi-server nature is not preserved, compared to when multi-server nature is preserved. Although the page load times are comparable over a 1 Mbit/s link, the lack of multi-server emulation yields significantly worse performance at higher link rates.

4.2 Isolation

By creating a new network namespace for each shell, Mahimahi eliminates much experimental variability that results from interfering cross traffic during an experiment. Each namespace is separate from the host machine's default namespace and every other namespace and thus, processes run inside the namespace of a Mahimahi tool are completely isolated from those running directly on the host or in other namespaces. As a result, host machine traffic does not affect the measurements reported by Mahimahi. Similarly, network emulation done by Mahimahi's tools does not affect traffic outside of Mahimahi's network namespaces. This prop-

	Machine 1	Machine 2
CNBC	7584 ms +- 120 ms	7612 ms +- 111 ms
wikiHow	4804 ms +- 37 ms	4800 ms +- 37 ms

Table 3: Mean and standard deviation for page load times across two similarly configured machines.

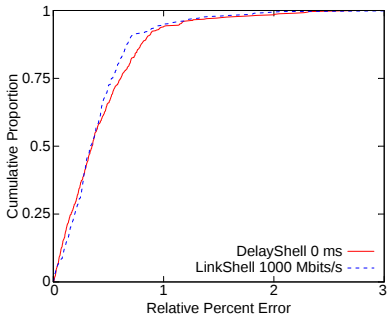


Figure 4: DelayShell and LinkShell have a negligible effect on page load times in ReplayShell.

erty of Mahimahi, along with the fact that its shells can be arbitrarily nested, enables many different configurations to be simultaneously tested on a host machine, in complete isolation from one another.

Using distinct network namespaces for each shell also enables Mahimahi to produce reproducible results while imposing low overhead on collected measurements.

Reproducibility: To evaluate the reproducibility of Mahimahi's measurements, we perform repeated experiments on the same host machines and across different host machines with similar hardware specifications. We choose two sites from the Alexa US Top 500 for this experiment, `http://www.cnbc.com/` and `http://www.wikihow.com/`, as they are are at the median and 95%ile site sizes (1.2 MB and 5.5 MB, respectively).

We use two different Amazon EC2 m3.large instances, each in the US-east-1a region and running Ubuntu 13.10. On each machine, we load the CNBC and wikiHow Web pages 100 times each inside ReplayShell, over a 14 Mbits/s link with a minimum RTT of 120 ms. Table 3 shows a summary of the distribution of page load times from these experiments. Mean page load times for each site are less than 0.5% apart across the two machines suggesting that Mahimahi produces comparable results across different host machines. Similarly, standard deviations are all within 1.6% of their corresponding means, implying that Mahimahi produces consistent results on a single host machine.

Fidelity: Mahimahi's shells impose low overhead on collected measurements, even when they are nested within one another, leading to high fidelity in the results. We illustrate this property in Figure 4, which shows the overhead DelayShell and LinkShell impose on page load time measurements. We first load our recorded copies of the Alexa US Top 500 sites inside ReplayShell, without LinkShell or DelayShell. For comparison, we then load the 500 sites inside DelayShell, with 0 ms fixed per-packet delay, inside ReplayShell. Separately, we load the 500 sites inside LinkShell, with 1000 Mbits/s uplink and downlink traces, inside ReplayShell.[5] Each of these experiments was performed on the same Amazon EC2 m3.large instance configured with Ubuntu 13.10 and located in the US-east-1a region. We find that the median per-site errors with DelayShell and LinkShell, relative to ReplayShell alone, are 0.33% and 0.31%, respectively.

4.3 Composability and extensibility

Unmodified application clients can be run within any of Mahimahi's shells. For instance, as described in §5.5, a mobile device emulator can be run within Mahimahi to measure mobile app performance. Similarly, to measure new performance metrics such as the speed index, virtual machines can be run within Mahimahi's shells (§5.4).

The default replay algorithm is but one instance of a server-side HTTP matching algorithm. Mahimahi's replay semantics can be easily extended to support the server-side logic of many other applications and multiplexing protocols; for example, in §5.1.1, we extend ReplayShell to use QUIC Web servers rather than default Apache Web servers. It has also been extended to handle record-and-replay for YouTube videos (§5.5).

In addition to DelayShell and LinkShell, which emulate different minimum RTTs and link rates, Mahimahi can be extended to support other network characteristics. For example, to emulate different levels of stochastic packet loss, we created *LossShell* [24], which probabilistically drops packets stored in LinkShell's upstream and downstream queues. Similarly, Mahimahi can be modified to evaluate in-network algorithms such as queuing disciplines. By default, LinkShell implements a Drop Tail FIFO queue, but we have extended it to implement CoDel, an active queue management scheme [32].

Mahimahi could also be used to replay recorded content to a different physical machine. Consider a scenario where the application to be evaluated is only available on Machine *M*, and a separate Linux Machine, *A*, is available. An *EthShell* could ferry packets from an Ethernet interface between *M* and *A* to a virtual network interface on *A*. Analogously, a *UsbShell* could ferry packets between an Ethernet-over-USB interface connected to a phone and a virtual interface on *A*. UsbShell could be used to run performance regression tests on actual phones rather than emulators. Neither of these has been developed yet, but Mahimahi's design allows these shells to be nested inside any of Mahimahi's existing shells. For instance, to test a mobile phone's browser over an LTE link with a 100 ms RTT, we would nest UsbShell inside DelayShell inside LinkShell inside ReplayShell.

[5] We chose 1000 Mbits/s to ensure that link capacity was not a limiting factor in page load time.

5 Case Studies

5.1 Understanding Web Performance

We use Mahimahi to evaluate Web page load times under three multiplexing protocols: HTTP/1.1, SPDY [3], and QUIC [30], a protocol currently in development at Google. To put these measurements in context, we compare each protocol with an optimal protocol for each network configuration.

To automate the page load process and measure page load times, we use Selenium, a widely used browser-automation tool, along with Chrome Driver version 2.8 and the Web Driver API [38]. We measure page load time by calculating the time elapsed between the navigationStart and loadEventEnd events [38].

In all evaluations, traffic originates from the Web browser alone. We emulate link rates and minimum RTTs (§3), but do not emulate competing cross traffic. For each network configuration, we emulate a buffer size of 1 bandwidth-delay product and evaluate all sites in the Alexa US Top 500.

5.1.1 Setup

HTTP/1.1: We evaluate HTTP/1.1 using ReplayShell running unmodified Apache 2.2.22.

SPDY: To evaluate SPDY, we create *SPDYShell*, which enables the `mod_spdy` extension on all Apache servers within ReplayShell. The SPDY configuration evaluated here does not include server push because the push policy is specific to each website and is hard to infer automatically. If push policies were known, however, the CGI script within ReplayShell's servers could be modified to reflect them.

QUIC: QUIC inherits several SPDY features, such as multiplexing streams onto a single transport-protocol connection and stream priorities. By using UDP and its own security instead of TCP and TLS, QUIC overcomes two drawbacks of SPDY: head-of-line blocking between streams due to lost packets and the three-way handshake required to establish a secure connection.

Unlike SPDY, Apache currently has no extensions for QUIC. We create *QUICShell* by replacing Apache within ReplayShell with an adapted version of the QUIC toy server [29] from the Chromium project (commit *5bb5b95* from May, 2015, available at `https://goo.gl/Jdr8hi`). We modify the toy server, which originally searched for exact URL matches, to use the matching semantics in ReplayShell's CGI script.

5.1.2 Optimal page load time

We define the optimal page load time for a website as:

$$minimumRTT + (siteSize/linkRate) + browserTime.$$

The first term represents the minimum time between when the first HTTP request is made at the client and the first byte of the first HTTP response is received by the client, ignoring processing time at the server.

The second term represents the minimum time to transfer all bytes belonging to the Web page over a fixed capacity link. We calculate the site size by counting the total number of bytes delivered over the emulated link from the Web servers to the browser between the navigationStart and loadEventEnd events.

The third term represents the time for the browser to process all the HTTP responses and render the Web page (using the definition of "loaded" above). We measure this as the page load time in ReplayShell alone without network emulation, emulating an infinite-capacity, zero-delay link.

5.1.3 Canonical network results

We evaluate each protocol on 110 configurations: link rates in $\{0.2, 0.3, 0.6, 1, 1.7, 2.9, 5, 8.5, 14, 25\}$ Mbits/s and RTTs between 0 ms and 300 ms in steps of 30 ms. These link rates and RTTs cover the majority of global network conditions reported by Akamai [4]. We also perform evaluations over cellular networks using modified versions of the Verizon and AT&T traces collected in [40]. For each network configuration, we compare HTTP/1.1, SPDY, and QUIC (and in the next subsection, Cumulus) with the optimal page load times defined above.

Figure 5 shows the distributions of page load times with each protocol for six of these configurations: 1 Mbit/s and 25 Mbits/s, with RTTs of 30 ms, 120 ms, and 300 ms. We find that the gap from optimal for HTTP/1.1, SPDY, and QUIC grows quickly with the RTT, and grows with the link rate (although not as quickly). For example, on a 1 Mbit/s link with a minimum RTT of 30 ms, the median page load time for SPDY is 1.08× worse than optimal. When the minimum RTT increases to 120 ms, the median SPDY page load time is 1.63× worse than optimal, worsening to 2.02× worse than optimal when the minimum RTT rises to 300 ms. For this RTT, increasing the link rate from 1 Mbit/s to 25 Mbits/s degrades median SPDY performance to 4.93× worse than optimal.

5.1.4 Understanding suboptimality

In addition to quantifying the extent of suboptimality of multiplexing protocols for the Web, the results presented in this case study corroborate the qualitative findings of many previous measurement studies [26, 37, 39]. We used Mahimahi in conjunction with browser developer tools to identify the root cause of this suboptimality. We found that the suboptimal performance of each multiplexing protocol is a result of request serialization caused by source-level dependencies between objects on a Web page; this problem is exacerbated by small limits on the number of concurrent connections from the browser, but persists even if those browser limits are removed.

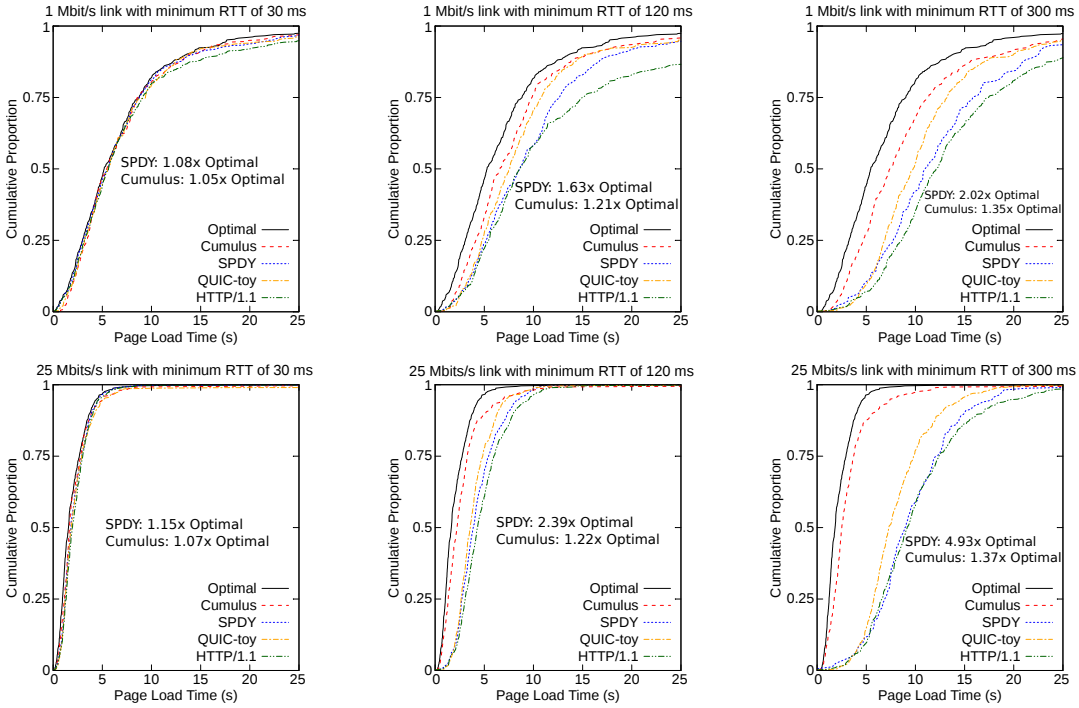


Figure 5: The gap between page load times with HTTP/1.1, SPDY, or QUIC and Optimal grows as link rate or minimum RTT increases. Cumulus is introduced in §5.2.

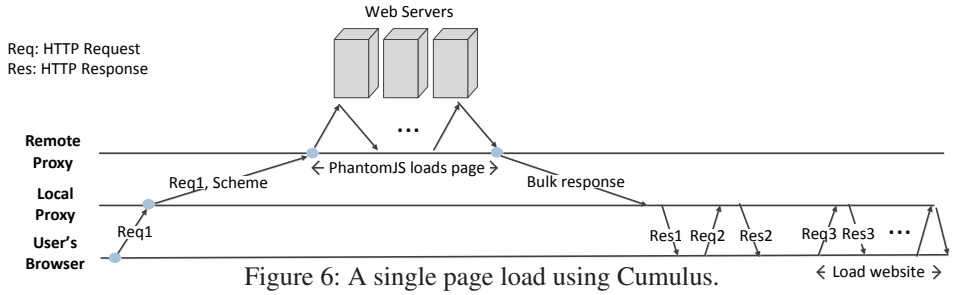


Figure 6: A single page load using Cumulus.

The fundamental issue is that resolving each dependency requires a round-trip communication between the client and origin Web servers. As a result, the negative effect of request serialization is more pronounced at high RTTs (Figure 5). This finding motivated us to develop Cumulus, a system that uses Mahimahi to improve page load times on long-delay paths.

5.2 Improving Web performance with Cumulus

Cumulus has two components: the "Remote Proxy," a headless browser that the user runs on a well-provisioned cloud server, and the "Local Proxy," a transparent, caching HTTP proxy that runs on the user's computer. These two components cooperate to move the resolution of object dependencies closer to origin Web servers—reducing the effective RTT—without modifying Web browsers or servers.

The Remote Proxy listens for new requests from the Local Proxy. For each incoming request, the Remote Proxy launches an unmodified RecordShell and runs a headless browser, PhantomJS [2], to load the specified URL using the original HTTP headers. Once the page is loaded, the Remote Proxy packages and compresses the recorded HTTP request/response pairs into a *bulk response*, which it sends to the Local Proxy.

The Local Proxy is a modified version of RecordShell that caches HTTP objects rather than storing them in files. When the user's browser requests a URL not resident in the Local Proxy's cache, the Local Proxy forwards the request to the Remote Proxy. Upon receiving a bulk response from the Remote Proxy, the Local Proxy responds to the user's browser with the appropriate response and caches the remaining objects to handle subsequent browser requests. Figure 6 illustrates how Cumulus loads a single Web page.

5.3 Evaluating Cumulus with Mahimahi

We first evaluate Cumulus over each emulated network configuration listed in §5.1.3. Page loads with Cumulus used Google Chrome and a Remote Proxy running on the

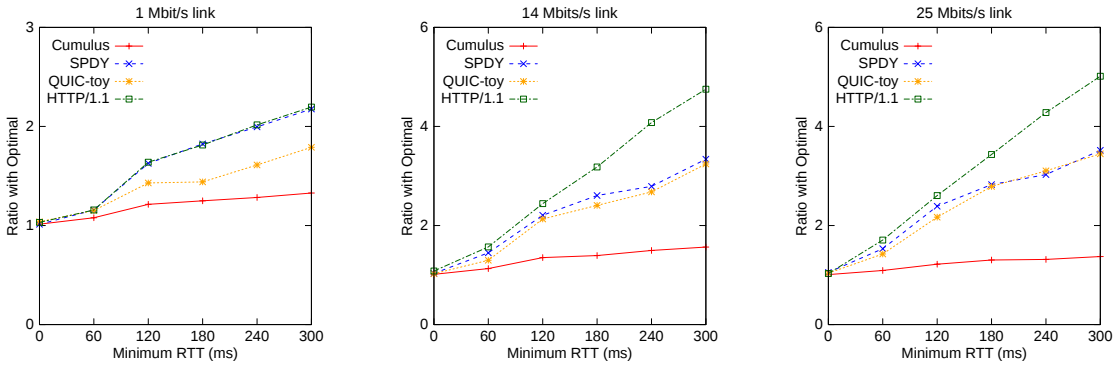


Figure 7: Cumulus's performance does not degrade dramatically as RTTs increase (at fixed link rates), unlike HTTP/1.1, SPDY, and QUIC. Each point plots the ratio of median protocol performance to median performance of the optimal scheme (lower is better).

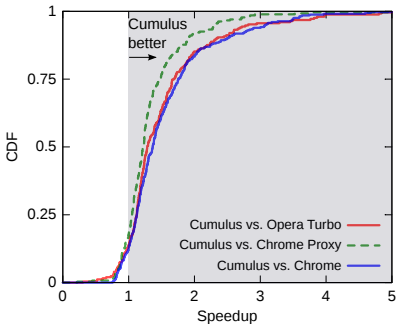


Figure 8: Evaluating Cumulus on the live AT&T Cellular Network in Boston.

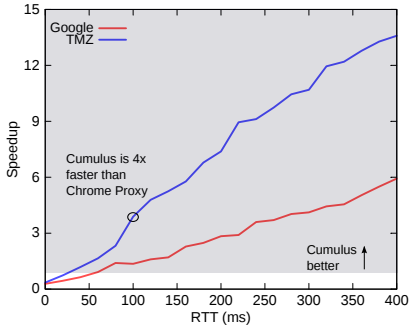


Figure 9: Benefits with Cumulus increase as RTT or Web page complexity increase.

other side of each emulated link. We find that Cumulus outperforms SPDY by 1.03–3.60× over these configurations (Figure 5). Figure 7 shows how the ratio between median page load times with each protocol and the optimal varies as RTTs increase at fixed link rates. We find that Cumulus is less affected by increases in RTT compared to today's multiplexing protocols. For example, at a link rate of 14 Mbits/s and an RTT of 60 ms, Cumulus is 1.13× worse than optimal while SPDY is 1.44× worse than optimal. When RTT increases to 180 ms, Cumulus is 1.39× worse, whereas SPDY is 2.61× worse than optimal.

5.3.1 *Some live experiments*

We also compare the performance of Google Chrome run inside Cumulus with Chrome, and with Chrome Data Compression Proxy [15, 16] and Opera Turbo [1], which are cloud browsers that use proxy servers for compression. We load each page in the Alexa US Top 500 five times with each system, rotating among the systems under test to mitigate the effects of network variability. We define Cumulus's "speedup" relative to a system as the ratio of the page load time using that system to the page load time using Cumulus.

We ran experiments over the live AT&T LTE/GSM/WCDMA cellular network in Boston using a PC laptop tethered to a Samsung Galaxy Note running Android OS version 4.2.2. Cumulus used a Remote Proxy running on an Amazon EC2 instance in Virginia. Cumulus had median speedups of 1.36×, 1.23×, and 1.28× over Chrome, Chrome Data Compression Proxy, and Opera Turbo, respectively. Figure 8 shows the CDF of speedups.

5.3.2 *Understanding Cumulus' gains*

Cumulus moves dependency resolution to the Remote Proxy where RTTs to Web servers are lower than from the client. The benefit of this technique depends on:

1. The RTT between the user and origin Web servers.
2. The complexity of the Web page.

To understand the importance of each factor, we use Mahimahi's shell abstraction to load two Web pages in emulation: TMZ's homepage with 508 objects and the Google homepage with only 15 objects. We use DelayShell to emulate fixed minimum RTTs from 0 ms to 400 ms. For each RTT, we load each page five times with Chrome Data Compression Proxy—which compresses objects in-flight, but does not perform dependency resolution on the user's behalf—and Cumulus, which performs dependency resolution *and* compresses objects in-flight.

Page loads with Cumulus used a Remote Proxy running on the other side of the emulated long-delay link. Speedups for Cumulus relative to Chrome Data Compression Proxy are shown in Figure 9.

We observe two trends:

1. For a given Web page, speedups with Cumulus increase as RTT increases.
2. For a fixed RTT, speedups with Cumulus are larger for more complex Web pages.

Our results show a 4× speedup relative to Chrome Data Compression Proxy at an RTT of 100 ms, a typical RTT for cellular and transcontinental links. This corroborates the well-known intuition that Web page load times are dominated by network latencies rather than link rates, and suggests that the combination of remote dependency resolution and object compression helps Cumulus achieve performance not far from optimal.

5.4 Speed index

All of our measurements thus far have been of page load time. We now show that it is straightforward to use a different performance metric. We use Google's proposed speed index [17] as an example.

5.4.1 Definition

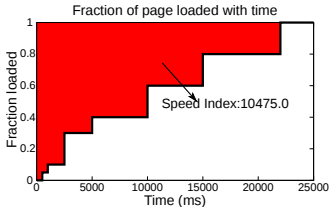
Page load time may not accurately measure when a page is usable by the client. For long Web pages, content "above-the-fold" of the screen is important to retrieve quickly, but other content may not be. Taking this point into consideration for measurement is especially relevant for pages that support infinite scrolling. For example, Facebook "preloads" wall postings below the user's current location on its page in anticipation of a user scroll. In such cases, the "onload" event used to measure page load time would fire long after the page is ready for user interaction. Speed index is an attempt to address this issue.

Speed index tracks the visual progress of a Web page in the visible display area. A lower speed index signifies that the content is rendered more quickly. For example, a page that immediately paints 90% of its visual content will receive a lower speed index than a page that progressively paints 90% of its content, even if both pages fire their onload event at the same time.

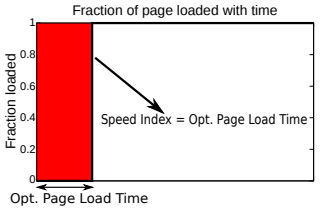
Speed index is calculated by measuring the completeness of a page's display area over time. Completeness is defined as the pixel-by-pixel difference of a page snapshot with the final loaded Web page. Once the entire page has loaded, the completeness percentage of the page rendering over time is plotted. Speed index is defined as the area "above-the-curve" (Figure 10a).

5.4.2 Measuring speed index

We calculate speed index using WebPagetest [17], which records videos of page loads at 10 frames per second



(a) Speed index is the area above the curve of the completeness of a page load as a function of time.



(b) We define an upper bound on optimal speed index by assuming that a page instantaneously jumps from 0% to 100% completeness at the optimal page load time.

Figure 10: Speed index calculation.

and plots the percentage completeness over time by comparing each frame with the final captured frame. To measure speed index, we create *SpeedIndexShell* where we run a private instance of WebPagetest inside ReplayShell. To automate testing, we use WebPagetest's `wpt_batch.py` API [18]. Because WebPagetest runs only on Windows, we run WebPagetest within a VirtualBox Windows virtual machine, inside ReplayShell.

5.4.3 Optimal speed index

Calculating an optimal speed index is difficult. Instead, we define an upper bound[6] on the optimal speed index. We assume that a site renders in one shot at the optimal page load time; Figure 10b illustrates its implications on the "optimal" speed index. As shown, the percentage completeness of a given Web page is 0% until the optimal page load time where the percentage completeness jumps to 100%. As a result, the "area above the curve," or optimal speed index, equals the optimal page load time. There could be better rendering strategies that more gradually render the page between 0 and the optimal page load time, but such improved characterizations of the optimal speed index will only further increase the already large slowdowns (Figure 11) from the optimal speed index.

5.4.4 Static link results

We measure the speed index for each site in the Alexa US Top 500 over networks with link rates between 1 Mbit/s and 25 Mbits/s and a fixed minimum RTT of 120 ms (Figure 11). We notice similar patterns to those discussed with page load times: the gap between speed index with HTTP/1.1 and optimal speed index grows as link rates

[6] Recall that a lower speed index is better.

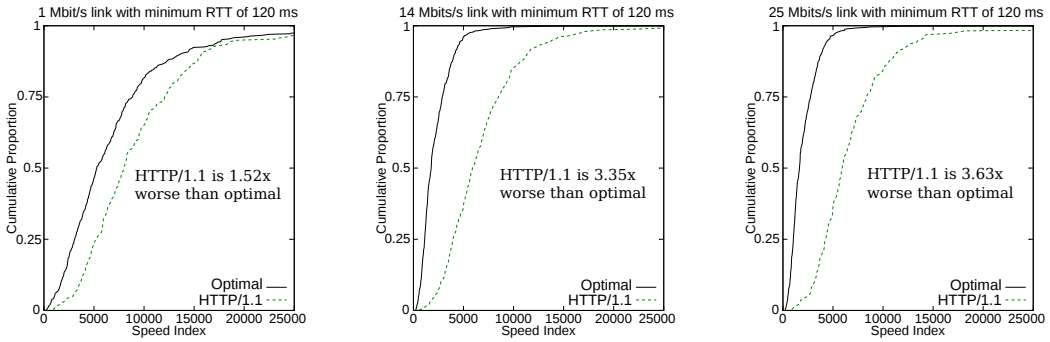


Figure 11: Gap between speed index with HTTP/1.1 and Optimal grows as link rate increases (fixed minimum RTT).

increase; over a 1 Mbit/s link with a 120 ms minimum RTT, speed index with HTTP/1.1 is 1.52× worse than optimal at the median, while over a 25 Mbits/s link with a 120 ms minimum RTT, the median speed index with HTTP/1.1 is 3.63× worse than optimal.

5.5 External case studies

This section describes external use cases of Mahimahi in research, educational, and industrial settings.

Mobile app record-and-replay: RecordShell has been used to characterize mobile app traffic by recording all HTTP traffic to and from mobile apps running inside an Android emulator [13]. Using this recorded traffic, they evaluated the performance of mobile apps over Wi-Fi and LTE networks by running an Android emulator inside ReplayShell to measure the duration of data transfers for mobile apps over these wireless networks. The results showed that LTE outperforms Wi-Fi 40% of the time on flow completion time.

Mobile multi-homing: To emulate mobile multi-homing with Wi-Fi and LTE, the authors in [12] extended LinkShell to create *MpShell* [33]. They then compared single-path TCP and MPTCP by replaying mobile app traffic over 20 different emulated network conditions.

Record-and-replay for video streaming: Mahimahi has been extended to handle record and replay for YouTube videos [36]. Compared to Web pages, video replay requires more involved matching logic on the server side. HTTP requests encode the location (start and end time) and quality of video chunks requested by the client's video player. Both the location and quality attributes can change significantly from run to run, and between record and replay.

Educational uses: Mahimahi is being used by students in Stanford's graduate networking course [41] to understand the performance of their networked applications under controlled conditions. As part of a protocol design contest conducted in the same course, students used LinkShell's live graphing of network usage and per-packet queuing delay to obtain real-time feedback on the performance of their congestion-control protocols.

Browser networking: Engineers at Mozilla are using Mahimahi to improve the speed of Firefox's networking. Here, Mahimahi is helpful in understanding how improvements to link utilization and pipelining of HTTP requests affect Web performance over various networks.

6 Conclusion

Mahimahi is an accurate and flexible record-and-replay framework for HTTP applications. Mahimahi's shell-based design makes it composable and extensible, allowing the evaluation of arbitrary applications and network protocols. It accurately emulates the multi-server nature of Web applications during replay, and by isolating its own traffic, allows several instances to run in parallel without affecting collected measurements.

We presented several case studies to evaluate Mahimahi and demonstrate its benefits. These include a study of HTTP/1.1, SPDY, and QUIC under various emulated network conditions. We used Mahimahi both to conduct the experiments and to understand the reasons for the suboptimality of these protocols. We then used our key finding—that these protocols are suboptimal due to source-level dependencies in Web pages—to design Cumulus. Mahimahi was useful in our implementation of Cumulus, as well as in our experiments to measure its performance. As round-trip times and link rates increase, the performance of Cumulus degrades much slower than previous HTTP multiplexing protocols.

We have released Mahimahi under an open source license at `http://mahimahi.mit.edu`.

7 Acknowledgements

We thank Amy Ousterhout, Pratiksha Thaker, the ATC reviewers, and our shepherd, Liuba Shrira, for their helpful comments and suggestions. This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-1407470. We thank the members of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support.

REFERENCES

[1] Opera Turbo. http://www.opera.com/turbo.

[2] PhantomJS. http://phantomjs.org/.

[3] SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[4] Akamai. State of the Internet. http://www.akamai.com/stateoftheinternet/, 2013.

[5] Alexa. Top sites in the United States. http://www.alexa.com/topsites/countries/US.

[6] S. Andrica and G. Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN*, 2011.

[7] E. W. Biederman. ip-netns. http://man7.org/linux/man-pages/man8/ip-netns.8.html.

[8] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *UIST*, 2013.

[9] K. Calvert, M. Doar, and E. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, 1997.

[10] M. Carbone and L. Rizzo. Dummynet revisited. *SIGCOMM CCR*, 40(2):12–20, 2010.

[11] Chromium. web-page-replay. https://github.com/chromium/web-page-replay.

[12] S. Deng. Intelligent Network Selection and Energy Reduction for Mobile Devices. http://people.csail.mit.edu/shuodeng/papers/thesis.pdf.

[13] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. WiFi, LTE, or both? Measuring multi-homed wireless Internet performance. In *IMC*, 2014.

[14] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube traffic characterization: A view from the edge. In *IMC*, 2007.

[15] J. Glowacki. Data compression proxy. https://chrome.google.com/webstore/detail/data-compression-proxy/ajfiodhbiellfpcjjedhmmmpeeaebmep.

[16] Google. Data compression proxy. https://developer.chrome.com/multidevice/data-compression.

[17] Google. Speed Index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[18] Google. WebPagetest batch processing APIs. https://sites.google.com/a/webpagetest.org/docs/advanced-features/webpagetest-batch-processing-apis.

[19] R. Jennings, E. Nahum, D. Olshefski, D. Saha, Z.-Y. Shae, and C. Waters. A study of Internet instant messaging and chat protocols. *Network, IEEE*, 20(4):16–21, 2006.

[20] A. Jurgelionis, J. Laulajainen, M. Hirvonen, and A. Wang. An empirical study of netem network emulation functionalities. In *ICCCN*, 2011.

[21] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *HotNets*, 2010.

[22] K. Ma, R. Bartos, S. Bhatia, and R. Nair. Mobile video delivery with HTTP. *Communications Magazine, IEEE*, 49(4):166–175, 2011.

[23] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, and H. Balakrishnan. Mahimahi: A lightweight toolkit for reproducible web measurement (demo). In *SIGCOMM*, 2014.

[24] R. Netravali, K. J. Winstein, and A. Sivaraman. LossShell. https://github.com/ravinet/mahimahi/tree/lossshell.

[25] C. Paasch, S. Barre, et al. Multipath TCP in the Linux kernel. http://multipath-tcp.org/.

[26] J. Padhye and H. F. Nielsen. A comparison of SPDY and HTTP performance. Technical report, Microsoft, 2012.

[27] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *Hotnets*, 2010.

[28] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *OSDI*, 2012.

[29] J. Roskind. Experimenting with QUIC. http://blog.chromium.org/2013/06/experimenting-with-quic.html.

[30] J. Roskind. QUIC: Multiplexed stream transport over UDP. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit .

[31] N. Rudrappa. Defeat SSL Certificate Validation for Google Android Applications. Technical report, McAfee, 2013.

[32] A. Sivaraman, R. Netravali, and K. J. Winstein. CodelShell. https://github.com/ravinet/mahimahi/releases/tag/old/codel.

[33] A. Sivaraman, R. Netravali, and K. J. Winstein. MPShell. https://github.com/ravinet/mahimahi/

tree/old/mpshell_scripted.
[34] Telerik. Fiddler. http://www.telerik.com/fiddler.
[35] Telerik. Fiddler documentation. http://docs.telerik.com/fiddler/Configure-Fiddler/Tasks/TrustFiddlerRootCert.
[36] V. Vasiliev, R. Netravali, K. J. Winstein, and A. Sivaraman. YoutubeShell. https://github.com/vasilvv/mahimahi.
[37] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.
[38] Z. Wang and A. Jain. Navigation timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html.
[39] G. White, J.-F. Mule, and D. Rice. Analysis of SPDY and TCP initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00.
[40] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, 2013.
[41] K. J. Winstein. (Your) great ideas for networked applications. https://web.stanford.edu/class/cs344g/.

Slipstream: Automatic Interprocess Communication Optimization

Will Dietz, Joshua Cranmer, Nathan Dautenhahn, and Vikram Adve
University of Illinois at Urbana-Champaign
{wdietz2,cranmer2,dautenh1,vadve}@illinois.edu

Abstract

We present **Slipstream**, a userspace solution for transparently selecting efficient local transports in distributed applications written to use TCP/IP, when such applications communicate between local processes. Slipstream is easy to deploy because it is language-agnostic, automatic, and transparent. Our design in particular (1) requires no changes to the kernel or applications, (2) correctly identifies (and optimizes) pairs of communicating local endpoints, without knowledge of routing performed locally or by the network, and (3) imposes little or no overhead when optimization is not possible, including communication with parties not using our technology. Slipstream is sufficiently general that it can not only optimize traffic between local applications, but can also be used between Docker containers residing on the same host. Our results show that Slipstream significantly improves throughput and latency, 16-100% faster throughput for server applications (and 100-200% with Docker), while imposing an overhead of around 1-3% when not in use. Overall, Slipstream enables programmers to write simpler code using TCP/IP "everywhere" and yet obtain the significant benefits of faster local transports whenever available.

1 Introduction

TCP has become one of the most commonly used communication protocols because of its ubiquity on standard platforms (e.g., Windows, Android, Linux) and its *location transparency*: instead of creating one communication channel for host-local and another for remote communications, which would reduce portability and increase complexity of the application, developers use TCP because it works for all cases. Unfortunately, by using TCP, developers eschew faster host-local transport mechanisms (e.g., Unix domain sockets, pipes, or shared memory) resulting in missed performance opportunities: a claim supported by a comprehensive study providing clear evidence for the *potential* improvements to be had by replacing the TCP transport with other local IPC mechanisms [27].

Using TCP for its *location transparency* to reduce programming burden and enhance portability is common in several application domains. Web sites are commonly deployed on a single system, yet the Web server front-end communicates with database engines and application servers over TCP, hurting both latency and throughput of Web requests (e.g., LAMP). In some instances, application logic depends on TCP parameters [16], e.g., Memcached uses the IP address and TCP port to implement consistent hashing [10], thereby requiring TCP addressing to function correctly, but *not* necessarily TCP data transport.

Perhaps the most compelling future need for optimizing local TCP communication is the increasing popularity of lightweight virtualized environments like Docker. Docker strongly encourages separating communicating services into distinct Linux containers, using TCP to communicate with each other because portability is a key goal [21]. For example, Yelp developers created a service discovery architecture in which client applications communicate with remote endpoints through a host-local proxy (`haproxy` [1]) via TCP [8]. One of the primary deployment scenarios for this architecture is to run a client application and its local proxy in separate containers, which puts the TCP latency on the critical path for every client request and response. Optimizing local TCP communication over Docker can therefore provide important performance gains (as our experiments with other containerized Docker services show).

In fact, the problem is important enough that there are several approaches that optimize TCP communication between communicating processes within single operating system environments. This includes commercial operating systems like Windows [5], AIX [23] and Solaris [19] and several userspace libraries [2, 25, 29]. However, these approaches either require changes to the operating system [5, 19, 23] or application code [2, 25, 29]—thus eliminating one of the key benefits of using TCP in the first place—or they are only applicable to specific language runtimes. In legacy deployments, it might not be possible to modify the OS or application, and furthermore, in cases where modifications are possible, they may not be feasible: any modifications may cost too much to make to existing application logic.

We present **Slipstream**, a userspace solution that identifies and optimizes the use of TCP between two host-local communicating endpoints without requiring changes to the operating system or applications (except for the use of a shim library above `libc`). Slipstream

transparently reduces latency and increases throughput without requiring modifications to either the kernel or the application by interposing on TCP related events to detect and optimize the communication channel.

We built a Linux prototype of Slipstream that detects TCP-based host-local communications by inserting an optional shim library above `libc` to intercept TCP communication operations.[1] Our solution is portable across UNIX-like systems. Slipstream uses this vantage point to collect information on connections in order to apply a general detection algorithm that relies only on observable characteristics of TCP, without knowledge of the underlying network topology, to detect host-local communication endpoint pairs. Once a host-local TCP communication stream is detected, Slipstream transparently replaces TCP with a faster host-local transport while emulating all TCP operations to maintain application transparency. The primary complexity in the design and implementation of Slipstream arises in replicating kernel-level TCP state at the user-level and preserving the interface semantics of the TCP sockets API on top of the host-local transport.

Our results indicate significant performance benefits for applications using Slipstream: throughput improves up to 16-100% on server applications and 100-200% with Docker, and microbenchmarks show that latency is cut in half. Our results also show that when Slipstream tries but fails to optimize a connection (e.g., because one of the two endpoints is not using Slipstream), the throughput is impacted by only 1-3% on average. Moreover, Slipstream is an opt-in system that imposes *zero* overhead for applications that do not explicitly request the optimizations.

Our work makes the following contributions:

- We describe a novel backwards-compatible, transparent algorithm for classifying communication between two endpoints as host-local or remote.
- We describe a fully automatic optimization to replace TCP with Unix domain sockets as the transport layer, while preserving the interfaces, reliability guarantees, and failure semantics of the original transport.
- We show by use of microbenchmarks and server applications that Slipstream achieves significant improvements in throughput, without requiring manual tuning, custom APIs, or special configuration.

There are certain system configurations that will cause our system to mismatch connections; violations of our correctness conditions, while unlikely in practice, must be avoided during system setup.

[1] The source code for Slipstream and the scripts used in the evaluation are available at: `http://wdtz.org/slipstream`.

Overall, our experience suggests that the improvements achieved automatically by Slipstream are comparable in terms of performance (as we show in the Netperf results) to those that can be achieved by modifying applications to explicitly use Unix domain sockets for host-local connections.

2 Slipstream Overview

Slipstream transparently identifies and dynamically transforms TCP streams between local endpoints into streams that employ more efficient IPC mechanisms, such as Unix domain sockets (UDS). This optimization improves communication performance for many existing applications, and alleviates the burden on programmers to manually detect and select the fastest transport mechanism. To accomplish this task, Slipstream interposes on TCP interactions between the application and the operating system to track TCP endpoints, detect local TCP streams, switch the underlying transport mechanism *on-the-fly*, and emulate TCP functionality on top of the local transport mechanism.

Performing all of these in userspace means that Slipstream must replicate critical stream state at the userspace level and it must adequately emulate TCP on a non-TCP-based transport mechanism. In this section we describe the key design goals we aim to meet, discuss the challenges presented by TCP, and then provide a high level description of Slipstream.

2.1 Design Goals

We specify three key design goals for the optimization, which are desirable for real-world use and present new design challenges. None of the previous systems meet all three requirements. First, we aim to preserve application transparency, i.e., requiring *no changes* to application code in order to perform this replacement. However, application end-users can choose whether or not to enable the optimization. Second, we aim to avoid any operating system changes, not even through dynamically loadable kernel modules, i.e., the optimization should be implemented entirely via userspace code (in fact, we do not even require root privileges). Although aspects of the optimization would be simpler to implement within the OS, those extra challenges are solvable in userspace as well (as we show), and a solution that does not require kernel changes is far easier to deploy. Third, unoptimized communication (i.e., between an optimized component and an unoptimized one) must continue to work correctly and, again, with no application changes.

Of course, the system must also meet essential correctness requirements: the reliable stream delivery guarantees of TCP must be preserved, and the semantics of socket operations must be implemented correctly.

2.2 TCP Optimization Challenges

Within an OS, a *TCP endpoint* is a kernel-level object represented as a 2-tuple, ⟨local IP address, TCP port number⟩. A TCP stream is a pair of *TCP endpoints*, and is also referred to as a *socket pair*. The kernel provides userspace access to TCP via the standard socket interface, which applications use for creating and managing connections and for sending and receiving data. The socket interface represents instances of streams in the form of a socket descriptor, a special case of a file descriptor. These file descriptors are the only way in which userspace code can access TCP endpoints. Since Slipstream is a userspace mechanism but must emulate details of the TCP protocol, it operates by replicating the notion of a TCP endpoint in userspace.

TCP, the POSIX socket interface, and their implementation in modern operating systems present some key design challenges:

- Although a socket pair uniquely identifies TCP connections in a network, a single host can be part of multiple networks with overlapping network name spaces. For example, via use of virtual environments, it is possible to have the same IP address assigned to multiple network interfaces within the same system. This allows duplicate combinations of IP address / TCP port pair to be used for distinct TCP streams within the same host. Each such combination would be a distinct TCP endpoint in the kernel.
- It is common for the kernel to map a single TCP endpoint into multiple process address spaces, thereby creating several userspace file descriptors for a single kernel-level TCP endpoint. This feature is widely used in applications, such as Apache. Consequently, Slipstream must also track all application interactions with the kernel that might create or delete multiple instances of such endpoints.
- The TCP protocol does not support any reliable mechanism for transferring extra data "out-of-band" between the endpoints[2]. On the other hand, injecting any such data into the stream "in-band" would break an application that isn't using Slipstream and so cannot filter out the extra data. This significantly complicates the task of detecting when two endpoints are on the same machine and capable of using Slipstream.
- POSIX file descriptors support a large number of non-trivial functional features through numerous system calls, which must be correctly emulated to preserve application functionality transparently. Some are specific to sockets (e.g., `bind`, `connect`, `listen`, `accept`, `setsockopt`) while others are generic to file descriptors (e.g., `poll`, `dup`, `dup2`, `fcntl`, `fork`, `exec`, and the various send/receive operations). Slipstream supports all of these system calls and other less common ones that interact with TCP.

[2]There is a TCP urgent data feature, but its use to communicate lengthy amounts of data is unreliable.

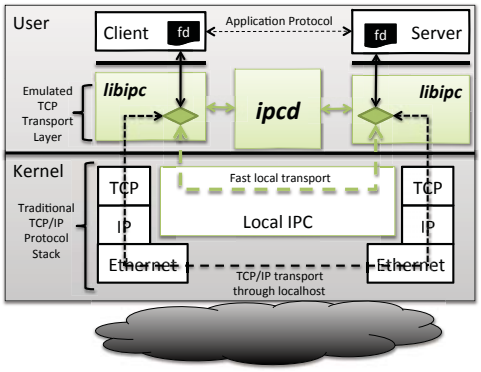


Figure 1: Network layers and local IPC within Slipstream.

2.3 Slipstream Overview

Figure 1 shows a high-level diagram of a typical OS network stack, enhanced with Slipstream. Slipstream inserts a shim library we call `libipc` that interposes on all TCP interactions between the application and the kernel. `libipc` is responsible for tracking TCP endpoints at all TCP-relevant system calls and for reporting all TCP stream identifying information to a system-wide process, `ipcd`. `ipcd` collects and records all stream information and analyzes all existing streams using a stream matching algorithm.

Once Slipstream detects that both endpoints of a stream are local, `libipc` modifies the underlying communication channel to use a local IPC transport (in the case of our implementation, UDS). The use of emulation also indicates one of the major contributions of our efforts: emulating a sufficient subset of the TCP protocol in userspace to correctly support real applications, as demonstrated in our evaluation.

Overall, this sequence of steps ensures that (a) Slipstream can replace TCP with a local IPC transport without requiring any changes to application code; (b) Slipstream does *not* break remote streams or local streams that cannot be identified by Slipstream; and (c) the protocols used by Slipstream *never* introduce new errors in communication for identified local streams. In this sense, the optimizations are *transparent* to application code, are *backwards-compatible* with non-participating endpoints, and do not require kernel modification.

3 Design and Implementation

The functionality of Slipstream has three major aspects: (1) identifying TCP communication streams in which both endpoints are local; (2) replacing TCP with an alternative local transport; and (3) emulating most of the functionality

of the TCP sockets interface. The first two subsections below describe preliminary design aspects for interposing on and tracking TCP events. Subsections 3.3– 3.5 then discuss the three major aspects of the design.

3.1 Interposing on TCP Events

In order to identify local streams and emulate TCP on optimized transport, we monitor applications and track the creation and use of TCP endpoints. We do this using our per-process library, `libipc`, which intercepts all TCP-related calls exposed by the system. In our implementation, we insert `libipc` into a client application by using the `LD_PRELOAD` environment variable or by specifying the full path to `libipc` in `/etc/ld.so.preload`. We prefer dynamic interposition in favor of replacing `libc` so as to avoid requiring modifications to the system libraries and, importantly, to enable applications to choose whether or not to use our technology.

3.2 TCP Endpoints in Userspace

In order to track TCP streams, Slipstream assigns a unique *endpoint identifier* (*EPid*) to each TCP endpoint created and used by the application. Each *EPid* represents an in-kernel TCP endpoint. To manage the optimization state, `libipc` assigns a state to each *EPid* to track its optimization status:

Pre-opt	Optimization not yet attempted.
No-opt	Optimization attempted and failed.
Opt	Optimization successful.

As explained in Section 2.2, in order to fully track streams, Slipstream must replicate endpoint state at the user level (in `libipc`) by tracking TCP state at critical system calls, such as `fork`, as well as traditional TCP modifying operations. For each *EPid*, `libipc` maintains a reference count representing how many processes have at least one file descriptor open to this endpoint. `libipc` updates this reference count on events that affect it, such as `fork`, `exec`, and `exit`. Moreover, instead of communicating with `ipcd` on every use of a socket, as many details as possible about file descriptors and *EPid*s are retained by `libipc`.

In our implementation, `libipc` state information is maintained in two tables, one tracking file descriptors and the other tracking endpoints. The file descriptor table tracks much of what the kernel also tracks at a per-file descriptor granularity, such as the `CLOSE_ON_EXEC` flag or `epoll` state. In addition, this table also tracks the mapping of file descriptors to endpoint identifiers. The *EPid* table tracks the optimization state for each *EPid*, explained above. It also tracks information that is relevant for the optimization procedure, such as the handle for local transport if optimized and running hashes of sent and received data.

3.3 Identifying Host-Local Flows

The first major step of Slipstream is to identify when two endpoints of a TCP stream are located on the same host. As noted in Section 2.2, the combination of IP address and TCP port is insufficient to do so because it is possible to have access to two network domains on a single host, even though this may be rare. Instead, to identify local TCP streams, Slipstream augments the usual IP address and port pairs with extra information passively obtained by watching the initial TCP conversation. This information consists of hashes of the first N bytes of the stream and precise timing of the connection creation calls. Together, these components are sufficient to pair endpoints in the vast majority of situations. When they are not sufficient, Slipstream detects such situations and does not attempt to optimize the socket.

More specifically, the steps Slipstream takes are as follows. When a new TCP socket is connected, `libipc` immediately records the time of the connection attempt and forwards it to `ipcd` along with the IP and port information. `ipcd` uses this information (all but the hashes) to identify endpoints that are likely candidates for pairing. By receiving this information *immediately*, without waiting for the hashes, `ipcd` can eagerly detect if multiple pairings are possible due to overlapping address/port pairs and timing information. After N bytes have been sent in one direction on the stream, `libipc` contacts `ipcd` to attempt to find a matching endpoint. Since the N-byte transfer almost certainly takes significantly longer than reporting the connection information to `ipcd` for reasonable[3] values of N, this ensures that if a mis-pairing is possible *it is detected before the optimization happens*. In this case, the stream is conservatively switched to the No-opt state, and optimization is aborted.

If a single matching endpoint is found, `ipcd` initiates the optimization procedure, explained in Section 3.4. If a matching endpoint is not found, `ipcd` records the current endpoint in a list and waits for a match, while `libipc` tries again several times after which it declares the procedure has failed. In this case, `libipc` changes the state of the *EPid* to the 'No-opt' state. The last request by `libipc` for a matching endpoint is sent with a flag telling `ipcd` to remove the list entry if it is not matched. This removal serves two purposes: first, it helps eliminate matching errors by preventing stale endpoint data from being matched; second, the atomic "request-or-removal" avoids the issue of having only one endpoint aware of a pairing: `ipcd` only pairs endpoints if they have both indicated they will check again for a pair.

[3] In our prototype, we use $N = 2^{16}$.

3.4 Transparent Transport Switching

Once ipcd has determined that two local endpoints are communicating with each other over TCP, ipcd generates a pair of connected sockets using a faster transport (in our implementation, Unix domain sockets) and passes them to the libipc instances controlling the communicating endpoints. Generating the new sockets in ipcd and not libipc allows the procedure to work even when the two libipc instances cannot directly communicate, such as between separate Docker containers. Upon receiving its side of the faster socket, libipc copies appropriate flags from the old socket to the new socket and then changes the state of the *EPid* to 'Opt'.

libipc then migrates communication to the new channel for improved throughput and latency. The primary challenge in doing this correctly is ensuring that both endpoints will switch to the new channel at the same position in the data streams.

To make this switch, libipc ensures that a send or recv request ends at exactly *N* bytes. For a non-blocking send or recv operation on the TCP socket, libipc merely truncates this request, returns a short write and lets the application issue the next request as normal. For blocking operations, in which a short write could be misinterpreted by the application, libipc instead splits the request into two pieces and processes them internally as two requests but only returns control to the application after both requests have been processed. After splitting the request, libipc attempts to optimize the endpoint as detailed above, and subsequently transfers the remaining bits of the request using the selected transport, except that the request is handled in a non-blocking manner to better emulate what would have occurred if the entire request had been processed without libipc intervention.

3.5 Emulating TCP in User-Space

The bulk of the socket API has a straightforward implementation in libipc: whenever a file descriptor that has an underlying endpoint identifier is mentioned, the real API is called with either the optimized transport's file descriptor (if in the 'Opt' state) or the original TCP socket's file descriptor (otherwise). Some system calls, however, require a more complex implementation.

A global connection table is maintained by ipcd to coordinate the matching process. Entries are created in this global table whenever functions like socket and accept request to create a TCP socket and are initialized with properties about the socket such as the local and remote IP address. Removal from the global table only happens when a libipc instance determines that all file descriptors within that process that refer to a single endpoint have been closed. Since it is is possible to share endpoints across multiple processes via use of fork, ipcd maintains a count of the number of processes using a single endpoint identifier.

The basic read and write functions (send, recv, recvmsg, etc.) require more work when the endpoint identifier is in the 'Pre-opt' state, where the tracking of the first *N* bytes and the seamless transition steps described in Section 3.4 need to be executed in addition to the normal I/O.

Emulating fork requires more care due to the introduction of multiple processes that could potentially race on communication to ipcd. This race is resolved by having libipc inform ipcd that all of its endpoint identifiers are about to be duplicated before making the call to fork; should the call to fork fail, libipc tells ipcd to close the duplicated file descriptors. If libipc were to notify ipcd of the endpoint identifier duplication after the call to fork, then a child process that immediately calls close on its copy of the file descriptor would cause ipcd to prematurely clean up the connection.

While fork is emulated by libipc, the implementation is only sufficient to support sharing file descriptors across multiple processes if only one process at a time communicates on it. This level of support is sufficient to support most forking server applications, in which the parent creates a socket, forks, and then closes the socket while the child process does all of the communication on said socket.

The exec family of functions implicitly refer to file descriptors by the need to clean up those that have the CLOSE_ON_EXEC flag, but they also pose a challenge to support since the internal memory, including both the code and data segments of libipc, is completely wiped. libipc retains its memory across the exec call by copying it to a shared memory object tied to a file descriptor that is retained across the exec call. If the new process uses libipc, the initialization process first reads this table and initializes its current state. If the new process does not use libipc, the tie to the file descriptor at least ensures that the eventual death of the process will clean up the system resources allocated by libipc.

4 Discussion

The current design of Slipstream assumes no ability to communicate protocol data using packets within the original stream, i.e., to add reliable "out-of-band" (OOB) signaling between two libipc instances connected in a stream. This has implications for performance, correctness, and security.

4.1 Performance

The implication for performance is that Slipstream may fail to optimize some instances of host-local TCP communication, i.e., we may have *false negatives*. For example, Slipstream may conservatively decide not to

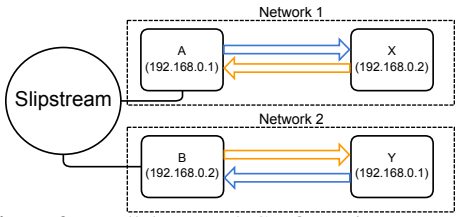


Figure 2: Parallel Network Configuration Example

optimize a stream if it exchanges fewer than N bytes, or if the match is not identified within a short time interval.

More generally, we focus on optimizing using observations external to the TCP stream itself that are readily available on any system. Moreover, we design the protocol conservatively to avoid *false positives*, which represent correctness violations. To avoid such errors, Slipstream must be able to disambiguate TCP streams in fairly complicated scenarios such as multiple identical virtual networks with endpoints having identical conversations.

4.2 Correctness

It is virtually impossible *through an accidental misconfiguration* for Slipstream to incorrectly match and optimize endpoint communication. A "mispairing" (or "false positive" or "incorrect match") can only occur if *all* of the following are true:

1. There are multiple TCP streams described by the same 4-tuple ⟨SrcIP, SrcPort, DstIP, DstPort⟩, and there must be at least one common host system running more than one stream.
2. These streams were established with overlapping timings.
3. The first N bytes of these streams are identical.
4. Slipstream is deployed to some, but not all, of the endpoints described in these streams.

An example of a scenario that would be needed to cause false positives is shown in Figure 2. In this example, A and B are using Slipstream but X and Y are not. The ports used by A and Y (not shown in figure) must be the same, and so must be the ports used by B and X. This scenario would then satisfy condition 1 as the same 4-tuple of the IP addresses and ports identifies both concurrent TCP streams, A−X and B−Y. In addition, these endpoints must establish connections at approximately the same time (within the time window used by a `libipc` instance to poll its associated `ipcd` for a match; not greater than 100ms), and both connections must communicate the same first N bytes of data. Only if all these conditions hold will Slipstream erroneously attempt to pair endpoints A and B.

The need for the first three conditions is obvious from our endpoint matching protocol. The first condition cannot occur within a single well-behaved network; for the same IP address to occur in multiple distinct streams, there must be local endpoints that reside on distinct networks making use of the same IP addresses. Even in such scenarios, the ports used must match as well, which is unlikely because it is very common for clients to use randomly generated port numbers (called *ephemeral ports*) when setting up connections with servers, by using a range of dynamic port numbers set aside for this purpose [12].

Condition 4 is required because if Slipstream is deployed to all endpoints, the possibility of mispairing will be detected before optimization is attempted, and Slipstream will conservatively avoid the pairing.

While these four conditions are possible, they are unlikely to occur accidentally. A well-configured system would not assign identical IP addresses to different interfaces. The use of ephemeral ports, which are drawn randomly from a fairly large range (e.g., 32768-61000 by default on recent Linux kernels, for IPv4) makes condition (1) even more unlikely. The two connections using those two ports must both be started within a very small window of time. Finally, the connections must send exactly the same N bytes of data, for moderately large values of N (e.g., 2^{16}). As a result, Slipstream is well-suited for most real-world applications and is only unsafe when deployed to applications known to intentionally violate one or more conditions (e.g., regularly sending the same first N bytes).

4.3 Security

The key new security risk posed by Slipstream is that an attacker (any unauthorized third party) could try to force a mispairing, i.e., that the attacker is given read or write access to a TCP stream to which she did not have access previously. The threat model we assume is that the attacker must have local access to a machine where either endpoint of some local stream lives (necessary to talk to `ipcd`), *and* does *not* have root privileges on that machine (with root, more powerful attacks are possible even without Slipstream).

In the absence of root access, it is impossible for the attacker to forge IP headers or to misconfigure a second network to obtain duplicate IP addresses as an existing network. In our current *implementation*, however, we trust that each `libipc` is being honest in describing the information about its socket connection. A `libipc` controlled by a local attacker could simply "lie" about its IP address and port number and could potentially construct the remaining information, including the N-byte hash, in order to fool `ipcd` into giving it an optimized endpoint incorrectly. A simple solution is to give `ipcd` sufficient privileges to verify the IP address sent to it. On a well-configured system that is not running a Docker-like environment, this is sufficient to prevent a non-root attacker from impersonating another endpoint.

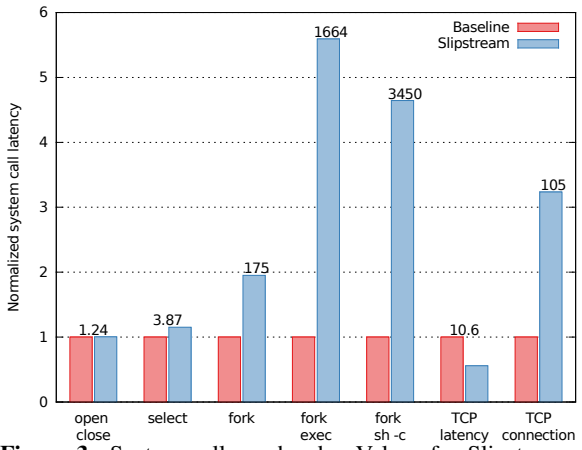


Figure 3: System call overheads. Values for Slipstream in microseconds are shown above the bars. (Lower is better.)

On a system running a Docker-like environment, it is plausible that multiple containers are assigned the same IP address in a virtual network configuration. The default Docker configuration, however, is to assign all containers unique IP addresses from a single virtual network, which prevents condition 1. Restricting Slipstream to use only in the default configuration therefore eliminates any security risk from untrusted containers. We leave it to future work to support non-default Docker configurations with duplicate IP addresses securely. For example, it might be possible to abandon our initial assumption and extend Slipstream to exchange data reliably "out-of-band" but within the TCP stream by building on existing approaches in the literature, e.g., through covert use of various TCP/IP headers [31]. This would add some complexity to `libipc`, but would be justified in large installations (e.g., a data center) in which the one-time cost of enhancing `libipc` would benefit many customers.

5 Results

To evaluate Slipstream, we use a suite of microbenchmarks and applications that measure the performance of various aspects of networking. We have two primary goals in this evaluation. First, we aim to measure the performance impact of Slipstream for network microbenchmarks and for networked applications. Second, to investigate the performance impacts in more detail, we measure the performance *overheads* incurred by common system calls due to the extra bookkeeping necessary for Slipstream.

We perform all of our experiments on a workstation with a 4-core Intel x86-64 processor with 16GB of DDR3-1333 RAM. This workstation runs Ubuntu 14.04 using stock packages, including most notably Linux 3.13.0-36 as the base kernel, Docker 1.0.1, and OpenJDK 1.7.0_75 for the Java VM. All of the networking configuration parameters for the Linux kernel have been left set to their default values.

5.1 Microbenchmarks

We use the NetPIPE and Netperf microbenchmarks to measure total networking throughput under different networking communication patterns. We use two variants of NetPIPE, one in C and one in Java, to show the impact for two different programming languages; in fact, Slipstream is able to optimize Java code running on OpenJDK as transparently as it does for C code, as explained below. Another microbenchmark, lmbench [20], measures the overhead of Slipstream on common system calls.

5.1.1 lmbench

lmbench [20] is a microbenchmark that measures the overhead of various system calls, which gives an indication of the penalty incurred by using Slipstream in code that may not benefit from its improvements. Selected results are shown in Figure 3; the other numbers are omitted because they are unaffected by Slipstream.

Due to the tracking of file descriptors within userspace, Slipstream naturally adds a small amount of overhead to most system calls that interact with file descriptors. For example, an `open` and `close` pair is 5% slower with Slipstream, while a `select` over 100 socket descriptors is 15% slower.

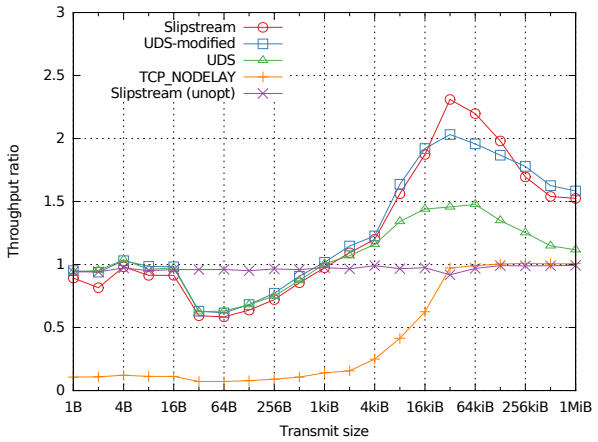
The tracking of file descriptors imposes the largest overheads on `fork` or `exec`. For `fork`, this is due in large part to synchronous communication between `libipc` and `ipcd` that blocks the actual system call. In contrast, the overhead in `exec` is due to loading `libipc` in the memory space of the new process, as well as the overhead of setting up the shared memory object to retain `libipc` state across the call (see Section 3.5).

TCP latency, when the connections have been optimized, is brought down to the same latency as UDS: about 10 microseconds, about half the original TCP latency. However, the initial connection latency is greatly increased due to our need to register the new connection to `ipcd`.

5.1.2 Netperf

Netperf [15] is a microbenchmark to measure total throughput of network connections. Netperf sends data unidirectionally, creating a new socket for each transfer size. The sizes transferred are chosen in logarithmic fashion. Results are presented in Figure 4.

At certain smaller buffer sizes (e.g., 32B-256B), both Slipstream and UDS perform roughly 25-50% worse than the TCP baseline in terms of total throughput. This effect is due to synchronization overhead inside the Linux kernel, which is not observed by the baseline because `TCP_NODELAY` is disabled and TCP buffer coalescing occurs. To validate this, we also compared TCP



Experiment	1 B	32 B	1 KiB	32 KiB	1 MiB
Baseline	2.06	98.28	1625.83	3891.92	5193.10
TCP_NODELAY	0.22	7.11	228.34	3785.77	5232.23
Slipstream	1.84	58.30	1585.19	8988.88	7917.70
Slipstream (unopt)	1.94	94.32	1585.71	3576.18	5159.05
UDS	1.94	61.36	1636.47	5670.26	5801.53
UDS (modified)	1.96	61.93	1655.32	7905.42	8226.89

Figure 4: Throughput as measured by Netperf, with a baseline of TCP without Slipstream or `TCP_NODELAY` specified. The table contains a subset of the throughput results, measured in MB/s.

performance with `TCP_NODELAY` enabled; that curve clearly shows the negative impact of eliminating TCP buffering. At higher data sizes, the relative overhead of the synchronization vs. data transfer is reduced, and Slipstream and UDS sockets both perform better than the baseline, mimicking the results for other benchmarks.

Surprisingly, we observed an increase in throughput with Slipstream compared to using UDS for some of the larger data sizes. On further investigation, we found that this extra speed is primarily due to Netperf using a very small socket receive buffer size (2304 bytes) for the UDS tests. When we changed the Netperf code to not set a socket buffer size (labeled "UDS-modified" in the graph), the apparent effect largely disappears.

Finally, to measure the impact of using Slipstream when optimization is not possible, we ran Netperf using Slipstream only on the client. As shown in the figure, performance was generally very close to that without using Slipstream, on average 3.5% slower than baseline.

5.1.3 NetPIPE

NetPIPE [28] is another microbenchmark that measures the throughput of TCP. NetPIPE differs from Netperf in that it transfers its data bidirectionally and that it reuses the same socket for all of the transfer sizes.

Both variants of NetPIPE use the same basic networking structure, in which the client socket sends data of a given buffer size that the server receives, and then the server sends back that data; the process repeats until sufficient measurements are taken to reliably

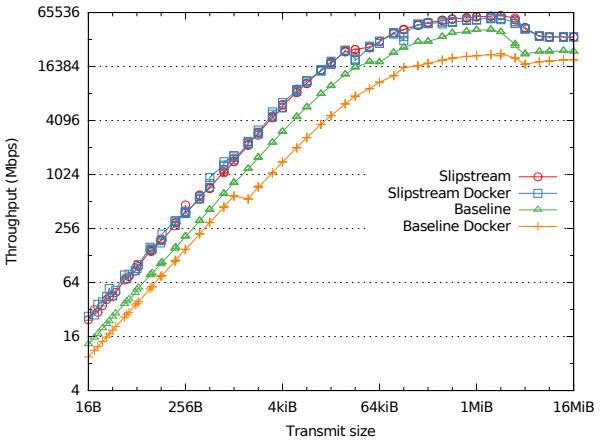


Figure 5: NetPIPE-C performance both with and without Docker containers.

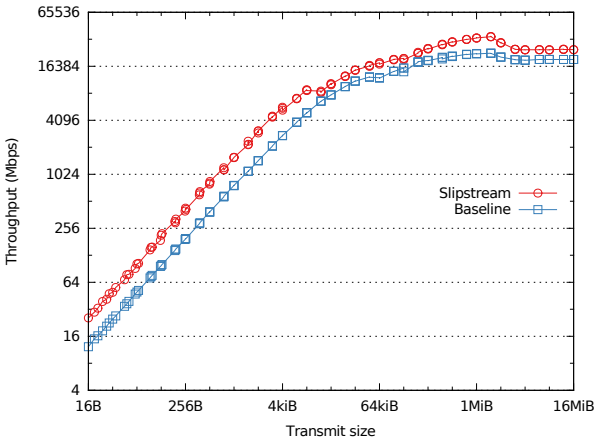


Figure 6: NetPIPE-Java, run only outside of Docker.

estimate the throughput. They also both use the idiomatic synchronized socket functions for the language—`send` and `recv` in C, and `java.io.InputStream` and `java.io.OutputStream` in Java.

The results of running NetPIPE-C are presented in Figure 5, and the results for NetPIPE-Java are presented in Figure 6. For sizes less than about 64KB, Slipstream is able to consistently achieve around 70-150% more throughput compared to baseline TCP. At higher sizes, Slipstream does not provide as much improvement, but is still able to produce at least a 40% increase in throughput.

We also observe that Slipstream optimizes Java networking performance transparently, with no changes to the JVM or the application. Slipstream is able to achieve this because it interposes on `libc`, which is also used by OpenJDK, providing essentially the same benefits to Java programs as to C programs.

5.2 Application Benchmarks

Memcached and PostgreSQL are two example applications that are sometimes used in ways that put the client and server on the same host. We evaluate

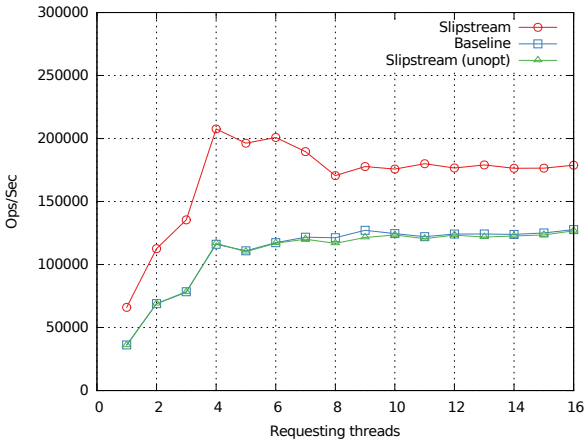


Figure 7: Memcached throughput on host system

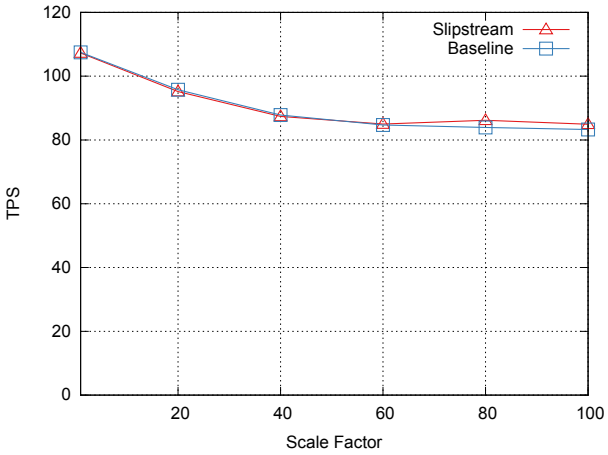


Figure 8: pgbench transactions per second, TPC-B

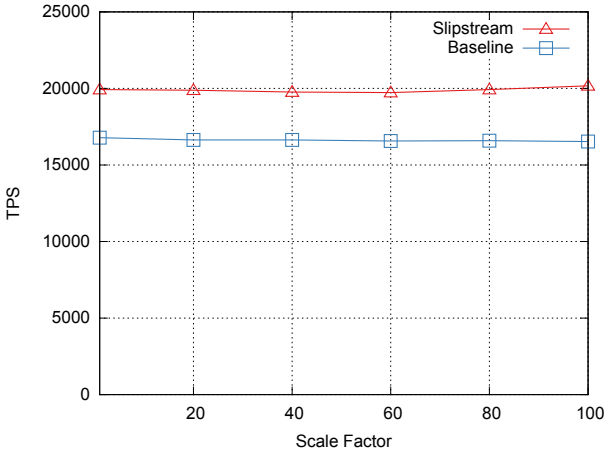


Figure 9: pgbench transactions per second, Select

these applications with representative workloads to (a) demonstrate that Slipstream is indeed fully transparent for important real applications; and (b) to determine the impact of improving TCP performance for real applications. In addition to measuring performance benchmarks, we used Slipstream on a set of applications—ZeroMQ, OpenSSH, Jenkins (Java), Apache, iperf, simple python TCP client/server, nepim—to informally evaluate compatibility when local to remote TCP communications operate through Slipstream: all of these functioned correctly, and all except OpenSSH were successfully optimized. OpenSSH writes to a socket from multiple processes in a way that we do not currently support in our implementation.

5.2.1 Memcached

Memcached [10] is a distributed, in-memory key-value store that is primarily intended to be used to cache database queries for web applications. While Memcached can be configured to listen on Unix domain sockets instead of TCP, feature requests to allow it to listen on both have been rejected since the particulars of the socket it listens on are used in the distributed hash function [16].

For testing, we run Memcached using a single server and 2GB of storage. Queries are executed against a pool of 10000 items each between 1 byte and 4KB in size. The number of connections is varied, and the average number of operations executed per second is observed. Results are presented in Figure 7. Slipstream provides a 25%-to-100% improvement in throughput, which would be a substantial benefit for small Web sites where most of the traffic is local. When using Slipstream on the client but not the server, we measured on average 1-3% slowdown.

5.2.2 pgbench

pgbench [13] is a benchmark for PostgreSQL, a widely used open-source relational database. We run pgbench with two separate workloads, one based on the industry-standard TPC-B benchmark and the other based on a `SELECT`-only benchmark, which spends more time in communication. Slipstream successfully optimizes all TCP communication in both cases. A benchmark scale factor, N, creates $100000N$ rows in the database or about $N \cdot 16$ MB of total database size [26]. Figures 8 and 9 show the results (in database transactions per second, or TPS). Each data point represents the average of multiple runs; the variance observed was negligible.

The TPC-B workload shows little improvement, which is not surprising because the workload is designed to stress the database's internals (primarily disk access) [30]: communication changes have little impact.

In contrast, the `SELECT` workload shows 16-23% improvement in database throughput for all scale factors. This workload performs simple queries that are processed by the database at a much higher rate and, as a result, benefits significantly from communication optimization.

5.3 Docker

Slipstream is able to detect and optimize local TCP traffic within a single network (i.e., to localhost), but also across virtualized networks executing on the same machine such as those created by Docker. To demonstrate

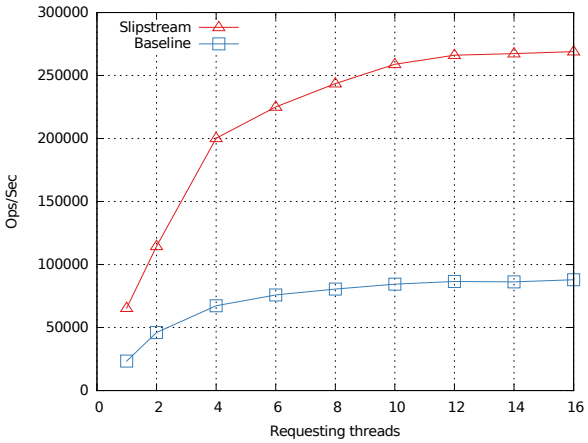


Figure 10: Memcached throughput with Docker

this functionality, and evaluate the performance characteristics of Slipstream in this environment, we conducted additional experiments across Docker containers on the same host. Rerunning our experiments within the same Docker container do not produce different results than by running them on a regular Linux setup.

As different Docker containers have distinct filesystems from the host system, using Slipstream from within a container requires an extra configuration step. If `ipcd` is installed into a Docker container of its own, then the directory containing the UDS that `ipcd` listens on can be also installed on other containers by leveraging the volumes feature of Docker. Alternatively, it is also possible to bind this directory from the host system to a Docker container. Either way, using Slipstream with Docker only requires ensuring that `libipc` is installed within the container and adding a single flag when running the container. *This flexibility is a key benefit of the routing-oblivious design of Slipstream.*

5.3.1 Docker Microbenchmark

We reran our NetPIPE-C microbenchmark using Docker to illustrate the basic performance of Docker networking, also shown in Figure 5. The graph shows that Slipstream is not slowed down when used across Docker containers, whereas normal TCP is, magnifying the TCP throughput improvement to between 150% and 350%. Since Docker containers use separate networking namespaces, the kernel layers need to swap packets between different interfaces, which imposes an extra overhead on TCP transfer, layers which are bypassed by Slipstream. Thus Slipstream's benefits are only enhanced when Docker is in use.

5.3.2 Docker Application Benchmark

In addition to the basic TCP throughput benchmark, we also evaluate the performance of Memcached across Docker containers on the same host. This experiment is identical to the Memcached experiment without containers, with the sole difference that the server and client live in separate containers. These results are shown in Figure 10. Comparing with Figure 7, we again see substantially greater improvements due to Slipstream with Docker than without, ranging from about 100% to 200% speedup. These are very large *application-level* improvements, showing that Slipstream can be a valuable and transparent way to improve the overall performance of services that use Docker containers.

6 Related Work

A number of previous systems aim to optimize local interprocess communication. A brief summary and a feature comparison are presented in Table 1, and are discussed in more detail below.

In-Kernel Solutions Recently, operating systems such as Windows [5], AIX [23], and Solaris [19], have made available localhost TCP optimizations. In general, they all bypass the lower levels of the kernel networking stack, only forming TCP and not IP packets. Performing these optimizations within the existing networking stack simplifies identifying local-only traffic and provides a fast-path for local streams.

Unfortunately, performing these optimizations within the OS has several drawbacks: the implementations are kernel-specific and other systems cannot benefit (e.g., Linux does not support it, although there have been efforts to add it [7]); OS upgrades are often slow to be adopted widely; and applications have no control over whether or not the optimization is available on a given system. In contrast, Slipstream is relatively easy to port, at least across Unix-like systems; it is easy to deploy (e.g., it does not even require superuser privileges to install); and application developers that choose to do so can incorporate the system fairly easily.

VMM solutions for inter-VM communication: Several approaches to improving performance of communication between co-located virtual machines have been described [17, 34, 35], all focusing on Xen. These solve similar communication inefficiencies as Slipstream, but either require application modification [35], guest kernel modification [17, 34, 35], are not fully automatic [17, 35], or operate at the IP layer so TCP overheads are not eliminated [34].

Language-Specific Solutions: The interfaces provided by languages for IPC are often at a much higher level than the basic operations provided by the system. For example, Java Fast Sockets [29] is able to greatly improve communication of Java applications with techniques such as avoiding costly serialization in situations in which the data can be passed through shared memory. While these optimizations are difficult with a language-agnostic solution like Slipstream, Slipstream is able to

Category	Prior Work	Application Transparency[1]	OS Transparency[2]	Sockets[3]	Misc.
OS Impl.	Win. FastPath [5]	✗ Opt-in	✗ Included in OS already	✗	
	Solaris TCP Fusion [19]	✓ Opt-Out	✗ Included in OS already	✓	
	AIX fastlo [23]	✗ Opt-in	✗ Included in OS already	✓	
	Linux TCP Friends [7]	?	✗ Floating Kernel patch	✓	
VM-VM	XWay [17]	✓	✗ Guest kernel patch		
	XenSocket [35]	✗ `AF_XEN`	–	✓	
	XenLoop [34]	✓	✗ Guest kernel module		
User. Stack	mTCP [14]	✗ Similar API	✗ NIC driver for packet library	✗	
	Sandstorm [18]	✗ Specialized for App.	✗ requires netmap [24] kernel support	✗	
User. Shim	Fable [27]	✓ Limited, ns-based	✗ System call for name service	✓	
	Java Fast Sockets [29]	✓ Java.net.Socket	✓	✗	Java-Only
	Universal Fast Sockets [2]	✓	✓	✓	Commercial
	FastSockets [25]	✓	✓ Uses Active Messages	✓	
	Slipstream	✓	✓	✓	

[1] Application Transparency (no application modifications required)
[2] OS Transparency (no OS modifications required, no use of OS-specific tech)
[3] Provides Socket Interface (POSIX, Linux)

Table 1: Prior Work: Categories and Features

optimize applications that use sockets regardless of source language, as our results illustrate for C/C++ and Java.

Transparent Userspace Libraries: The FABLE library, which is only described in a position paper [27], provides automatic transport selection and dynamic reconfiguration of the I/O channel. FABLE provides a socket compatibility layer that uses a new system call for looking up a name mapping (implying that FABLE is not a pure userspace solution) to identify communication with hosts for which it may be able to provide a more efficient transport. Without any information about an implementation, it is unclear how well this compatibility layer works. However, the dynamic switching of transports in Slipstream is very similar to their reconfigurable I/O channels.

Fast Sockets [25] is a userspace library that provides a sockets interface on top of Active Messages [33]. This is superficially similar to Slipstream, but Fast Sockets assumes it can determine which transport to use by inspecting the address, which requires static configuration and a rigid network topology. In contrast, Slipstream focuses on automatic detection of inefficient communication without relying on network topology details and switches transports on-the-fly.

Universal Fast Sockets (UFS)[2] is a commercial solution to optimize local communication transparently. Like Slipstream, UFS uses a shared userspace library to interpose on application activity, but other details of how it operates are proprietary and unclear.

Explicit Userspace Solutions Many software libraries provide *explicit* messaging abstractions for application use [3, 6, 32]. Without the limitations of existing interfaces, impressive performance results are possible, but applications need to be modified to use these frameworks, and seeing the best results may require deep changes to the fundamental structure of an application.

Several researchers have explored moving the network stack out of the operating system and entirely into userspace, citing many performance benefits [14, 22, 24] Userspace networks stacks are components of popular OS designs including the microkernel [11] and exokernel [9] approaches. Some work goes further and collapses the entire network stack into the application [18], providing a specialized stack entirely in userspace. More recent work refactors the OS network stack into a control plane (which stays in the kernel) and separate data planes (which run in protected, library-based operating systems) [4]. All of these efforts redesign the networking stack from the ground up and require kernel modification, application modification, or both. These solutions do not directly aim to optimize local communication, but similar to the in-kernel approaches described above this could likely be added in a straightforward if not portable manner.

7 Acknowledgements

This work was supported in part by the Office of Naval Research grant numbers N00014-12-1-0552 and N00014-4-1-0525, and by the AFOSR under MURI award FA9550-09-1-0539.

8 Conclusion

Slipstream is a novel system for the optimization of TCP communication that requires neither OS nor application modification, which allows it to be easily and rapidly deployed. Our evaluations show that our system is capable of achieving significant performance benefits, at least 16% more throughput than TCP, and up to 200% if Docker is involved, both on real applications in real usage scenarios. Slipstream's minimal assumptions allow it to be used in a variety of network topologies and to use a variety of faster local transports, capabilities we plan to explore in future work. We believe that Slipstream provides an excellent base for reducing the overhead of IPC in applications that is usable across a wide variety of applications and setups.

References

[1] HAProxy. http://www.haproxy.org.

[2] Universal Fast Sockets. http://torusware.com/product/universal-fast-sockets-ufs/.

[3] ZeroMQ. http://zeromq.org.

[4] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 49–65.

[5] BRIGGS, E. Fast TCP loopback performance and low latency with Windows Server 2012 TCP Loopback Fast Path. http://blogs.technet.com/b/wincat/archive/2012/12/05/fast-tcp-loopback-performance-and-low-latency-with-windows-server-2012-tcp-loopback-fast-path.aspx, Dec. 2012.

[6] CAMERON, D., AND REGNIER, G. *The virtual interface architecture*. Intel Pr, 2002.

[7] CORBET, J. TCP friends. http://lwn.net/Articles/511254/, Aug. 2012.

[8] DORAN, T. Building a smarter application stack, 2014. Presented at DockerCon.

[9] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[10] FITZPATRICK, B. Distributed caching with Memcached. *Linux journal 2004*, 124 (2004), 5.

[11] GOLUB, D. B., JULIN, D. P., RASHID, R. F., DRAVES, R. P., DEAN, R. W., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., AND BOHMAN, D. Microkernel operating system architecture and Mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (1992), pp. 11–30.

[12] INTERNET ASSIGNED NUMBERS AUTHORITY. Service Name and Transport Protocol Port Number Registry, May 2015. (Date last updated.).

[13] ISHII, T. pgbench. http://www.postgresql.org/docs/devel/static/pgbench.html.

[14] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 489–502.

[15] JONES, R. The Netperf benchmark. http://www.netperf.org.

[16] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[17] KIM, K., KIM, C., JUNG, S.-I., SHIN, H.-S., AND KIM, J.-S. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 11–20.

[18] MARINOS, I., WATSON, R. N. M., AND HANDLEY, M. Network stack specialization for performance. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks - HotNets-XII* (2013), 1–7.

[19] MAURO, J., AND MCDOUGALL, R. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[20] MCVOY, L., AND STAELIN, C. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), ATEC '96, USENIX Association, pp. 23–23.

[21] MERKEL, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[22] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 337–350.

[23] QUINTERO, D., CHABROLLES, S., CHEN, C., DHANDAPANI, M., HOLLOWAY, T., JADHAV, C., KIM, S., KURIAN, S., RAJ, B., RESENDE, R., ET AL. *IBM Power Systems Performance Guide: Implementing and Optimizing*. IBM Redbooks, 2013.

[24] RIZZO, L. netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 9–9.

[25] RODRIGUES, S. H., ANDERSON, T. E., AND CULLER, D. E. High-performance local area communication with Fast Sockets. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1997), ATEC '97, USENIX Association, pp. 20–20.

[26] SMITH, G. Introduction to pgbench. http://www.westnet.com/~gsmith/content/postgresql/pgbench-intro.pdf, 2009. Presented at PG East.

[27] SMITH, S., MADHAVAPEDDY, A., SMOWTON, C., SCHWARZKOPF, M., MORTIER, R., WATSON, R. M., AND HAND, S. The case for reconfigurable I/O channels. In *RESoLVE workshop at ASPLOS* (2012), vol. 12.

[28] SNELL, Q. O., MIKLER, A. R., AND GUSTAFSON, J. L. NetPIPE: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems* (1996), vol. 6, Washington, DC, USA).

[29] TABOADA, G. L., TOURIÑO, J., AND DOALLO, R. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun. 31*, 17 (Nov. 2008), 4049–4059.

[30] TÖZÜN, P., PANDIS, I., KAYNAK, C., JEVDJIC, D., AND AILAMAKI, A. From a to e: Analyzing TPC's OLTP benchmarks: The obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology* (New York, NY, USA, 2013), EDBT '13, ACM, pp. 17–28.

[31] VASSERMAN, E. Y., HOPPER, N., AND TYRA, J. SilentKnock: practical, provably undetectable authentication. *International Journal of Information Security 8*, 2 (2009), 121–135.

[32] VINOSKI, S. Advanced message queuing protocol. *IEEE Internet Computing 10*, 6 (Nov. 2006), 87–89.

[33] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1992), ISCA '92, ACM, pp. 256–266.

[34] WANG, J., WRIGHT, K.-L., AND GOPALAN, K. XenLoop: A transparent high performance inter-VM network loopback. *Cluster Computing 12*, 2 (June 2009), 141–152.

[35] ZHANG, X., MCINTOSH, S., ROHATGI, P., AND GRIFFIN, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware* (New York, NY, USA, 2007), Middleware '07, Springer-Verlag New York, Inc., pp. 184–203.

FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency

Jihyung Lee, Sungryoul Lee*, Junghee Lee*, Yung Yi, KyoungSoo Park

Department of Electrical Engineering, KAIST
The Attached Institute of ETRI*

Abstract

Network packet capture performs essential functions in network management such as attack analysis, network troubleshooting, and performance debugging. As the network edge bandwidth exceeds 10 Gbps, the demand for scalable packet capture and retrieval is rapidly increasing. However, existing software-based packet capture systems neither provide high performance nor support flow-level indexing for fast query response. This would either prevent important packets from being stored or make it too slow to retrieve relevant flows.

In this paper, we present FloSIS, a highly scalable, software-based flow storing and indexing system. FloSIS is characterized as the following three aspects. First, it exercises full parallelism in multiple CPU cores and disks at all stages of packet processing. Second, it constructs two-stage flow-level indexes, which helps minimize expensive disk access for user queries. It also stores the packets in the same flow at a contiguous disk location, which maximizes disk read throughput. Third, we optimize storage usage by flow-level content deduplication at real time. Our evaluation shows that FloSIS on a dual octa-core CPU machine with 24 HDDs achieves 30 Gbps of zero-drop performance with real traffic, consuming only 0.25% of the space for indexing.

1 Introduction

Network traffic capture plays a critical role in attack forensics and troubleshooting of abnormal network behaviors. Network administrators often resort to packet capture tools such as tcpdump [6], wireshark [8], netsniffng [3] to dump all incoming packets and to analyze the behaviors from multiple dimensions. These tools can help pinpoint a problem in network configuration and performance and even reconstruct the entire trace of an intrusion attempt by malicious hackers.

As the edge network bandwidth upgrades to beyond 10 Gbps, existing packet capture systems have exposed a few fundamental limitations. First, they exhibit poor performance in packet capture and dumping, sometimes unable to catch up with packet transmission rates in a high-speed network. This is mainly because these tools do not properly exploit the parallelism with multiple CPU cores and disks, and the existing kernel is not optimized for high packet rates. Second, most existing tools do not support indexing of stored packets. Lack of indexing would incur a long delay to retrieve the content of interest since the search performance would be limited by sequential disk access throughput, often translating to hours of delay in a multi-TB disk. Third, the huge traffic volume at a high-speed link would significantly limit the monitoring period. For example, it takes less than 17 hours to fill up 24 disks of 3TB at the rate of 10 Gbps. For extended monitoring period, one must enhance the storage efficiency by compressing the stored content.

Recent development of high-performance packet I/O libraries [1, 22, 26, 28] has in part alleviated some of the problems. For instance, n2disk10g [17] and tcpdump-netmap [7] exploit a scalable packet I/O library to achieve a packet capture performance of multi-10 Gbps. Moreover, n2disk10g allows parallel packet dumping to disk and packet-level indexing for fast access. However, the primary problem with these solutions is that they still deal with *packets* instead of network *flows*. Working with packets presents a few fundamental performance issues. First, query processing would be inefficient. Most content-level queries would inspect the packets in the same TCP flow rather than those that belong to completely unrelated flows. Time-ordered packet dumping would scatter the packets in the same flow across a disk. Even with indexing, gathering all relevant packets for a query could either increase disk seeks or waste disk bandwidth from reading unrelated packets nearby the targets. Second, per-packet indexing would be more expensive than per-flow indexing. Per-packet indexing would not only use more metadata disk space but also significantly increase the search time.

In this paper, we present FloSIS (Flow-aware Storage and Indexing System), a highly scalable software-based network traffic capture system that supports efficient flow-level indexing for fast query response. FloSIS is characterized by three design choices. First, it achieves high performance packet capture and disk writing by exercising full parallelism in computing resources such as network cards, CPU cores, memory, and hard disks. It adopts the PacketShader I/O Engine (PSIO) [22] for scalable packet capture, and performs parallel disk write for high-throughput flow dumping. Towards high zero-drop performance, it strives to minimize the fluctuation of packet processing latency. Second, FloSIS generates two-stage flow-level indexes in real time to reduce the query response time. Our indexing utilizes Bloom filters [13] and sorted arrays to quickly reduce the search space of a query. Also, it is designed to consume small amount of memory while it allows flexible queries with wildcards, ranges of connection tuples, and flow arrival times. Third, FloSIS supports flow-level content deduplication in real time for storage savings. Even with deduplication, the system preserves the packet arrival time and packet-level headers to provide exact timing and size information. For an HTTP connection, FloSIS parses the HTTP response header and body to maximize the hit rate of deduplication for HTTP objects.

We find that our design choice brings enormous performance benefits. On a server machine with dual octa-core CPUs, four 10 Gbps network interfaces, and 24 SATA disks, FloSIS achieves up to 30 Gbps for packet capture and disk writing without packet drop. Its indexes take up only 0.25% of the stored content while avoiding slow linear disk search and redundant disk access. On a machine with 24 hard disks of 3 TB, this translates into 180 GB for 72 TB total disk space, which could be managed entirely in memory or stored into solid state disks for fast random access. Finally, FloSIS deduplicates 34.5% of the storage space for 67 GB of a real traffic trace only with 256 MB of extra memory consumption for a deduplication table. In terms of performance, it achieves about 15 Gbps zero-drop throughput with real-time flow deduplication.

We believe that our key techniques in producing high scalability and high zero-drop performance are applicable to other high-performance flow processing systems like L7 protocol analyzers, intrusion detection systems, and firewalls. We show how one should allocate various computing resources for high performance scalability. Also, our efforts for maintaining minimal processing variance should be valuable in high-speed network environments where packet transmission rates vary over time.

2 Design Goals and Overall Architecture

In this section, we highlight our system design goals and describe overall system architecture of FloSIS.

2.1 Design Goals

(1) High scalability: A high-speed network traffic capture system should acquire tens of millions of packets per second from multiple 10G network interfaces, and write them to many hard disks without creating a hotspot. To achieve high performance, the system requires a highly scalable architecture that enhances the overall performance by utilizing multiple CPU cores and hard disks in parallel. Moreover, it should be easily configurable.

(2) High zero-drop performance: The system should ensure high zero-drop performance. A peak zero-drop performance refers to the maximum throughput that the system can achieve without any packet drop. It is an important performance metric of a network traffic capture system since we need to minimize packet drop for accurate network monitoring and attack forensics. It typically differs from the peak throughput that allows packet drop. Even if the input traffic rate is much lower than the peak throughput, a temporary spike of delay caused by blocking I/O calls or scheduling can incur packet drop regardless of the amount of available computation resources.

(3) Flow-level packet processing: Indexing plays a critical role in fast query response. Instead of packet-level indexing [17, 19], we make an index per each flow. Flow-level indexing significantly reduces the index space overhead and improves disk access efficiency. However, it requires managing the packets by their flows so that the packets in the same flow are written to the same disk location. While this incurs extra CPU and memory overheads, it also provides an opportunity to deduplicate the flow content, making more efficient use of the storage.

(4) Flow-aware load balancing: For efficient flow processing, the packets in the same flow need to be transferred to the same CPU core from the NICs while the per-core workload should be balanced on a multicore system. Otherwise, packets should be moved across the CPU cores for flow management, which might cause severe lock contention and degrade CPU cache efficiency.

2.2 Overall Architecture

The basic operation of FloSIS is simple. It captures the network packets mirrored by one or a few switch ports, manages them by their flows [1], and writes them to disk. It also generates per-flow metadata and its associated index entries, and responds to user queries that find a set of matching flows on disk. In case a flow content is redundant, the system deduplicates the flow by having its on-disk metadata point to a version that exists in a disk.

For scalable operation, FloSIS adopts a parallelized pipelined architecture that effectively exploits the parallelism in modern computing resources, and minimizes

[1] We mainly focus on TCP flows in this paper, and non-TCP packets are written to disk as they arrive without flow-level indexing.

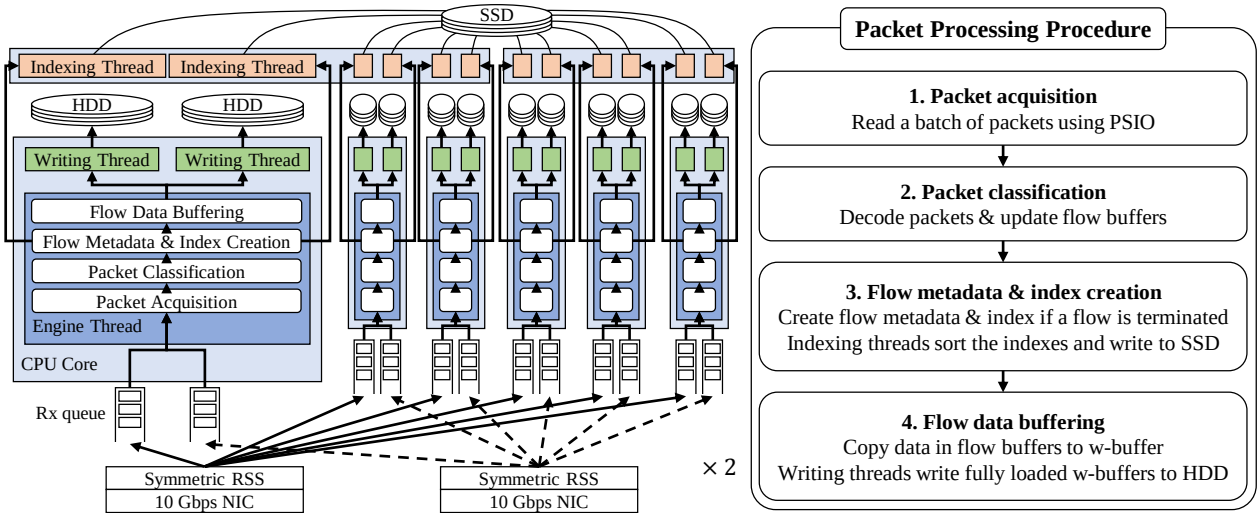


Figure 1: The half of overall architecture and thread arrangement of FloSIS on a machine with 16 CPU cores, four 10G NIC ports, 24 HDDs, and 2 SSDs

the contention among them. It is multi-threaded with three distinct thread types: engine, writing, and indexing threads. The engine thread is responsible for packet acquisition from NICs, flow management, and index generation. It also coordinates flow and index writing to disk with writing and indexing threads. The writing thread shares the buffers for writing with an engine thread, and periodically writes them onto a disk. All packets in the same flow are written consecutively to disk unless they are deduplicated. The indexing thread shares the flow metadata and indexes with an engine thread, and sorts the indexes when it gathers enough entries. Also, it writes the metadata and indexes to disk for durability and helps with resolving the queries by searching through the indexes. We use hard disk drives (HDDs) for storing the flow data and solid state drives (SSDs) for metadata and index information. Our system benefits from cost-effective packet data dumping and prompt query response from fast retrieval of flow metadata and indexes.

Figure 1 shows the mapping of FloSIS threads into a server machine with 16 CPU cores, four 10G NIC ports, and 24 HDDs and 2 SSDs, which we use as a reference platform in this paper. The guiding principle in resource mapping is to parallelize each thread type as much as possible to maximize the throughput while minimizing packet drop from resource contention. Every thread is affinitized to a CPU core to avoid undesirable interference from inter-core thread migration. We co-locate multiple writing threads with an engine thread on the same CPU core since writing threads rarely consume CPU cycles and mostly perform blocking disk operations. Since a writing thread shares the buffers that its engine thread fills in, it would benefit from shared CPU cache if it is co-located with an engine thread. In contrast, an indexing thread runs on a different CPU core since it periodically executes CPU-intensive operations, which might interfere with high-speed packet acquisition with engine threads. Since indexing thread operations are not time-critical, we can place multiple indexing threads on the same CPU core. Each writing thread has its own disk to benefit from sequential disk I/O without concurrent access by other threads. However, SSDs are shared by multiple indexing threads since they do not suffer from performance degradation by concurrent random access.

3 High-speed Flow Dumping

In this section, we describe scalable packet capture and flow dumping of FloSIS. FloSIS uses a fast user-level packet capture library that exploits multiple CPU cores, and performs direct I/O to bypass redundant memory buffering at disk writing while minimizing CPU and memory consumption for disk I/O.

3.1 High-speed Packet Acquisition

The first task of FloSIS is to read packets from NICs. It is of significant importance to allocate enough resource on the packet acquisition, because its performance serves as an upper-bound of the achievable system throughput. Thus, we allocate so many CPU cores as to read the packets from NICs at line rate. For scalable packet I/O, FloSIS employs the PSIO library [22], which allows parallel packet acquisition by flow-aware distribution of incoming packets across available CPU cores. PSIO is known to achieve a line rate regardless of packet size with batch processing and efficient memory management [22]. Also, we use symmetric receive-side scaling (S-RSS) [31] that maps both upstream and downstream packets in the same TCP connection to the same CPU core. S-RSS hashes the 4-tuple of a TCP packet to place the packets in the same connection in the same RX queue inside a NIC. Each RX

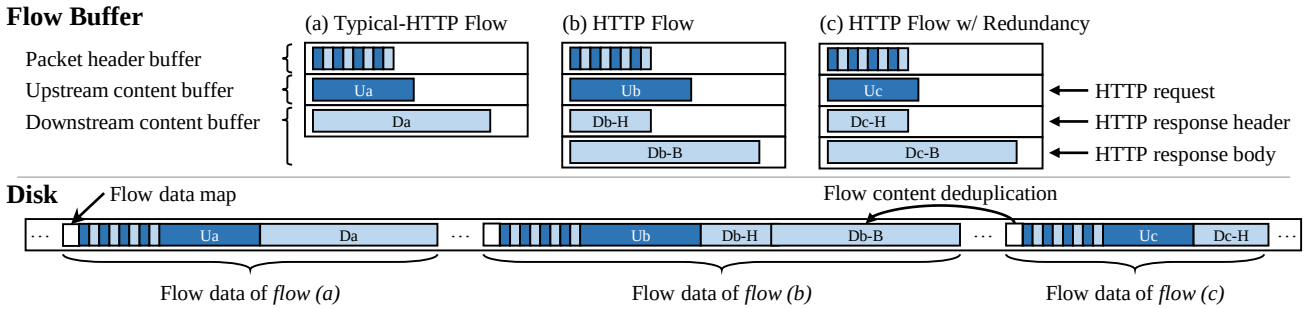


Figure 2: Flow data in the flow buffers and the disk (U: Upstream, D: Downstream, H: Head, B: Body). (a) a typical TCP flow, (b) an HTTP flow with a request in an upstream buffer, and a response in a downstream buffer, and (c) another HTTP flow with the same response body as (b)'s.

queue in a NIC is dedicated to one of the engine threads, and all packets in the queue are processed exclusively by the thread. This architecture eliminates the need of inter-thread synchronization and achieves high scalability without sharing any state across engine threads.

One may have concern that S-RSS may not distribute the packet load evenly across multiple CPU cores. However, a recent study reveals that S-RSS effectively distributes real traffic at a 10 Gbps link almost evenly across multiple CPU cores, by curbing the maximum load difference at 0.2% for a 16-core machine [31].

3.2 Flow-aware Packet Buffering

When packets are read from NIC RX queues, the engine thread manages them by their *flows*. The engine thread classifies the received packets into individual flows, and each flow maintains its current TCP state as well as its packet content. FloSIS keeps all packets in the same TCP connection at a single conceptual buffer called *flow buffer*. A flow buffer consists of three data buffers: packet header, upstream and downstream buffers as shown in Figure 2. The packet header buffer collects all packet headers (*e.g.*, Ethernet and TCP/IP headers) and their timestamps in the order of their arrival. While FloSIS focuses on flow-level indexing and querying, it also supports packet-level operations to provide information such as packet retransmission, out-of-order packet delivery, inter-packet delay, etc, where packet header buffers are needed to reconstruct such information. The upstream and downstream buffers maintain reassembled content of TCP segments so that the user can check the entire message at one disk seek. In case of an HTTP flow, the downstream buffer is further divided into response header and body buffers. This header-body division adds efficiency in deduplication, since the response header tends to differ for the same object (see Section 5 for more details).

One challenge in maintaining the flow buffer is memory management of three data buffers. Since a flow size is variable, one might be tempted to dynamically (re)allocate the buffers. However, we find that dynamic memory allocation often induces unpredictable delays, which increases random packet drops at engine threads. Instead, FloSIS uses user-level memory management with pre-allocated memory pools of fixed-sized chunks. For the flow buffer, one chunk is allocated initially for each data buffer when a flow starts. If more memory is needed, a new chunk is allocated but the new chunk is linked to the previous one with doubly-linked pointers. The flow management module has to follow the links to write the data into right memory chunks. While managing non-contiguous memory chunks is more difficult, the benefit of fixed-cost operations ensures a predictable performance at high-speed packet processing.

When a TCP flow terminates, the engine thread generates a flow metadata consisting of the following information: (i) start and end timestamps, (ii) source and destination [2] IP addresses and port numbers, and (iii) the location and length of the flow data on disk. Then the data in the flow buffer is moved to a large contiguous buffer called *w-buffer*, and the flow buffer is recycled for other flows. The flow data is also moved to w-buffer if its size grows larger than a single w-buffer. The w-buffer serves two purposes. First, it enables to have all flow data at a contiguous location before disk writing, which ensures to read all flow data with one disk seek. Second, it buffers the data from multiple flows to maximize the sequential write throughput of an HDD. It is the unit of disk writing, where a larger size would lead to a higher disk throughput while reducing the CPU usage. When the w-buffer is filled up, the engine thread gives the ownership to its writing thread, which writes to disk and recycles it.

For each flow, a flow data map is written prior to its flow data in w-buffer. The flow data map contains an array of disk locations and lengths of flow data buffers. In most cases, three (or four) flow data buffers follow the flow data map, but for a deduplicated buffer, the flow data map points to an earlier version. If a flow consists of multiple w-buffers, the flow data map is responsible for keeping track of all data buffers in the same flow, allowing flows of arbitrary size.

[2] Interpreted from the client side

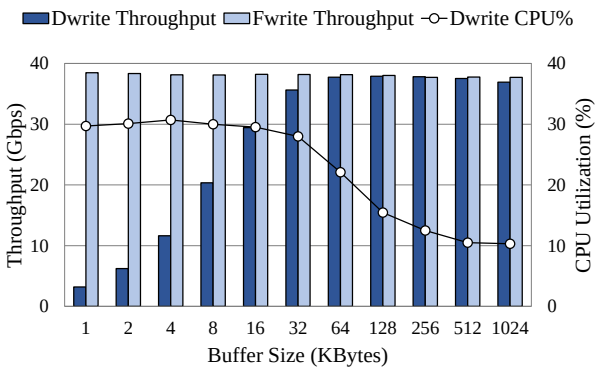


Figure 3: Disk writing throughput and CPU utilization

We comment that FloSIS avoids using buffered I/O, because the file system cache would be of little use for flow dumping and random disk access for user queries. Worse yet, the file system cache behavior becomes unpredictable when the cache buffer is full, which often leads to a catastrophic result such as kernel thrashing. In contrast, direct I/O in FloSIS performs I/O operations by direct DMA-mapping of user buffer to disk, and minimizes the effect of file system cache.

To figure out what size of w-buffer is appropriate as a good design choice in our reference platform, we run a benchmark test with varying buffer sizes. Figure 3 shows disk writing throughputs and CPU utilization of direct I/O using the `O_DIRECT` flag (Dwrite) and buffered I/O using `fwrite()` (Fwrite), where we compare the performance of buffered I/O just to understand the maximum disk performance. In these microbenchmarks, we run 24 threads on our platform, each of which occupies its own HDD and calls `write()` (`fwrite()` in Fwrite) continuously, and measure the aggregate throughputs and CPU utilization. Not surprisingly, Fwrite shows good performance regardless of the buffer size due to its data batching in the file system buffer. Since memory copying to a file system write buffer is much faster than actual disk I/O, the kernel always has enough data to feed to disk. This allows Fwrite to achieve the maximum disk throughput, whereas Dwrite achieves a similar performance only at 128 KB or larger buffer sizes due to the absence of kernel-level buffering. As expected, the CPU usage of Dwrite decreases as the buffer size increases. In FloSIS, we choose 512 KB as the size of w-buffer, which achieves almost peak performance only with 10% CPU usage.

4 Flow-level Indexing & Query Processing

In this section, we explain the two-stage flow-level indexing of FloSIS and real-time index generation.

4.1 Two-stage Flow-level Indexing

FloSIS writes flow data to a file (called a dump file) on each disk. When the current file size reaches a given threshold (*e.g.*, 1GB), FloSIS closes it and moves on to the next file for flow data dumping. The reason for file I/O instead of raw disk I/O is to allow other tools to access the dumped content. For sequential disk I/O, we preallocate the disk blocks contiguously in the files (*e.g.*, using Linux's `fallocate()`).

FloSIS handles flow data stored on each disk by two-stage flow-level indexes as shown in Figure 4. It uses the first-stage indexes to determine the files that contain the queried flows (file indexing). If a dump file has relevant flows, it filters the exact flows using the second-stage indexes (flow indexing). The details are as follows.

File indexing: The first-stage indexing is *file indexes*. Each dump file has in-memory file indexes that consist of two timestamps and four Bloom filters. These indexes are used to determine whether queried flows do **not** exist in the dump file, and the file can be passed. The two timestamps record the arrival times of the first and the last packet stored in the file. Each Bloom filter holds the information about one of the 4-tuple of a flow. For example, the source IP Bloom filter records all source IPs of the flows stored in the dump file. Using the filters, we can quickly figure out whether the dump file does **not** have any flow with a queried IP (or a port number).

Even if a Bloom filter confirms a hit with a queried IP address or a port number, there is still a small probability that it is a false positive. To achieve a tolerable false positive rate, we need to adjust the size of a Bloom filter and the number of hash functions, considering the number of elements. Assuming that an average flow size is 32 KB [31], we expect 32K flows in a dump file of 1 GB. A Bloom filter with 7 hash functions and 128 KB (or 2^{20} bits) of size, would reduce the false positive rate to 0.0011%, which should be small enough. This means that the memory requirement for file indexes is about 1.5 GB per 3 TB HDD or 36 GB for 24 HDDs.

Flow indexing: The second-stage is *flow indexes*. Flow indexes of a dump file consist of all flow metadata of those stored in the file and four sorted arrays. These are used to retrieve the flow metadata entries that match a user query. Using these entries, FloSIS reads the matched flow data from the dump file.

Each sorted array contains one of the flow tuple values in increasing order. An element in the array consists of a tuple value and a pointer to the flow metadata with that value. For example, a sorted array for destination IP has the entries with (a flow's destination IP, a pointer to the flow metadata) for all flows in the dump file, sorted by the destination IPs. When a query determines a target dump file in the first stage, sorted arrays are used to confirm if there are such flows that match the query. Using a binary search, one can locate the flow metadata fast with a specific tuple value in a query.

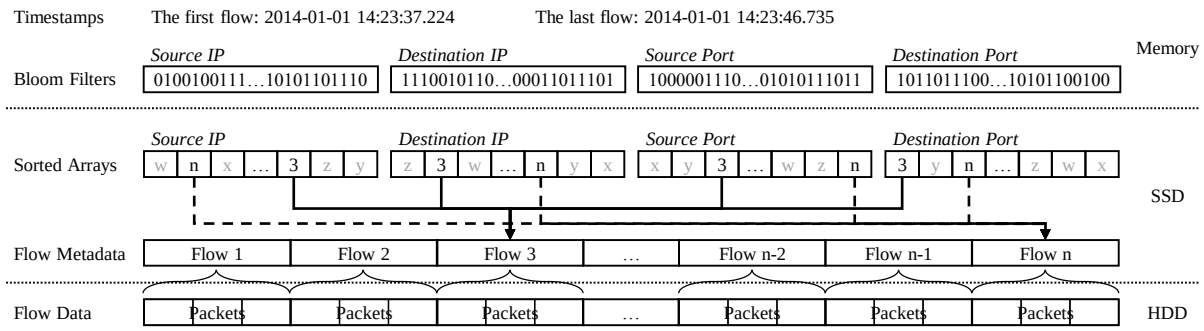


Figure 4: A hierarchical indexing architecture for each file in FloSIS

In terms of the space overhead, assuming a dump file contains 32K flows, four sorted arrays would take up 1 MB, and the array of 32-byte flow metadata would consume another 1 MB. The total space overhead for flow indexes for 24 HDDs of 3 TB would be 140 GB. Since this might be too large to fit in memory, we store these indexes on SSDs, and keep a fraction of them in memory if they are necessary for query processing.

4.2 Real-time Index Generation

To create the two-stage flow-level indexes in real time, FloSIS needs to produce the flow metadata, and updates the Bloom filters and the sorted arrays, for each completed flow with a budget of a few CPU cycles. However, sorting is a memory-intensive operation that consumes many CPU cycles. To avoid the interference caused by sorting, FloSIS separates the index creation in two steps.

In the first step, an engine thread creates the flow metadata with the 4-tuple values, start time, duration, and disk location of the flow when the flow terminates. Then, the engine thread computes Bloom filter hash values of source and destination IP addresses and port numbers, and updates the Bloom filters. Also, it adds an element at the end of each sorted array and leaves the array unsorted. The new element has one of the 4-tuple values and a pointer to the newly-created flow metadata. Sorting of the arrays is postponed until the engine thread fills up the dump file with flow data. Since sorting is skipped, the engine thread creates an index just in a few CPU cycles.

When a dump file is filled up, the second step begins. The engine thread sends an event to an indexing thread to sort the arrays for the 4-tuple values. The indexing thread sorts the arrays, and writes the flow metadata and the sorted arrays into the SSD. While the indexing thread performs CPU-intensive sorting, other threads working on the same core could suffer from a lack of computation cycles, which could affect the overall performance. To prevent this contention, FloSIS runs the indexing threads on dedicated CPU cores. We note that array sorting and index writing happen once every few seconds since it would take at least 5 seconds to fill a dump file of 1 GB, assuming the disk write throughput is 200 MBps. Dedicating a CPU core for each indexing thread would waste CPU cycles, so we co-locate several indexing threads on the same CPU core as shown in Figure 1.

4.3 Flow Query Processing

A FloSIS query consists of all or a part of six fields; a period of time (a time range), source and destination IP addresses and port numbers, and a packet-level condition in the BPF syntax. Each field can be a singleton value or a range of values using an IP prefix, a hyphen (-), or a wildcard (‘*’). FloSIS searches the flows that satisfy every condition in the query, *i.e.*, the query fields are interpreted as conjunctive. Every query field is optional and it is assumed to be any (’*’) in case it is missing in a query.

An example query looks like following.

```
./flowget -st 2015-1-1:0:0:0 -et
2015-1-2:0:0:0 -sip 10.1.2.3 -dip
128.142.33.1/24 -sp 20000-30000 -dp
80 -bpf tcp[tcpflags]&(tcp-syn) != 0
```

`flowget` is a query client program that accepts a user query and sends it to the query front-end server of FloSIS. The query searches the flows that overlap with the time range of one day from January 1st in 2015, and the source IP is 10.1.2.3 with ports between 20000 and 30000 with any destination IPs in 128.142.33.1/24 with port 80. It wants to see only the SYN packets of the matching flows.

When the query front-end server receives the user query, it distributes the query to all indexing threads and collects the results. Each indexing thread processes the query independently using the two-staged indexes of the dump files on their disks. The first step is ‘dump-file filtering’ using the file indexes. In this step, the indexing thread identifies which dump files might have matching flows. The indexing thread first finds a set of dump files whose timestamps overlap with the queried time range, and then checks the Bloom filters to see if the dump file might contain the flows with the requested fields. If a query field is a range, the indexing thread assumes that the corresponding Bloom filter returns a hit since checking all possible values would be too costly.

The second step is ’flow filtering’ using flow indexes. With the target dump files to look up, the indexing thread performs a binary search on each sorted array to find the

entries with the requested field. If the field is a range, it first checks whether the array has any overlap with the range, and a lookup for the closest entry to one end of the range would find all matching entries. Assuming the number of entries in an array is 32K, each binary search would require at most 15 entry lookups. That is, 60 entry lookups would be enough to find all matching flows in a dump file in the worst case, which takes a few microseconds on our platform. For each dump file, the indexing thread takes an intersection of the results from the four sorted arrays. After that, the indexing thread retrieves the matching flow metadata entries, and reads the flow data using the disk location in each flow metadata entry.

The last step in query processing is 'packet filtering'. The indexing thread runs BPF for detailed query condition on each packet header in the matching flows. A BPF provides a user-friendly filtering syntax, and is widely used for packet capture and filtering. While this step requires sequential processing, it is applied to a much reduced set of packets compared to brute-force filtering. If a BFP field is missing, then the last step is omitted and all data is passed to the query front-end, which relays it to the query client on a per-flow basis.

5 Flow-level Content Deduplication

It is well-known that the Internet traffic exhibits a significant portion of redundancy in the transferred contents [11, 31]. For example, a recent study reveals that one can save up to 59% of the cellular traffic bandwidth with simple network deduplication with 4KB fixed-sized chunking [31]. Thus, if we apply flow content deduplication to FloSIS, we may expect a non-trivial saving on storage space, which would help to extend the monitoring period at a high-speed network.

In FloSIS, we choose to adopt "whole-flow deduplication," which uses the entire flow data as a unit of redundancy (*i.e.*, a chunk). If the size of a flow exceeds that of the w-buffer, we make each w-buffer for the flow as a chunk. Whole-flow deduplication is a reasonable design choice, given that 80 to 90% of the total redundancy comes from serving the same or aliased HTTP objects [31], and that it consumes a smaller fraction of CPU cycles compared to other alternatives with fine-grained chunking [11]. Note that CPU cycles are important resources in FloSIS that regularly runs CPU-intensive tasks such as packet acquisition and index sorting.

For flow-level deduplication, an engine thread calculates the SHA-1 hash of the flow-assembled content in each direction (or only the response body in case of an HTTP transaction). CPU cycles for hash computation are amortized over the packets in a flow as the hash is partially updated on each packet arrival. This minimizes the fluctuation of CPU usage, which produces more predictable latency in hash computation. Out-of-order packets are buffered until missing packets arrive, and the hash is updated over the partially reassembled data. When a flow finishes (or when the flow size exceeds the w-buffer size), the SHA-1 hash is finalized. The hash is used as the content name to look up in a global deduplication table for cache hit. The deduplication table is a hash table that stores previous content hashes and their disk locations. It is shared by all engine threads to maximize the deduplication rate. When a lookup is a cache hit, the flow data map would be made point to the location of the earlier version when the flow is written to disk. If it is a cache miss, the SHA-1 hash and its disk location is inserted to the table.

We implement the deduplication table so as to minimize CPU usage and maximize the deduplication performance. We use a per-entry access lock to minimize the lock contention due to frequent accesses by multiple engine threads. We also pre-allocate a fixed number of table entries at initialization and use FIFO as the replacement policy (known to perform as well as LRU in network deduplication [31]). Each entry in the table is designed to be as small as possible to maximize the number of in-memory entries: 44 bytes per entry, 20 bytes SHA-1 hash, 8 bytes for disk location, and 16 bytes for doubly-linked pointers. The size of the deduplication table (or simply cache size) is a design parameter that trades off memory usage and deduplication performance. However, it is expected that the cache size does not have to be very big, because it is known that even a small cache significantly helps and the performance improvement diminishes as the cache size increases (i.e., diminishing return) [31, 11]. A recent study reveals that one can achieve 30 to 35% of deduplication rates with only 512 GB of content cache for a 10 Gbps cellular traffic link [31]. This translates to 16 million entries (or 1 GB) in a deduplication table assuming 32KB of average flow size. We understand that the actual numbers would vary in different environments, but we believe that a few GB of table entries should suffice in most cases.

6 Implementation

We implement FloSIS with 7,271 lines of C code that include engine, writing, and indexing threads as well as a query front-end server and a client. We mainly use Linux kernel version 2.6.32-47 for development and testing, but the code does not depend on the kernel version except for the PSIO library that requires Linux kernel 2.6.3x due to its driver code.

The number of engine, writing and indexing threads and their CPU core mapping are configurable depending on the hardware configuration. Each thread is affinitized to a CPU core and we replicate the thread mapping per each non-uniform memory access (NUMA) domain. FloSIS ensures that the network cards deliver the packets

only to the CPU cores in the same NUMA domain since crossing NUMA domains is expensive [22].

Since high zero-drop performance is one of our design goals, we pay special attention to the implementation that avoids unpredictable processing delay, whose key techniques are summarized in what follows: First, FloSIS limits maximum processing latency per each batch of packets from a NIC. Normally, an engine thread reads a batch of packets, processes them all, and moves on to read the next batch. However, when the number of packets in a batch is too large, it handles only a fraction of them at a time while leaving the rest in the buffer to minimize packet drop in a NIC. This technique is applied to other implementations such as SHA-1 hash calculation and flow management. Second, FloSIS pins each thread to a CPU core to avoid random scheduling delay and to improve CPU cache utilization. Also, it pre-allocates performance-critical memory chunks such as data, flow, and write buffers, and core data structures like flow metadata, indexes, flow and deduplication table entries at initialization to avoid the run-time overhead of dynamic memory allocation. FloSIS efficiently recycles the memory space, and dynamically allocates memory only when there is no more available pre-allocated memory space. The amount of pre-allocated memory is configurable, which in our prototype is set based on the recent traffic measurement [31]. Third, FloSIS minimizes unpredictable disk I/O delay as much as possible. We use direct I/O with enough buffer size to saturate the disk I/O capacity, and pre-allocate disk blocks in each file so that they are accessed sequentially. This significantly reduces the variance in disk I/O latency.

For evaluating FloSIS's performance, we extend a network workload generator initially developed for [23]. We implement the workload generator to (i) produce synthetic TCP packets with a random payload at a specified rate up to 40 Gbps regardless of packet size, and (ii) replay a real traffic packet trace (in the pcap file format) at a configurable replay speed. We ensure that the packets in the same flow are forwarded to the same destination NIC port in the order of their original record.

7 Evaluation

The goals of our evaluation are three folds. First, we evaluate if FloSIS provides a good performance [3] of packet- and flow-level capture and disk dumping with synthetic and real traffic. Second, we show how much reduction in query response time FloSIS's two-stage flow-level indexing brings over the existing state-of-the-art packet-level indexing. Third, we evaluate the effectiveness of flow content deduplication with a real traffic trace and measure the overhead.

[3] Unless specified otherwise, the throughput numbers include Ethernet frame overhead (24 bytes) to reflect the actual line rate.

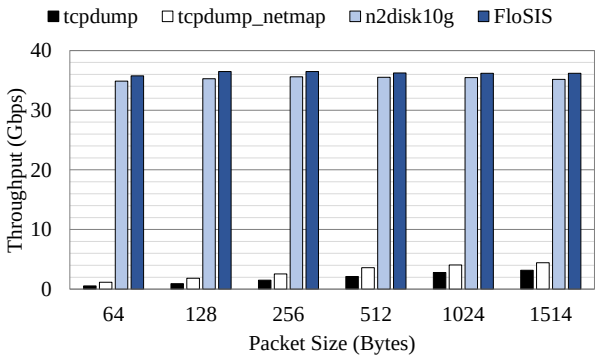


Figure 5: Synthetic TCP packet dumping throughputs of FloSIS, n2disk10g, and tcpdump at 40 Gbps input rate with various packet sizes

7.1 Experiment Setup

We run FloSIS and other packet capture systems on our reference server machine with dual Intel E5-2690 CPUs (2.90 GHz, 16 cores in total), two dual-port Intel 10 Gbps NICs with 82599 chipsets, 128 GB of RAM, 24 3TB SATA HDDs of 7200 RPM, and 2 SSDs of 512 GB Samsung 840 Pro. For workload generation, we run our packet generator on a similar machine with the same number of CPUs and NICs as the server. For synthetic workload, the packet generator sends the packets of the same size at a specified rate. For real traffic tests, the packet generator replays 67.7 GBs of a packet trace obtained from a large cellular ISP in South Korea [31]. This trace represents a few minutes of real traffic at one of 10 Gbps cellular backhaul links, and has 89 million TCP packets with 760 bytes of average packet size. The client and server machines are directly connected by four 10G cables since the traffic is either synthetically generated or replayed. However, in practice, the live traffic should be port-mirrored to the server via a switch.

7.2 Packet Capture & Dumping

We measure the performance of traffic capture and disk dumping of FloSIS for synthetic and real traffic workloads and compare them with those of existing solutions.

7.2.1 Synthetic Packet Workload

We first evaluate whether FloSIS achieves a good performance with packet capture and disk dumping regardless of incoming packet size. We have the packet generator transmit the packets of the same size at 40 Gbps, and measure the disk writing performance of captured packets. To compare the performance of packet-level capture and disk dumping with other tools, we disable other features of FloSIS such as flow management, index generation and writing, and deduplication.

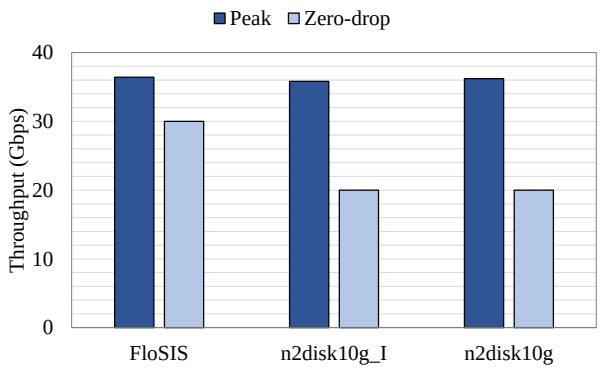


Figure 6: Peak and zero-drop throughput of FloSIS and n2disk10g with real traffic trace replay

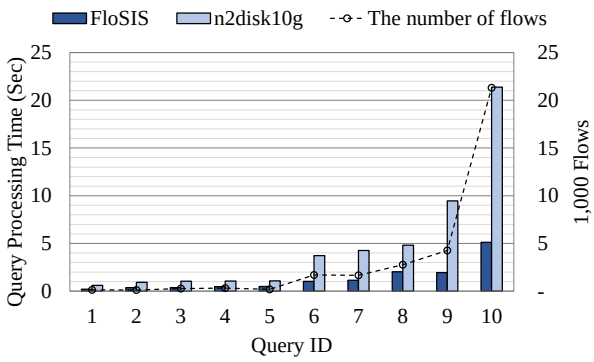


Figure 7: Query processing times over various number of matching flows

We compare the performances with tcpdump [6], tcpdump-netmap [7], and n2disk10g [17]. tcpdump is a popular packet capture system based on the libpcap library, but the libpcap library allows only one process/thread for each network interface. Since our machine has four 10G interfaces, we run 4 tcpdump processes and have each process write the captured packets to its dedicated HDD. On the other hand, tcpdump-netmap uses the netmap packet I/O framework [28] to accelerate the libpcap library performance. n2disk10g [17] uses PF_RING DNA [26] as scalable packet acquisition library, and can be configured to write to multiple disks in parallel. We select the parameters suggested by the manual of n2disk10g [2] for the best behavior. For fair comparison, we disable the indexing module for this test.

Figure 5 shows the throughputs [4] of the systems. FloSIS achieves 35.8 to 36.5 Gbps regardless of packet size, which is close to the peak aggregate disk writing performance as shown in Figure 3. This implies that FloSIS's packet buffering works well to achieve a sequential disk write performance. We observe that n2disk10g performs as well (34.9 to 35.6 Gbps), which is not surprising since its packet capture and parallel disk I/O is similar to that of FloSIS. Unfortunately, tcpdump and tcpdump-netmap perform poorly, achieving only 0.4 to 3.2 Gbps and 0.9 to 4.4 Gbps, respectively. While the netmap support improves the performance of tcpdump, the inefficiency in the libpcap library itself seems to limit the improvement.

7.2.2 Real Traffic Workload

We now measure the performance of handling the real traffic by replaying the LTE packet trace at a high speed. For this test, we enable flow management and indexing in FloSIS but disable flow deduplication to focus on the performance of flow processing. For comparison, we show the performances of n2disk10g with and without packet-level indexing. We also measure the peak zero-drop performances of both systems to estimate the practical performance for accurate monitoring.

As depicted in Figure 6, the peak performances of FloSIS and n2disk10g with/without indexing are similarly high as 36.4 Gbps, implying that disk writing bandwidth is the primary bottleneck, as in the synthetic workload case. This, in turn, implies that the extra overhead for flow management and two-stage flow index generation is small enough not to degrade the overall performance. However, FloSIS achieves better than n2disk10g in terms of zero-drop performance, 30 Gbps by FloSIS and 20 Gbps by n2disk10g regardless of indexing. We believe that our design choices such as minimizing the contention by separating the resources between interfering tasks, and aggressive amortization of resource consumption help produce more predictable processing delays, despite extra tasks like flow management and indexing. We do not know the root cause for lower zero-drop performance with n2disk10g since we do not have access to the source code, but it seems to pay less attention to the latency burstiness in packet capturing and disk dumping.

7.3 Query Processing

We evaluate the effectiveness of flow-level indexing of FloSIS. For the experiments, we replay the LTE traffic trace, and issue 10 queries to retrieve all flows between two IP addresses randomly chosen by us. The number of matching flows ranges from 119 to 21,300, and they are scattered around the disks. We compare the query response times with those of n2disk10g. The indexing of n2disk10g similar to that of FloSIS in that it uses tuple-based Bloom filters to skip the dump files, but it performs linear search on per-packet indexes called packet digests. Per-packet indexing not only incurs more space overhead but also requires more disk seeks to read the packets scattered around the disks.

Figure 7 compares the response times of 10 queries. We find that FloSIS outperforms n2disk10g by a factor of 2.2 to 4.9. As the number of flows increases, the per-

[4] We do not include the Ethernet frame overhead in throughput calculation in this test to focus on the performance of disk writing rather than packet capturing.

Queries	Q1	Q2	Q3	Q4
Bloom filter	100%	0.05%	-	-
Sorted array	-	10.83%	100%	99.43%
Flow read	-	89.12%	-	0.57%
Latency	3.3ms	330.2ms	80.5s	82.2s
Response	0B	2.7KB	0B	2.7KB

Table 1: Query processing times of singleton and range queries with/without disk access to read indexes and flow data

formance gap widens since n2disk10g suffers from more disk activity to gather all data. Unfortunately, we could not fill more data to examine the performance difference further since n2disk10g that we have is an evaluation version that runs for only a few minutes.

To investigate the FloSIS behavior with more flow data, we disable deduplication in FloSIS and fill the LTE trace repeatedly until we have 10 TBs of stored data. During the data filling phase, we feed in a few random flows that we retrieve in our queries. We use four types of queries to evaluate the effectiveness of our indexing structure. We use two singleton (Q1, Q2) and two range queries (Q3, Q4) for the entire time period. One of the two queries in each query type (Q1, Q3) looks for a flow that does not exist on disk, while the other queries (Q2, Q4) would return a small flow (2.7 KB) as a response.

Table 1 shows the response times for all queries. Q1 takes only 3.3 milliseconds (ms) since it would require checking only the Bloom filters in memory. Q2 takes more time (330.2 ms) since it has to read in the sorted arrays and flow metadata for the matching dump file from an SSD after cache hits with the Bloom filters. It also needs to read in the flow data from an HDD. Q3 and Q4 take much longer, 80.5 and 82.2 seconds, respectively, since FloSIS skips the Bloom filters for range queries, and reads in sorted arrays and flow metadata for all dump files. Since there are about 10,000 dump files for 10 TBs of data, the total size of all sorted arrays and flow metadata would be 20 GB. Actually, we could optimize the behavior further by reading the sorted arrays first, and read the flow metadata only if the query is a hit with the sorted arrays. Without indexing, it would require reading all data from HDDs which could take at least a few hours to resolve the queries.

7.4 Deduplication

We evaluate flow content deduplication with the real traffic trace. Deduplication is the most CPU-intensive task in FloSIS due to the overhead of real-time content hashing. We see almost full CPU utilization when we enable deduplication at a high traffic rate. While we expect to improve the performance in a cost-effective manner with SHA-1 computation offloading to off-chip GPUs [24], we focus on the CPU-only performance here. We also measure the level of storage compression by deduplication as we use more deduplication table entries.

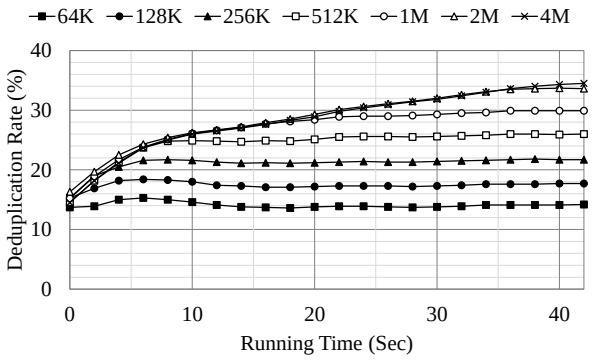


Figure 8: Deduplication rates over various numbers of deduplication table entries

FloSIS achieves about 15 Gbps of the peak zero-drop performance with deduplication. The performance is almost halved from the peak zero-drop performance without deduplication (in Figure 6), but its performance reaches 75% of the peak zero-drop performance of n2disk10g. Even if we amortize the latency of SHA-1 hashing over time, even a short spike of SHA-1 computation delay induces random packet drop, which drags the zero-drop performance. Nevertheless, we believe that one can monitor a full 10 Gbps link with deduplication without worrying about packet drop.

Finally, we estimate how many deduplication table entries are needed for a good deduplication rate. This question is difficult to answer given that we have only 67 GB of real traffic workload. For this workload, we draw a graph that shows the deduplication rates over various deduplication table size. Figure 8 presents the results over the entire period of traffic replay at 13.5 Gbps. The trend we find here is that (i) even a small cache works very effectively: A table with only 64K entries (or just 4 MB of content cache on disk) provides about 14% deduplication rate, (ii) it requires almost eight times more entries to double the deduplication rate, (iii) once the entries are filled up, the deduplication rate stabilizes over time. We obtain 34.5% of deduplication rate with 4 million entries, which is enough to suppress the actual redundancy in the original data. We do note that our trace is too small to draw any definitive conclusions, but we observe that a much larger workload shares a few similarities as our workload [31]. Given that 512 GB of content cache with 4KB chunks (or 16 million entries in the deduplication table) gives 30 to 35% of deduplication rate at a busy 10 Gbps link, we believe that a few GB of entries would be enough to achieve a good performance in practice.

8 Related Work

We briefly discuss the related work here. Due to a large body of works, comprehensive coverage is difficult, so we attempt to provide a summary of representative works.

High-speed packet acquisition and writing: Scalable and fast packet capture heavily affects the performance of network security monitoring systems such as firewalls and intrusion detection/prevention systems (IDS/IPS). High-end security systems typically adopt a parallel packet acquisition architecture that exploit the parallelism in modern CPUs and network cards. For example, software-based IDSes such as SnortSP [4], Suricata [5] and Para-Snort [14] deploy parallelized pipelining architectures suitable for a multi-core environment. Gnort [20], MIDeA [30] and Kargus [23] adopt a similar parallelized architecture for fast packet capture, but use general-purpose GPUs to offload the heavy pattern matching operations for extra performance improvement. n2disk10g [17] is a recent high-speed packet capture and storing system that adopts PF_RING DNA for packet acquisition, and uses direct disk I/O for scalable disk throughput. FloSIS shares the parallel packet capture architecture with these systems, but it goes beyond scalable packet capture and shows how one should map heterogeneous tasks of different computation budget (e.g., engine, writing, indexing threads) to computing resources for high zero-drop performance.

Intelligent indexing for fast retrieval: It is of critical importance to have an intelligent indexing structure for fast query response. pcapIndex [19] uses compressed bitmap indexes to index pcap files at off-line, and supports user queries with the BPF syntax. Hyperion [18] presents a stream file system optimized for sequential immutable streaming writes to store high-volume data streams, and proposes a two-level index structure based on Bloom filters. Gigascope [16] supports SQL-like queries on a packet stream, however, without archiving long-duration data. n2disk10g [17] supports per-packet indexing based on the Bloom filter and a packet metadata called packet digest. However, the per-packet indexing structures are used only for rough filtering of the entire data and it still requires linear search for query processing, often causing a long delay. FloSIS adopts flow-level indexing and query processing, which eliminates linear disk search and minimizes disk access in query processing. The required storage space for indexing is much smaller than that of packet-level indexing.

Network Traffic Compression: There have been many works on network traffic deduplication [9, 10, 11, 12, 29, 31]. These works show that typical Internet traffic has significant content-level redundancy. In terms of duplicate suppression, a smaller chunking unit and content-based chunking algorithm like Rabin's fingerprinting [27] generally leads to a higher deduplication rate at the cost of a larger computation overhead. To the best of our knowledge, FloSIS is the first traffic capture system that employs deduplication, and our choice of whole-flow deduplication is reasonable given the trade-off of computation overhead and the level of storage savings.

Cooke *et al.* present a multi-format storage that stores detailed information for recent data, but maintains only the summary of old data as time goes by [15]. Horizon Extender [21] proposes Web traffic classification and storing based on white-listing, considering the Web service popularity. It mainly focuses on compressing stored HTTP data for storage savings while minimizing the loss of data required to find the evidence of data leakage. Time Travel [25] stores only the first part of the large flows considering the heavy-tailed nature of network traffic. While these works reduce the storage requirement at the cost of losing a fraction of the data, FloSIS focuses on lossless storing of full flow data at a high-speed network for accurate traffic monitoring. However, we believe a hybrid approach is possible for further storage savings.

9 Conclusion

As the network edge bandwidth exceeds 10 Gbps, the demand for scalable packet capture and retrieval, used for attack analysis, network troubleshooting and performance debugging, is rapidly increasing. However, existing software-based packet capture systems neither provide high performance nor support flow-level indexing for fast query response. In this paper, we have proposed a highly scalable, software-based flow storing and indexing system, FloSIS, characterized by three features: exercising full parallelism, flow-aware processing for minimizing expensive disk access for user queries, and deduplication for efficient storage usage.

We have demonstrated that FloSIS achieves up to 30 Gbps of zero-drop performance without deduplication, and 15 Gbps with deduplication with real traffic storing and indexing, at the indexing storage cost of only 0.25% of the stored data. The two-stage flow-level indexing of FloSIS completes searching and reading 2.7 KB of flow data from 10 TB in 330.2 ms. Finally, our flow content deduplication reduces 34.5% of storage space for 67 GB of the real traffic trace with 256 MB of additional memory consumption.

Acknlowdgments

We would like to thank Shinjo Park and Sanghyeok Chun for their help on the initial design of the system. We also thank the anonymous reviewers and our shepherd, Keith Adams for valuable comments on the paper. This work was supported in part by National Security Research Institute grants #N04130037, #N04140063, Institute for Information & communications Technology Promotion (IITP) [B0126-15-1078], and the ICT R&D program of MSIP/IITP, Republic of Korea [B0101-15-1368, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services].

References

[1] Intel DPDK: Data Plane Development Kit. http://dpdk.org/.

[2] n2disk User's Guide. http://www.ntop.org/.

[3] netsniff-ng. http://netsniff-ng.org/.

[4] SnortSP (Security Platform). http://www.snort.org/snort-downloads/snortsp/.

[5] Suricata Intrusion Detection System. http://www.openinfosecfoundation.org/index.php/download-suricata.

[6] Tcpdump & Libpcap. http://www.tcpdump.org/.

[7] Tcpdump: netmap support for lipcap. http://seclists.org/tcpdump/2014/q1/9/.

[8] Wireshark. http://www.wireshark.org/.

[9] AGARWAL, B., AKELLA, A., ANAND, A., BALACHANDRAN, A., CHITNIS, P., MUTHUKRISHNAN, C., RAMJEE, R., AND VARGHESE, G. Endre: An end-system redundancy elimination service for enterprises. In *Proceedings of USENIX NSDI* (2010).

[10] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: The implications of universal redundant traffic elimination. In *Proceedings of ACM SIGCOMM* (2008).

[11] ANAND, A., MUTHUKRISHNAN, C., AKELLA, A., AND RAMJEE, R. Redundancy in network traffic: Findings and implications. In *Proceedings of ACM SIGMETRICS* (2009).

[12] ANAND, A., SEKAR, V., AND AKELLA, A. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *Proceedings of ACM SIGCOMM* (2009).

[13] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM 13*, 7 (July 1970), 422–426.

[14] CHEN, X., WU, Y., XU, L., XUE, Y., AND LI, J. Para-Snort: A Multi-thread Snort on Multi-core IA Platform. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS)* (2009).

[15] COOKE, E., MYRICK, A., RUSEK, D., AND JAHANIAN, F. Resource-aware multi-format network security data storage. In *Proceedings of ACM SIGCOMM workshop on Large-scale attack defense* (2006).

[16] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proceedings of ACM SIGMOD* (2003).

[17] DERI, L., CARDIGLIANO, A., AND FUSCO, F. 10 gbit line rate packet-to-disk using n2disk. In *Proceedings of IEEE INFOCOM Workshop on Traffic Monitoring and Analysis* (2013).

[18] DESNOYERS, P. J., AND SHENOY, P. Hyperion: high volume stream archival for retrospective querying. In *Proceedings of USENIX ATC* (2007).

[19] FUSCO, F., DIMITROPOULOS, X., VLACHOS, M., AND DERI, L. pcapIndex: an index for network packet traces with legacy compatibility. *ACM SIGCOMM Computer Communications Review 42*, 1 (Jan. 2012), 47–53.

[20] G. VASILIADIS, S. ANTONATOS, M. POLYCHRONAKIS, E. P. MARKATOS, AND S. IOANNIDIS. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[21] GUGELMANN, D., SCHATZMANN, D., AND LENDERS, V. Horizon extender: long-term preservation of data leakage evidence in web traffic. In *Proceedings of ACM CCS* (2013).

[22] HAN, S., JANG, K., PARK, K., AND MOON, S. B. PacketShader: a GPU-accelerated software router. In *Proceedings of ACM SIGCOMM* (2010).

[23] JAMSHED, M. A., LEE, J., MOON, S., YUN, I., KIM, D., LEE, S., YI, Y., AND PARK, K. Kargus: A highly-scalable software-based intrusion detection system. In *Proceedings of ACM CCS* (2012).

[24] JANG, K., HAN, S., HAN, S., MOON, S. B., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of USENIX NSDI* (2011).

[25] MAIER, G., SOMMER, R., DREGER, H., FELDMANN, A., PAXSON, V., AND SCHNEIDER, F. Enriching network security analysis with time travel. In *Proceedings of ACM SIGCOMM* (2008).

[26] NTOP. Libzero for DNA: Zero-copy flexible packet processing on top of DNA. http://www.ntop.org/products/pf_ring/libzero-for-dna/.

[27] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Harvard University, 1981.

[28] RIZZO, L. netmap: a novel framework for fast packet I/O. In *Proceedings of USENIX ATC* (2012).

[29] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM* (2000).

[30] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of ACM CCS* (2011).

[31] WOO, S., JEONG, E., PARK, S., LEE, J., IHM, S., AND PARK, K. Comparison of caching strategies in modern cellular backhaul networks. In *Proceedings of ACM MobiSys* (2013).

Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems

Andrey Goder Alexey Spiridonov Yin Wang

{agoder, lesha, yinwang}@fb.com
Facebook, Inc

Abstract

Data-intensive batch jobs increasingly compete for resources with customer-facing online workloads in modern data centers. Today, the two classes of workloads run on separate infrastructures using different resource managers that pursue different objectives. Batch processing systems strive for coarse-grained throughput whereas online systems must keep the latency of fine-grained end-user requests low. Better resource management would allow both batch and online workloads to share infrastructure, reducing hardware and eliminating the inefficient and error-prone chore of creating and maintaining copies of data. This paper describes Facebook's Bistro, a scheduler that runs data-intensive batch jobs on live, customer-facing production systems without degrading the end-user experience. Bistro employs a novel hierarchical model of data and computational resources. The model enables Bistro to schedule workloads efficiently and adapt rapidly to changing configurations. At Facebook, Bistro is replacing Hadoop and custom-built batch schedulers, allowing batch jobs to share infrastructure with online workloads without harming the performance of either.

1 Introduction

Facebook stores a considerable amount of data in many different formats, and frequently runs batch jobs that process, transform, or transfer data. Possible examples include re-encoding billions of videos, updating trillions of rows in databases to accommodate application changes, and migrating petabytes of data among various BLOB storage systems [11, 12, 31].

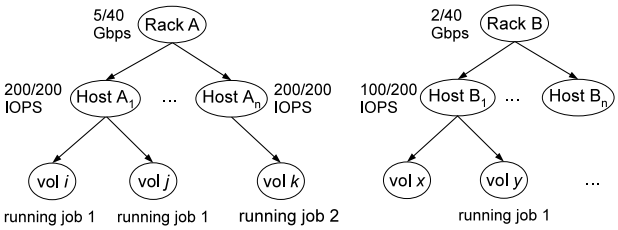
Typical large-scale data processing systems such as Hadoop [41] run against offline copies of data. *Creating* copies of data is awkward, slow, error-prone, and sometimes impossible due to the size of the data; *maintaining* offline copies is even more inefficient. These *troubles* overshadow the benefits if only a small portion of the offline data is ever used, and it is used only once. Furthermore, some batch jobs cannot be run on copies of data; e.g., bulk database updates must mutate the online data. Running batch jobs directly on live customer-facing production systems has the potential to dramatically improve efficiency in modern environments such as Facebook.

Unfortunately, existing batch job schedulers are mostly designed for offline operation [13, 15, 25, 28, 29, 32, 33, 36, 40, 42] and are ill-suited to online systems. First, they do not support hard constraints on the burdens that jobs place upon *data hosts*. The former may overload the latter, which is unacceptable for *online* data hosts serving end users. Second, batch schedulers assume immutable data, whereas online data changes frequently. Finally, a batch scheduler typically assumes a specific offline data ecosystem, whereas online systems access a wide range of data sources and storage systems.

Facebook formerly ran many data-intensive jobs on Hadoop, which illustrates the shortcomings of conventional batch schedulers. Hadoop tasks can draw data from data hosts outside the Hadoop cluster, which can easily overload data hosts serving online workloads. Our engineers formerly resorted to cumbersome distributed locking schemes that manually encoded resource constraints into Hadoop tasks themselves. Dynamically changing data poses still further challenges to Hadoop, because there is no easy way to update a queued Hadoop job; even pausing and resuming a job is difficult. Finally, Hadoop is tightly integrated with HDFS, which hosts only a small portion of our data; moving/copying the data is very inefficient. Over the years, our engineers responded to these limitations by developing many ad hoc schedulers tailored to specific jobs; developing and maintaining per-job schedulers is very painful. On the positive side, the experience of developing many specialized schedulers has given us important insights into the fundamental requirements of typical batch jobs. The most important observation is that many batch jobs are "map-only" or "map-heavy," in the sense that they are (mostly) embarrassingly parallel. This in turn has allowed us to develop a sched-

resources		jobs	
name	capacity	video re-encoding	volume compact
volume	1 lock	1 lock	1 lock
host	200 IOPS	100 IOPS	200 IOPS
rack	40 Gbps	1 Gbps	0 Gbps

(a) Resource capacity and job consumption



(b) Forest resource model

Figure 1: The scheduling problem: *maximum resource utilization subject to hierarchical resource constraints*.

uler that trades some of the generality of existing batch schedulers for benefits heretofore unavailable.

This paper describes Bistro, a scheduler that allows offline batch jobs to share clusters with online customer-facing workloads without harming the performance of either. Bistro treats data hosts and their resources as first-class objects subject to hard constraints and models resources as a hierarchical forest of resource trees. Administrators specify total resource capacity and per job consumption at each level. Bistro schedules as many tasks onto leaves as possible while satisfying their resource requirements along the paths to roots.

The forest model conveniently captures hierarchical resource constraints, and allows the scheduler to flexibly accommodate varying data granularity, resource fluctuations, and job changes. Partitioning the forest model corresponds to partitioning the underlying resource pools, which makes it easy to scale both jobs and clusters relative to one another, and allows concurrent scheduling for better throughput. Bistro reduces infrastructure hardware by eliminating the need for separate online and batch clusters while improving efficiency by eliminating the need to copy data between the two. Since its inception at Facebook two years ago, Bistro has replaced Hadoop and custom-built schedulers in many production systems, and has processed trillions of rows and petabytes of data.

Our main contributions are the following: We define a class of data-parallel jobs with hierarchical resource constraints at online data resources. We describe Bistro, a novel tree-based scheduler that safely runs such batch jobs "in the background" on live customer-facing production systems without harming the "foreground" workloads. We compare Bistro with a brute-force scheduling solution, and describe several production applications of Bistro at Facebook. Finally, Bistro is available as open-source software [2].

Section 2 discusses the scheduling problem, resource model, and the scheduling algorithm of Bistro. Section 3 explains the implementation details of Bistro. Section 4 includes performance experiments and production applications, followed by related work and conclusion in Sections 5 and 6, respectively.

2 Scheduling

Bistro schedules data-parallel jobs against online clusters. This section describes the scheduling problem, Bistro's solution, and extensions.

2.1 The Scheduling Problem

First we define a few terms. A **job** is the overall work to be completed on a set of data shards, e.g., re-encoding all videos in Haystack [12], one of Facebook's proprietary BLOB storage systems. A **task** is the work to be completed on one data shard, e.g., re-encoding all videos on one Haystack volume. So, a job is a set of tasks, one per data shard. A **worker** is the process that performs tasks. A **scheduler** is the process that dispatches tasks to workers. A **worker host**, **data host**, or **scheduler host** is the computer that runs worker processes, stores data shards, or runs scheduler processes, respectively.

Consider the scheduling problem depicted in Figure 1a. We want to perform two jobs on data stored in Haystack: video re-encoding and volume compaction. The former employs advanced compression algorithms for better video quality and space efficiency. The latter runs periodically to recycle the space of deleted videos for Haystack's append-only storage. Tasks of both jobs operate at the granularity of data volumes. To avoid disrupting production traffic, we constrain the resource capacity and job consumption in Figure 1a, which effectively allows at most two video re-encoding tasks or one volume compaction task per host, and twenty video re-encoding tasks per rack.

Volumes, hosts, and racks naturally form a forest by their physical relationships, illustrated in Figure 1b. Job tasks correspond to the leaf nodes of trees in this forest; each job must complete exactly one task per leaf. This implies a one-to-one relationship between tasks and data units (shards, volumes, databases, etc.), which is the common case for data-parallel processing at Facebook. A task

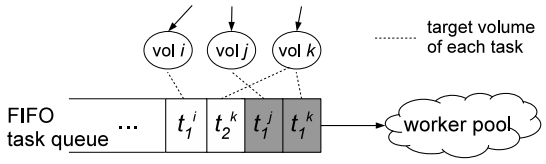


Figure 2: Head-of-line blocking using FIFO queue for our scheduling problem. *Here t_2^k can block the rest of the queue while waiting for the lock on vol k held by t_1^k, and possibly other resources at higher-level nodes.*

requires resources at each node along the path from leaf to root. In this figure, the two jobs have a total of four running tasks. Volumes i, j, k, and hosts A_1, A_n are running at full capacity, while other nodes have extra capacity for more tasks. More formally, our problem setting is the following.

- A resource forest with multiple levels, and a resource capacity defined at each level.
- A set of jobs $\mathcal{J} = \{J_1, ..., J_m\}$, where J_i consists of a set of tasks $\{t_i^a, t_i^b, ...\}$, corresponding to leaf nodes $a, b, ...$, respectively. This encodes the one-to-one relationship between tasks and data units.
- A task requires resources on nodes along its path to the root, and the demand at each level is defined per job.

Subject to resource constraints, *the scheduling objective is to maximize resource utilization.* The scheduler should never leave a task waiting if its required resources are available. High resource utilization often leads to high throughput, which in turn reduces the total *makespan*, i.e., the time required to finish all jobs. We do not directly minimize makespan, because the execution time of each task is unpredictable, and long tails are common [17]. We instead use scheduling policies to prioritize jobs, and mitigate long tails as described in Section 3.3. At large scale, the main challenge is to minimize scheduling overhead, i.e., to quickly find tasks to execute whenever extra resources become available.

In our scheduling problem formulation, each task requires resources along a unique path in our forest model. This is fundamentally different from the typical scheduling problem that considers only interchangeable computational resources on worker hosts. The latter is extensively studied in the literature, and First-In-First-Out (FIFO) queue-based solutions are common [13, 15, 25, 28, 29, 32, 33, 36, 40, 42]. For our problem, FIFO queues can be extremely inefficient. Figure 2 shows an example using the example of Figure 1. Assuming the worker pool has sufficient computational resources to run tasks in parallel, a task can easily block the rest of the FIFO queue if its required resources are held by running tasks. A non-FIFO scheduler might look ahead in the queue to find runnable tasks [42], but unless the scheduler examines the entire queue, it might overlook runnable tasks. Unfortunately, for large-scale computations, the overhead of inspecting the entire queue for every scheduling decision is unacceptable. Section 2.2 introduces a more efficient scheduling algorithm and contrasts it to this brute-force approach.

Algorithm 1 Brute-force scheduling algorithm (baseline)

```
1:  procedure ScheduleOne(M)
2:      for job J_i ∈ 𝒥 do
3:          for node l ∈ all leaf nodes of M do
4:              if task t_i^l has not finished and there are enough
                    resources along l to root then
5:                  update_resource(M, t_i^l)
6:                  run(t_i^l)
7:              end if
8:          end for
9:      end for
10: end procedure

11: procedure BruteForce( )
12:     while True do
13:         ScheduleOne(snapshot of the resource forest)
14:     end while
15: end procedure
```

Algorithm 2 Tree-based scheduling algorithm (Bistro)

```
1:  procedure TreeSchedule( )
2:      while True do
3:          t ← finished_task_queue.blockingRead()
4:          d ← the highest ancestor of the leaf node corre-
                sponding to t where t consumes resources
5:          ScheduleOne(snapshot of the subtree at d
                and the path from d to root)
6:      end while
7:  end procedure
```

2.2 Our Scheduling Algorithms

Our performance baseline is the aforementioned *brute-force* scheduler that avoids unnecessary head-of-line blocking by searching the entire queue for runnable tasks. The pseudocode is in Algorithm 1. The BruteForce function executes in an infinite loop. Each iteration examines all tasks of all jobs to find runnable tasks to execute, and updates resources on the forest model accordingly. The complexity of each scheduling iteration is $O(\text{number of jobs} \times \text{number of leaves} \times \text{number of levels})$.

In practice, each iteration of the brute-force scheduler

is slow—around ten seconds for a modest million-node problem. This incurs a significant scheduling overhead for short tasks. On the other hand, one iteration can schedule multiple tasks because it runs asynchronously using snapshots of the resource model. So, as tasks finishes more quickly, each iteration reacts to larger batches of changes, amortizing the scheduling complexity.

Algorithm 2 is Bistro's more efficient *tree-based* scheduling algorithm. The algorithm exploits the fact that a task requires only resources in one tree of the resource forest, or just a subtree if it does not consume resources all the way to root. We can therefore consider only the associated tree for scheduling when a task finishes and releases resources. This algorithm again executes the scheduling procedure in an infinite loop, but instead of blindly examining all tasks of all jobs in each iteration, it waits for a finished task, and invokes the scheduling iteration on the corresponding subtree only. Although TREESCHEDULE may not schedule tasks in large batches as BRUTEFORCE does, it enables multi-threaded scheduling since different tasks often correspond to non-overlapping trees or subtrees. Bistro uses reader-writer locks on tree nodes for concurrency control.

2.3 Model Extensions and Limitations

The forest resource model is easy to extend. First, the scheduling problem described in Section 2.1 assigns one resource to each level, but it is straightforward to have multiple resources [24]. This allows heterogeneous jobs to be throttled independently, e.g., I/O bound jobs and computation bound jobs can co-exist on a host. In addition, different jobs can execute at different levels of the trees, not necessarily the bottom level. For example, we may run maintenance jobs on server nodes alongside video encoding jobs on volume nodes.

For periodic jobs like volume compaction in Figure 1, Bistro has a built-in module to create and refresh time-based nodes. We add these nodes to volumes nodes as children, so volume compaction runs at this new level periodically as the nodes refresh.

Partitioning the forest model for distributed scheduling is flexible too. Node names are unique, so we can partition trees by a hash of the names of their root nodes. Partitioning by location proximity is another choice. Some of our applications prefer filtering by leaf nodes, such that after failover, the new host that contains the same set of volumes or databases is still assigned to the same scheduler.

In addition to the forest model, Bistro supports Directed Acyclic Graphs (DAGs). A common case is data replicas, where multiple servers store the same logical volume or database. With DAGs, resources are still organized by levels, and tasks run at the bottom level. For each bottom level node, Bistro examines all paths to root and can schedule a task if any path has all the resources it needs.

Finally, Bistro is designed for embarrassingly parallel jobs, or "map-only" jobs, where each task operates on its own data shard independently. At Facebook, other than data analytic jobs on Hive [37], many batch jobs are map-only or *map-heavy*. Bistro applies to map-heavy jobs by running the reduce phase elsewhere, e.g., as a Thrift service [1]. For jobs with non-trivial "reduce" phases, our engineers came up with a solution that runs "map" and "reduce" phases on separate Bistro setups, buffering the intermediate results in RocksDB [5] or other high-performance data store; see Section 4.2.1 for an example.

2.4 Scheduling Against Live Workloads

Bistro requires manual configuration of resource capacity based on the characteristics of live foreground workloads, and manual configuration of the resource consumption of background jobs. Users can adjust these settings at runtime upon workload changes, which Bistro will enforce in the subsequent scheduling iterations by scheduling more or killing running tasks. Bistro could monitor realtime resource usage at runtime and adjust these settings automatically. However, live workloads are often spiky, such that aggressive dynamic scheduling requires reliable and rapid preemption of background jobs. This is challenging if the data, worker, and scheduler hosts are all distributed, and it complicates the task design by the requirement of handling frequent hard kills. At Facebook, job owners usually prefer static resource allocation for simplicity.

3 Implementation

This section discusses the implementation details of Bistro, including its architecture and various components.

3.1 Architecture

Figure 3 depicts the architecture of Bistro, which consists of six modules. All of these modules work asynchronously, communicating via either snapshots or queues. The *Config Loader* reads and periodically updates the resource and job configuration from some source, such as a file, a URL, or a Zookeeper instance [27]. Based on the current resource configuration, the *Node Fetcher* builds and periodically updates the resource model. For example, if we want to process all files in a directory,

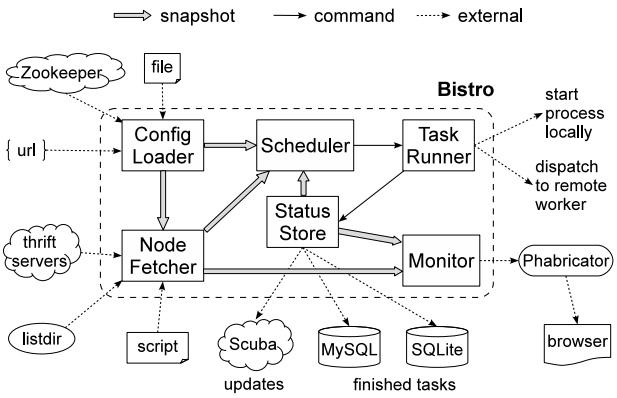


Figure 3: Bistro architecture

the Node Fetcher can periodically scan the directory to maintain an up-to-date view of its contents. Most of our large-scale jobs receive their nodes from a set of Thrift [1] servers for scalability and reliability. These servers obtain the resource model from databases or Zookeeper, typically with an intermediate Memcached layer. The *Scheduler* chooses tasks to execute based on the latest configuration and resource model, as well as the statuses of running and unfinished tasks. These statuses are maintained by the *Status Store*. The *Task Runner* receives a list of tasks to start, and can either run them locally or dispatch them to remote workers depending on the configuration, discussed in the next subsection. The Runner also monitors task execution and sends status updates to the Status Store. Users observe job progress through a Web UI provided by the *Monitor*.

3.2 Scheduling Modes

The forest resource model of Bistro facilitates flexible scheduler/worker configuration. We can have either one Bistro instance for centralized scheduling or multiple instances for distributed scheduling. Workers can reside on either data hosts or separate worker pool. This leads to four scheduling modes to accommodate diverse job requirements in large data centers, described below.

The *single/co-locate* mode has one central Bistro instance, and one worker on each data host, which receives only tasks that access local data. In addition to data locallity, the centralized scheduler can enforce optimal load balancing and fair scheduling. Since the entire forest model is on one Bistro instance, we can add a common root to all trees to enforce global resource contraints too. The scheduler is a single point of failure but Bistro can log task statuses to redundant storage for fail over. We employ this mode whenever the data hosts have sufficient compute resources and the total number of nodes is small.

The *multi/co-locate* mode is for large-scale jobs that a single scheduler cannot handle efficiently due to an excessively large resource model or an excessively high turnover rate. The latter happens if tasks are short or high concurrency. If each data host corresponds to an independent resource tree, we often run one Bistro instance on each data host too, which avoids network communication by connecting to the co-located worker directly. This scheduling mode is very robust and scalable since each Bistro instance works independently. One downside of share-nothing schedulers is poor load balancing. For a view of the global state, Bistro's monitoring tools efficiently aggregate across all schedulers.

If data hosts have limited compute resources, we have to move the workers elsewhere. The *single/remote* mode is similar to single/co-locate with one scheduler and multiple workers on dedicated worker hosts, good for load balancing. The *multi/remote* mode has multiple Bistro instances. In this case, similar to multi/co-locate, we can have one Bistro instance per worker to avoid network communication if we assign a fixed partition to each worker. Alternatively we can run schedulers on dedicated hosts for dynamic load balancing. Bistro's Task Runner module can incorporate probabilistic algorithms such as *two-choice* for scalability [30, 32].

3.3 Scheduling Policies

Bistro implements four scheduling policies to prioritize among jobs. Round robin loops through all jobs repeatedly and tries to find one task from each job to execute, until no more task can be selected. Randomized priority is similar to *weighted round robin*. We repeatedly pick a job with probability proportional to its priority, and schedule one of its tasks. Ranked priority sorts jobs by user defined priorities, and schedules as many tasks as possible for the job with the highest priority before moving to the next job. Long tail scheduling policy tries to bring more jobs to full completion via "ranked priority" with jobs sorted by their remaining task count in increasing order.

Round robin and randomized priority approximate fair scheduling, while ranked priority and long tail are prioritized. The latter two can be risky because one job that fails repeatedly on a node can block other jobs from running on that node. All these policies guarantee locally maximal resource utilization in the sense that no more tasks can run on the subtree. Supporting globally maximal resource utilization or globally optimal scheduling polices would require much more complex computation and incur greater scheduling overhead. We have not encountered any production application that requires optimal scheduling.

3.4 Config Loader and Node Fetcher

For each Bistro deployment, users specify the resource and job configuration in a file, a URL, or a Zookeeper cluster [27]. Most of our production systems use Zookeeper for reliability. Bistro's Config Loader refreshes the configuration regularly at runtime.

The resource configuration specifies the tree model, resource capacity at each level, and the scheduling mode. It also specifies a *Node Fetcher* that Bistro will call to retrieve nodes of the tree and periodically refresh them. Different storage types have different node fetcher plugins so Bistro can model their resource hierarchies.

A Bistro deployment can run multiple jobs simultaneously. Each job defaults to the same number of tasks, corresponding to all bottom nodes in the forest model. The job configuration specifies for each job the resource consumption and the task binary. Each tree node has a unique name, and Bistro passes the leaf node name to the command so it will operate on the corresponding data shard. Users can include `filters` in a job configuration to exclude nodes from executing tasks for the job. Filters are useful for testing, debugging, and non-uniform problems.

3.5 Status Store and Fault Tolerance

A task can be in one of several states, including ready, running, backoff, failure, and done. The Status Store module records status updates it receives from Task Runner. It also provides status snapshots to Scheduler and Monitor. A task is uniquely identified by a (job name, node name) pair, and therefore Status Store does not need to track resource model changes.

Status Store can log task statuses to external storage for failure recovery, such as Scuba [7], remote MySQL databases, and local SQLite databases. Scuba is scalable and reliable but has limited data size and retention. Replicated MySQL is more reliable than SQLite, while the latter is more scalable because it is distributed over scheduler hosts. Users choose the external storage based on job requirements. In practice, most failures are temporary. Therefore, users often choose SQLite for the best performance, and write *idempotent* tasks so Bistro can always start over for unrecoverable failures.

Users monitor tasks and jobs through a Web UI, which is an endpoint on Phabricator [3]. The endpoint handles browser requests and aggregates task status data from one or multiple Bistro instances, depending on the scheduling mode. Bistro logs task outputs to local disk, which can also be queried through Phabricator for debugging.

3.6 Task Runner and Worker Management

After scheduling, the Task Runner module starts scheduled tasks, monitors their executions, and updates the Status Store. Task Runner supports both running tasks locally and dispatching it to remote workers. In the latter case, it considers resource constraints on worker hosts as well as *task placement constraints* [23, 32, 35]. Currently, each Bistro instance manages its own set of workers, but it is straightforward to incorporate probabilistic algorithms such as two-choice for better load balancing [30].

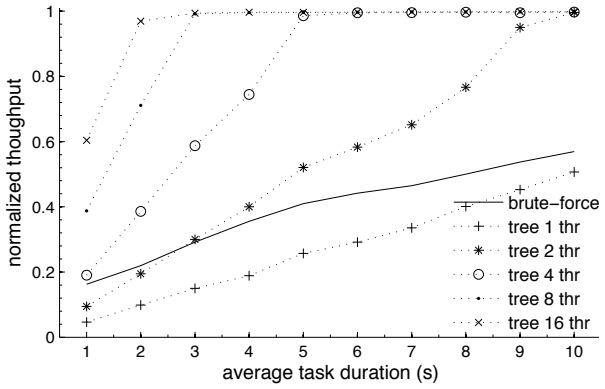
4 Evaluation

We implemented Bistro in C++ with less than 14,000 physical source lines of code (SLOC). As explained in Section 2.1, FIFO-queue-based schedulers lead to almost serial execution for our scheduling problem. Unfortunately most schedulers in the literature as well as commercial schedulers are queue-based, not very interesting for comparison. Therefore, we compare our tree-based scheduling algorithm with the brute-force approach, both explained in Section 2.2. The brute-force approach was widely used in our ad-hoc schedulers before Bistro.

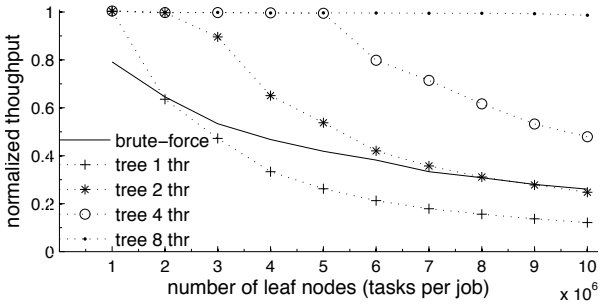
4.1 Microbenchmark

We run our experiments in multi/remote mode with 10 scheduler hosts and 100 worker hosts, each scheduler host has 16 Xeon 2.2 GHz physical cores and 144 GB memory. The resource setup and workload mimic our database scraping production application, which is most sensitive to scheduling overhead because of its short task duration. There are two levels of nodes (or three if adding a root node to enforce global resource constraints), host and database. Each data host has 25 databases; we vary the number of data hosts to evaluate Bistro's scalability. The resource model is partitioned among the 10 schedulers by hashing the data host name. Each task sleeps for a random amount of time that is uniformly distributed over $[0, 2d]$, where d is a job configuration value that we vary.

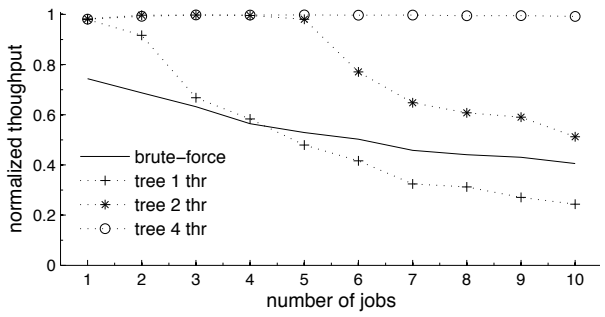
Figure 4 shows the performance results with different resource and job settings. In the legend, "brute-force" refers to brute-force scheduling, and "tree n thr" means tree-based scheduling with n threads, where different threads work on different subtrees, explained in Section 2.2. We show normalized throughput of different scheduling algorithms where 100% means no scheduling overhead, calculated as the maximum number of concurrent tasks allowed by resource constraints divided by the



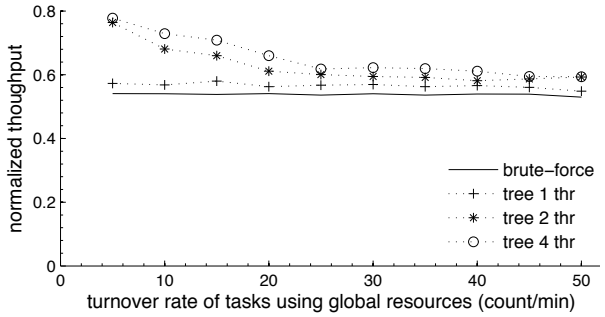
(a) varying task duration (3m nodes, 10 jobs)



(b) varying number of nodes (5 jobs, task duration 5s)



(c) varying number of jobs (3m nodes, task duration 5s)



(d) varying number of tasks using global resources (300k nodes, 5 jobs, task duration 5s)

Figure 4: Scheduling throughput using synthetic workload under different configurations.

average task duration. In all our experiments, the throughput increases slightly for both scheduling algorithms as more tasks finish because there are less tasks to examine in each scheduling iteration. It then drops quickly when there are not enough tasks to occupy all resources. We measure the throughput at the midpoint of each run, and report the average of five runs. The variation is negligible and therefore we do not include error bars in the figure.

Overall, brute-force scheduling shows relatively stable performance because of asynchronous scheduling. If a scheduling iteration takes too long, more tasks finish and release resources, such that the next iteration can schedule more tasks. For a model size of 300k nodes at one scheduler and 5 jobs, each scheduling iteration takes roughly 5 seconds. Tree-based scheduling achieves almost zero overhead when there are enough threads to handle the turnover rate, otherwise the performance drops quickly.

The worst case for the tree-based scheduling is with global resources, shown in Figure 4d. In this case there is only one scheduler, and we add a root node to model a global resource, as explained in Section 2.3. Only one job uses the global resource. We set the job to a high priority and use the *ranked priority* scheduling policy so it runs at the maximum throughput allowed by the global resource constraint. The result shows that the tree-based scheduling performs similarly to brute-force scheduling when the turnover rate of the tasks using global resources is high. This is because when global resources become available for scheduling, Bistro needs to lock the entire tree for scheduling. The overhead of brute-force scheduling is not affected by global resources since it always takes a snapshot of the entire model.

4.2 Production Applications

There are roughly three categories of data store at Facebook, SQL databases for user data, Haystack [12] and F4 [31] for Binary Large OBjects (BLOBs), and Hive [37] and HBase for mostly analytical data. Bistro is currently the only general-purpose scheduler for batch jobs on SQL databases and Haystack/F4. It has also replaced Hadoop for some map-only jobs on HBase, especially on clusters that serve live traffic.

Table 1 shows some of our production applications. The mode column indicates the scheduling mode discussed in Section 3.2. The resource model section shows the characteristics of the resources at each level. The concurrency column shows the resource capacity of that level divided by the default consumption. The change rate column is the average percentage of nodes added or removed per time interval, measured over 30 days in pro-

Application	Mode	Resource Model					Job			
		level	resource type	node count	concurrency	change rate	concurrent jobs	change rate	task data	avg. task duration
Database Iterator	single/ remote	1	root	1	various	-				
		2	host	∼10 k	2	2.1%/ hr				
		3	db	∼100 k	1	2.1%/ hr	5	10/ day	300k rows	5 min
Database Scraping	single/ remote	1	host	∼10 k	1	2.4%/ hr				
		2	db	∼100 k	1	2.3%/ hr	10	5/ hr	1 MB	5 sec
Storage Migration	single/ co-locate	1	host	∼1k	3	<1%/ day				
		2	dir	∼100 k	1	<1%/ day	1	<1/ day	100 GB	7 hr
Video Re-encoding	multi/ remote	1	host	∼1k	1	1.3%/ hr				
		2	volume	∼100 k	1	1.2%/ hr				
		3	video	∼1 b	1	1.3%/ hr	1	<1/ day	5 MB	20 min
HBase Compression	multi/ co-locate	1	host	∼100	3	2.6%/ min				
		2	region	∼1 k	3	3.1%/ min				
		3	time	∼10 k		8.3%/ min	10	10/hr	3m rows	1 min

Table 1: Summary of some production applications at Facebook. *We show node number in order of magnitude to preserve confidentiality, prefixed by ∼. Numbers in the Job section are approximate too for the same reason.*

duction. Node Fetcher refreshes the model every minute by polling the publishing source, the actual change rate can be higher. The job section shows the characteristics of jobs and tasks. The change rate shows the actual number of changes rather than a percentage, because often we just change the configuration of a job instead of adding or removing jobs. Next we discuss these applications.

4.2.1 Database Iterator

Database Iterator is a tool designed to modify rows directly in hundreds of thousands of databases distributed over thousands of hosts. For example, backfilling when new social graph entities are introduced, and data format changes for TAO caching [14]. Because of the lack of batch processing tools for large-scale online databases, this use case motivated the Bistro project.

We use a single Bistro instance since the resource model fits in memory, and some jobs need global concurrency limits. The scheduler sends tasks to a set of workers, i.e., single/remote scheduling mode. We measure 2.1% of hosts and databases change every hour, mostly due to maintenance and newly added databases. Users write their own iterator jobs by extending a base class and implementing `selectRows()`, which picks rows to process, and `processRows()`, which modifies the rows selected. Each task (one per database instance) modifies roughly 300 thousand rows on average. The task duration, while very heterogeneous, averages around five minutes. The total job span is much longer due to long tails.

Before Bistro, Database Iterator ran on a distributed system that executes arbitrary PHP functions asynchronously. Bistro replaced the old system two years ago and it has processed more than 400 iterator jobs and hundreds of trillions of rows. Table 2 compares both systems from the point of view of the production engineers.

Most Database Iterator jobs read and write data on a single database, so our forest resource model is sufficient for protection. Some jobs, however, read from one database and write to many other databases because of different hash keys. Database Iterator supports general "reduce" operations by buffering the intermediate results in RocksDB [5]. One such example was a user data migration. We used two Bistro setups in this case since the source and destination databases are different. The "map" Bistro runs tasks against source databases, writing to RocksDB "grouped by" the hash keys of destination. The "reduce" Bistro runs tasks against destination databases, reading from RocksDB using the corresponding keys. Both sets of databases, totalling hundreds of thousands, were serving live traffic during the migration.

4.2.2 Database Scraping

Database scraping is a common step in ETL (Extract, Transform, Load) pipelines. The resource configuration is similar to database iterator except that we include all replicas because the jobs are read-only; see Section 2.3 on how to model replicas. The scheduling mode is also single/remote but the worker hosts are dedicated to scraping jobs, which allows Bistro to manage their resources and enforce task placement constraints. For example, since scraping jobs run repeatedly, we monitor their resource usage and balance workload among workers.

Before Bistro, we ran scraping on a distributed execution system for time-based jobs, compared in Table 3.

	a proprietary asynchronous execution framework	Bistro
resource throttling	Similar to brute-force scheduling except that tasks have to lock resources themselves by querying a central database, which gets overloaded frequently.	Supports hierarchical resource constraints so tasks need not check resources themselves. Tree-based scheduling achieves better throughput.
model/job updates	Database failovers, new databases, and job changes are frequent, such that queued tasks become outdated quickly, and no job finishes 100% in one run.	Resource models and job configurations are always up to date. All jobs finish completely.
canary testing	No support. Canary testing is crucial because each job runs only once. We make adjustments frequently.	Supports whitelist, blacklist, fraction, and etc. Easy to pause, modify, and resume jobs at runtime.
monitor	Shows various performance counters but not the overall progress, since it does not know the entire set of tasks.	Shows a progress bar per job, with all tasks in different statuses.

Table 2: Feedback from Database Iterator operations team

	an execution framework for time-based jobs	Bistro
LOC	1,150 for an ad-hoc resource manager (RM)	135 for a new node fetcher
resource throttling	The framework applies batch scheduling. Tasks query RM to lock resources. The framework supports only slot based resource allocation on worker hosts.	Supports hierarchical resource constraints for external resources, as well as resource constraints and task placement constraints on workers.
model/job updates	RM takes 45 min to save all resources and tasks to database before scheduling. No update afterwards.	21 sec to get all resources and tasks, and constantly updated.
SPOF	RM, central database, and scheduler all failed multiple times, halting production.	Only the scheduler, which fails over automatically
Priority	No support. Often all jobs get stuck at 99%	Ranked Priority gets jobs to 100% one by one
Job filter	Choices of databases are hard coded in each job.	Automatically tracks database changes.

Table 3: Feedback from Database Scraping operations team

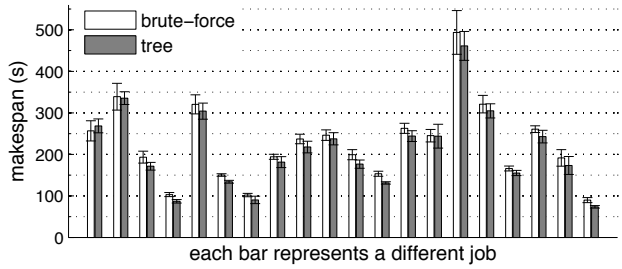
Similar to the asynchronous execution framework used by Database Iterator previously, the system does not consider data resources consumed by tasks, and our engineers had to write an ad-hoc resource manager to lock databases for tasks, which did not scale well. The scheduling algorithm of the distributed execution system is similar to brute-force scheduling, where the scheduler loops through all unfinished tasks repeatedly. Upon start, the resource manager takes 45 minutes to collect all resource and tasks, and log them to a central database. Discounting the 45 minute startup time, we compare the performance of Bistro with brute-force scheduling by replaying a real scraping workload, shown in Figure 5.

There are 20 jobs in the workload, and we are interested in the makespan of the schedule with different job settings. For each experiment, we take 10 runs with different leaf node ordering to average out the makespan variation due to the long tail. The figure shows both the average makespan and the standard deviation. Figure 5a is the makespan of scheduling only one job. Tree scheduling is only slightly better than brute-force scheduling, because long tail tasks often dominate the entire schedule. When there are multiple jobs, on the other hand, tree-based scheduling can be as much as three times faster than brute-force scheduling, shown in Figure 5b. Bistro took over scraping jobs a year ago, which significantly reduced the scraping time, and eased our daily operation.

4.2.3 Haystack/F4 Applications

Storage Migration in Table 1 refers to a one-time job that moved sixteen petabytes of videos from a legacy file system to Haystack. Bistro scheduled tasks on each server of the proprietary file system, which sent video files from local directories to Haystack. The destination Haystack servers were not serving production traffic until the migration completed, so we did not throttle tasks for destination resources. At the time of migration, newly uploaded videos already went to Haystack and the old file system was locked. Therefore, the model change rate was low. The overall job took about three months to finish.

Video re-encoding is a project to recompress old user videos with more advanced algorithms to save space without detectable quality degradation. We use fast compression algorithms for live video uploads in order to serve them quickly. Recompressing "cold" videos saves storage space substantially. Compressing each user video using the advanced algorithm takes roughly twenty minutes. Compressing a whole volume can take several days even with multithreading, during which time many videos may

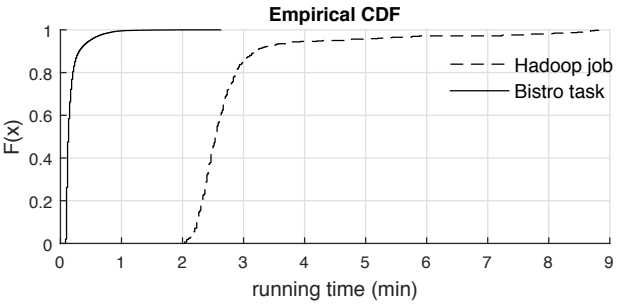


(a) Makespan of scheduling one job

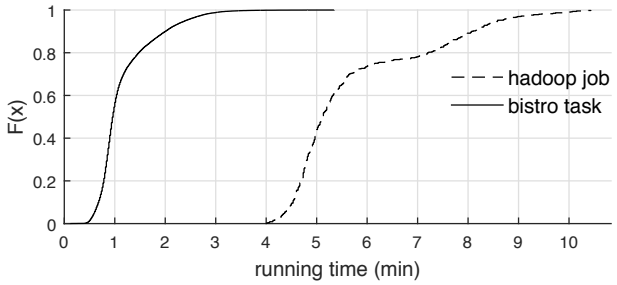


(b) Makespan of scheduling multiple jobs concurrently

Figure 5: Makespan of our Database Scraping jobs



(a) HBase table A



(b) HBase table B

Figure 6: Running time of Bistro tasks vs. Hadoop jobs for HBase Compression

be deleted or updated by users, leading to corrupted metadata. Therefore we process tasks at video level rather than volume level, and let Bistro keeps all videos updated. Each Bistro instance can store a hundred million nodes in memory so we only need a dozen or so scheduler hosts. Compressing a billion videos in a reasonable amount of time, however, requires hundreds of thousands of workers. We are working on a project that harnesses underutilized web servers for the computation.

4.2.4 Hbase Compression

HBase [21] is convenient for long-term data storage with schema. We often need to compress old data to save space. Our performance counter data is one such example. There are three million rows stored every second from all performance counters. We want to preserve old data at coarse granularity. This is done by a compression job that runs every 15 minutes.

We set up Bistro in multi/co-locate mode so each instance processes only local HBase regions. This essentially enforces *data locality*, which benefits our I/O heavy job. In addition to the node fetcher that retrieves HBase hosts and regions, we generate time-based nodes to run the compression job periodically; see Section 2.3 for detail. During our measurements, a few HBase servers did not return region information reliably, so we see a significant percentage of node changes.

Before Bistro, we ran the compression job on Hadoop, compared in Table 4. The HBase cluster is heavily used internally for performance counter queries, and Hadoop often disturbed the query workload. In addition, the MapReduce computation model waits for the slowest task in each phase, which caused 10 % jobs to miss deadlines due to long tail. In contrast, Bistro runs each task independently, and slow data shards may catch up later. Figure 6 shows the empirical CDF of running time for both Hadoop jobs and individual map tasks, measured over one week in a production cluster. There are two different tables we run compression jobs on, with different amounts of data. The figure shows that the Hadoop job histogram starts roughly at the tail of the individual task CDF, confirming the barrier problem.

5 Related Work

Early data center schedulers focus on compute-intensive workload and task parallelism, often associated with High Performance Computing (HPC) and Grid Computing. In these environments, a job can be one task or a series of tasks organized by a workflow, each corresponding to a different binary. The scheduler assigns tasks to different machines, considering their resource requirements, placement constraints, and dependencies. Scheduling objectives include minimiz-

	Hadoop	Bistro
duration	Depends on the slowest map task	Each task runs independently
skipped jobs	10%, which happens when the previous job did not finish in its time window.	No missing job since tasks run independently. The percentage of skipped tasks are less than 0.1%
Resource throttling	No support. Starting a new job often slows down the entire cluster, causing service outage	No detectable query delay with our hierarchical resource constraints.
Data locality	99%. Speculative execution kicks in for long tails, which makes it worse since our tasks are I/O bound.	100% in multi/co-locate mode.
other issues	Difficulty of deployment, no counters/ monitoring/ alerting, no flexible retry setting, no slowing down/ pausing/ resuming, no job filtering.	All supported.

Table 4: Feedback from Hbase Compression operations team

ing makespan, minimizing mean completion time, maximizing throughput, fairness, or a combination of these; see [15, 19, 33] for good reviews of the topic. There are many open source and commercial schedulers for HPC and Grid workloads [4, 25, 29, 36].

Since MapReduce [18], data-intensive job scheduling has become the norm in the literature [10,13,22,26,28,32, 34, 38–40, 42, 43]. However, since MapReduce is an offline data processing framework, all schedulers of which we are aware assume no external resources needed and focus on interchangeable compute resources of the offline cluster. Typically the scheduler assigns tasks to workers from either a global queue or a set of queues corresponding to different jobs.

The *data locality* problem can be viewed as a special and simple case of our data resource constraints, where we want to place a task on the worker that hosts the data [6, 16, 28, 39, 42]. However, since queue-based scheduling is ill suited to non-interchangeable resources (Section 2.1), these schedulers treat data locality as a preference rather than a hard constraint. For example, Delay Schedule skips each job up to k times before launching its tasks non-locally [42]. A variation of the scheduling problem considers *task placement constraints*, where a task can only be assigned to a subset of workers due to dependencies on hardware architecture or kernel version [23, 32, 35]. Task placement constraints are associated with jobs, so we cannot use them to enforce data-local tasks. Mitigating stragglers or reducing job latency is another popular topic [8–10, 17, 20, 43].

Bistro moves one step further by scheduling data-parallel jobs against online systems directly. It treats resources at data hosts, either local or remote, as first-class objects, and can strictly enforce data locality and other hierarchical constraints without sacrificing scheduling performance. Many of its data-centric features are not common in the literature, e.g., tree-based scheduling, updating resources and jobs at runtime, flexible and elastic setup.

Regarding performance, Bistro is optimized for high throughput, handling highly concurrent short-duration tasks. Many schedulers assume long running tasks, and sacrifice scheduling delays for the optimal schedule. For example, Quincy takes about one second to schedule a task in a 2,500-node cluster [28]. In contrast, our Database Scraping workload has a turnover rate of thousands of tasks per second. Recently, Sparrow aimed at query tasks of millisecond duration, and reduced scheduling latencies for fast responses [32, 40]. Bistro can incorporate Sparrow in its Task Runner module to reduce the task dispatching latency.

6 Conclusion

Data center scheduling has transitioned from compute-intensive jobs to data-intensive jobs, and it is progressing from offline data processing to online data processing. We present a tree-based scheduler called Bistro that can safely run large-scale data-parallel jobs against live production systems. The novel tree-based resource model enables hierarchical resource constraints to protect online clusters, efficient updates to capture resource and job changes, flexible partitioning for distributed scheduling, and parallel scheduling for high performance. Bistro has gained popularity at Facebook by replacing Hadoop and custom-built schedulers in many production systems. We are in the process of migrating more jobs to Bistro, and plan to extend its resource and scheduling models in the next version.

7 Acknowledgements

We thank Terence Kelly, Haibo Chen, Sanjeev Kumar, Benjamin Reed, Kaushik Veeraraghavan, Daniel Peek, and Benjamin Wester for early feedback on the paper. Anonymous reviews helped us improve the paper further, and we appreciate our shepherd Haryadi Gunawi for working on the final version.

References

[1] Apache Thrift. `thrift.apache.org`.

[2] Bistro. `bistro.io`.

[3] Phabricator. `phabricator.org`.

[4] Platform LSF. `www.ibm.com`.

[5] RocksDB. `rocksdb.org`.

[6] Under the hood: Scheduling MapReduce jobs more efficiently with Corona. `www.facebook.com`.

[7] L. Abraham, , et al. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[8] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, 2013.

[9] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. In *NSDI*, 2014.

[10] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[11] K. Bandaru and K. Patiejunas. Under the hood: Facebook's cold storage system. `code.facebook.com`.

[12] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, 2010.

[13] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

[14] N. Bronson et al. Tao: Facebooks distributed data store for the social graph. In *Usenix Annual Technical Conference*, 2013.

[15] P. Brucker. *Scheduling algorithms*. Springer, 2007.

[16] F. Chung, R. Graham, R. Bhagwan, S. Savage, and G. M. Voelker. Maximizing data locality in distributed systems. *Journal of Computer and System Sciences*, 72(8):1309–1316, 2006.

[17] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job scheduling strategies for parallel processing*, pages 1–34. Springer, 1997.

[20] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *EuroSys*, 2012.

[21] L. George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, 2011.

[23] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[24] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[25] R. L. Henderson. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*, pages 279–294. Springer, 1995.

[26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[28] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[29] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[30] M. Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001.

[31] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebooks warm blob storage system. In *OSDI*, 2014.

[32] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, 2013.

[33] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[34] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[35] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Symposium on Cloud Computing*, 2011.

[36] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*, pages 307–350. MIT press, 2001.

[37] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *International Conference on Data Engineering*, 2010.

[38] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Symposium on Cloud Computing*, 2012.

[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop YARN: Yet another resource negotiator. In *Symposium on Cloud Computing*, 2013.

[40] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[41] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[42] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[43] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

Rubik: Unlocking the Power of Locality and End-Point Flexibility in Cloud Scale Load Balancing

Rohan Gandhi, Y. Charlie Hu, Cheng-Kok Koh
Purdue University

Hongqiang (Harry) Liu, Ming Zhang
Microsoft Research

Abstract

Cloud scale load balancers, such as Ananta and Duet are critical components of the data center (DC) infrastructure, and are vital to the performance of the hosted online services. In this paper, using traffic traces from a production DC, we show that prior load balancer designs incur substantial overhead in the DC network bandwidth usage, due to the intrinsic nature of traffic redirection. Moreover, in Duet, traffic redirection results in extra bandwidth consumption in the core network and breaks the full-bisection bandwidth guarantees offered by the underlying networks such as Clos and FatTree.

We present RUBIK, a load balancer that significantly lowers the DC network bandwidth usage while providing all the performance and availability benefits of Duet. RUBIK achieves its goals by applying two principles in the scale-out load balancer design – exploiting locality and applying end-point flexibility in placing the servers. We show how to jointly exploit these two principles to maximally contain the traffic load balanced to be within individual ToRs while satisfying service-specific failure domain constraints. Our evaluation using a testbed prototype and DC-scale simulation using real traffic traces shows that compared to the prior art Duet, RUBIK can reduce the bandwidth usage by over 3x and the maximum link utilization of the DC network by 4x, while providing all the performance, scalability, and availability benefits.

1 Introduction

Load balancing is a foundational function of modern data center (DC) infrastructures that host online services. Typically, each service exposes one or more virtual IPs (VIPs) outside the service boundary, but internally runs on hundreds to thousands of servers, each with a unique direct IP (DIP). The load balancer (LB) stores the VIP-to-DIP mapping, receives the traffic destined to each VIP, and splits it across the DIPs assigned for that VIP. Thus, the LB touches every packet coming from the Internet, as well as a significant fraction of the intra-DC traffic. For a 40K-server DC, LB is expected to handle 44 Tbps of traffic at full network utilization [21].

Such enormous traffic volume significantly strains the data plane of the LB. The performance and reliability of the load balancer directly affects the performance (throughput and latency) as well as availability of the online services within the DC. Recently proposed scale-out LB designs such as Ananta [21] and Duet [14] provide low cost, high scalability and high availability by distributing the load balancing function among Multiplexers (Muxes), either implemented in commodity servers called software Muxes (SMuxes) or existing hardware switches, called hardware Muxes (HMuxes).

However, such LB designs incur high bandwidth usage of the DC network because of the intrinsic nature of traffic redirection. First, even if the traffic source and the DIPs that handle the traffic are under the same ToR, the traffic first has to be routed to the Muxes, which may be faraway and elongate the path traveled by the traffic. Second, in both Ananta and Duet, the Muxes select DIPs for a VIP by hashing the five-tuple of IP headers, and hence are oblivious to DIP locations. As a result, even if the Mux and some DIPs are located nearby the source, the traffic can be routed to faraway DIPs in the DC, again traversing longer paths. Lastly, these designs do not leverage the server location flexibility in placing the DIPs closer to the sources to shorten the path.

The second problem with the Duet LB design is that the traffic detouring through core links breaks the full-bisection bandwidth guarantees originally provided by full-provisioned networks such as Clos and FatTree.

Our evaluation of traffic paths in a production DC network shows that such traffic detour significantly inflates the bandwidth usage of the DC network. This high bandwidth usage not only requires the DC operator to provision high network bandwidth which is costly, but also makes the network prone to transient congestion which affects latency-sensitive services.

In this paper, we propose RUBIK, a new LB that sig-

nificantly reduces the high bandwidth usage by LB. Like Duet, RUBIK uses a hybrid LB design consisting of the HMuxes and SMuxes, and aims to maximize the VIP traffic handled by HMuxes to reduce the LB costs. While doing that, RUBIK reduces the bandwidth usage using two synergistic design principles. First, RUBIK exploits the locality, *i.e.,* it tries to load balance VIP traffic generated within individual ToRs across the DIPs residing in the same ToRs. This reduces the total traffic entering the core network. Second, RUBIK exploits end-point flexibility, *i.e.,* it tries to place the DIPs for a VIP in the same ToRs as the sources generating the VIP traffic.

To exploit locality, RUBIK uses a novel architecture that splits the VIP-to-DIP mapping for a VIP into multiple "local" and a single "residual" mappings stored in different HMuxes. The local mapping stored at a ToR handles the traffic generated in the ToR across the DIPs in the same ToR. The residual mapping assigned to an HMux handles the traffic not handled by local mappings and maximizes the total VIP traffic handled by HMuxes.

To exploit locality and end-point flexibility, RUBIK faces numerous challenges. First, there are limited resources – individual switches have limited memory (where VIP-to-DIP mappings are stored) and individual ToRs have limited servers (where DIPs can be assigned). Also, individual DIPs (servers) have limited capacities. Exploiting end-point flexibility is further compounded as there are dependencies across services. The dependencies arise because many large services are multi-tiered; when a subservice at tier i receives a request, it spawns multiple requests to the subservices at tier $(i+1)$. Because of such dependencies, traffic sources at a lower tier are not known until DIPs in the higher tier are placed. Furthermore, RUBIK needs to ensure that it assigns DIPs that satisfy SLAs.

We develop a practical two-step solution to address all of the above challenges. In the first step, we design an algorithm to jointly calculate the DIP placement and mappings to maximize the traffic contained in ToRs while satisfying various constraints using an LP solver. In the second step, we use a heuristic assignment to maximize the total traffic handled by HMuxes to reduce the costs.

Lastly, to adapt to the cloud dynamics such as changes in the VIP traffic, failures, *etc.*, RUBIK regularly updates its local, residual mappings and DIP placement while limiting the number of servers migrated.

We evaluate RUBIK using a prototype implementation and DC-scale simulation using real traffic traces. Our results show that compared to the prior art Duet, RUBIK can reduce the maximum link utilization (MLU) of the DC network by over 4x and the bandwidth usage by over 3x, while providing the same benefits as Duet.

In summary, this paper makes the following contributions. (1) Through careful analysis of the LB workload from one of our production DCs, we show the high DC network bandwidth usage by recently proposed LB design Duet and Ananta. (2) We present the design and implementation of RUBIK that overcomes these inefficiencies by exploiting traffic locality and end-point flexibility. To the best of our knowledge, this is the first LB design that exploits these principles. (3) Through testbed experiments and extensive simulations, we show that RUBIK reduces the DC network bandwidth usage by 3x and the MLU by over 4x while providing a high performance and highly available LB.

2 Background

In this section, we briefly explain the LB functionality, workloads, and the Duet LB.

2.1 Load balancer

VIP indirection: A DC hosts thousands of online services, *e.g.,* news, sports [21, 14]. Each service exposes one or more virtual IPs (VIPs) outside the service boundary to receive the traffic. Internally, each service runs on hundreds to thousands of servers. Each server in this set has a unique direct IP (DIP) address. The task of the LB is to forward the traffic destined to a VIP of a service to one of the DIPs for that service. Such indirection provided by VIPs provides location independence: each service is addressed with a few persistent VIPs, which simplifies the management of firewall rules and ACLs, while behind the scene individual servers can be maintained or migrated without affecting the dependent service.

VIP traffic: In the Azure DC, 18-59% (average 44%) of the total traffic is VIP traffic which requires load balancing [21]. This is because services within the same DC use VIPs to communicate with each other to use the benefits provided by the VIP indirection. As a result, all incoming Internet traffic to these services (close to 30% of the total VIP traffic in our DC) as well as a large amount of inter-service traffic (accounting for 70% of the total VIP traffic) go through the LB. For a DC with 40k servers, LB is expected to handle 44 Tbps of traffic at full network utilization [21]. Such indirection of large traffic volume requires a scalable, high performance (low latency, high capacity) and highly available LB.

2.2 Workload Characteristics

We make the following observations about the VIP traffic being load balanced in our production DC, by analyzing a 24-hour traffic trace, for 30K VIPs.

The traffic sources and DIPs for individual VIPs are scattered over many ToRs. Fig. 1 shows the number of ToRs where the traffic sources and DIPs for the top 10% VIPs which generate 90% of the total VIP traffic are located. We see that traffic sources are widely scattered – the number of ToRs generating traffic for each VIP varies between 0-44.5% of the total ToRs. Also, the number of

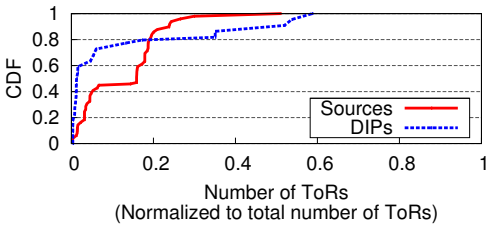


Figure 1: Distribution of the number of ToRs where the sources and DIPs are located.

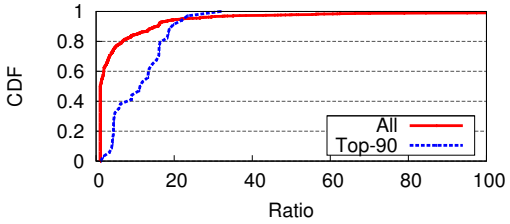


Figure 2: Ratio of 99^{th} percentile to average traffic volume for each VIP across all sources.

ToRs where the DIPs for a VIP are located varies between 0-58% of the total ToRs in the DC.

The traffic volume of sources per VIP are highly skewed. We measure the traffic from all the ToRs for each VIP. Figure 2 shows the CDF of the ratio of the 99^{th} percentile to the median per-ToR traffic volume for each VIP. We see that the source traffic volume for each VIP is highly skewed – the ratio varies between 1-35 (median 18) for the top VIPs generating 90% of the total traffic. The large skew happens for multiple reasons, including different numbers of servers, skew in the popularity of the objects that are served, and locality [17, 10].

VIP dependencies: Many large-scale web services are composed of multi-tier services, each with its own set of VIPs. When the top-level service receives a request, it spawns multiple requests to the services at the second tier, which in turn send requests to services at lower tiers. As a result, the VIP traffic exhibit hierarchical dependencies – the DIPs serving the VIPs at tier i become the traffic sources for the VIPs at tier $(i+1)$. We observe that 31.1% VIPs receive traffic from other VIPs. These VIPs employ 25.1% of the total DIPs and contribute to 27.6% of the total VIP traffic. The remaining 72.4% VIP traffic comes from the Internet, other DCs, and other servers in the same DC that are not assigned to any VIPs.

The dependency among the VIPs can be represented in a DAG. The depth of the DAG observed is similar to the depths reported by Facebook and Amazon [20].

2.3 Ananta LB

Ananta distributes the LB functionality among hundreds of commodity servers called software Muxes (SMuxes). Each SMux in Ananta stores VIP-to-DIP mapping for all the VIPs. When the VIP traffic hits one of the SMuxes, it selects the DIP based on the hash calculated over IP 5-tuple, and uses IP encapsulation to forward the VIP traffic to the selected DIP. Using SMuxes allows Ananta to be highly scalable, but is also costly, where supporting 15 Tbps VIP traffic for a 40K-server DC would require 4K SMuxes, and incurs high latency of 200 μsec to 1 msec for every packet handled by each SMux [14].

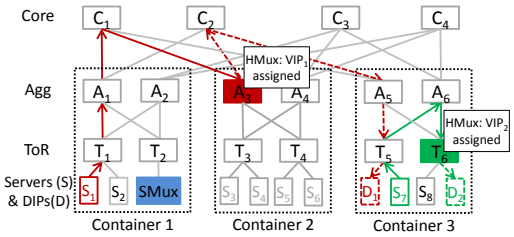


Figure 3: Duet architecture. Links marked with solid and dotted lines carry VIP and DIP traffic, respectively.

2.4 Duet LB

Duet [14] LB design consists of hardware switches and servers. Compared to Ananta, Duet lowers the LB cost by 12-24x and incurs a small latency of a few microseconds by using existing switches for load balancing. Duet runs hardware Mux (HMux) on every switch that stores the VIP-to-DIP mapping in the switch (ECMP) memory, splits the traffic for a VIP among its DIPs based on the hash value calculated over the 5-tuple in the IP header, and sends the packet to the selected DIP by encapsulating the packets using the tunneling table available in the switch.

However, switches have limited hardware resources, especially the routing and tunneling table space. The tunneling table size (typically 512 entries) limits the total number of DIPs (for multiple VIPs) that can be stored on a single HMux. Accordingly, Duet partitions the VIP-to-DIP mappings across HMuxes, where the mappings for a small set of VIPs are assigned to each HMux. This way of partitioning enables Duet to support a large number of DIPs. Second, the routing table size (typically 16K entries) per switch limits the total number of VIPs that can be supported in HMuxes. Therefore, Duet uses HMuxes to handle up to 16K *elephant VIPs*. The remaining mice VIP (that could not be assigned to any of the HMuxes) traffic is handled by deploying a small number of SMuxes, which have the same design as in Ananta. These SMuxes also load balance traffic otherwise handled by HMuxes during HMux failures which enables Duet to provide high availability.

3 Motivation

We next assess the impact of the VIP traffic characteristics (§2.2) on the DC network bandwidth usage under

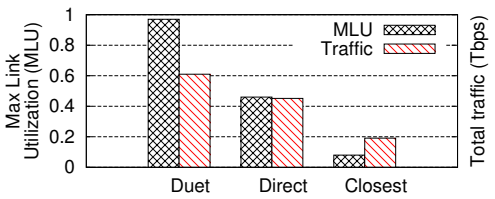


Figure 4: MLU and total traffic under various LB schemes. Total traffic is measured across all the DC network links.

Duet	Direct	Closest
5.47	3.94	1.78

Table 1: Path length for different LB designs.

the Duet LB. We simulate how Duet handles the VIP traffic using a 24-hour traffic trace from our production DC on a network topology that closely resembles our production DC. The topology, workload, and results are detailed in §10. Duet maximizes the total traffic handled by HMuxes, on average 97% in the 24-hour period.

High link utilization. Figure 4 shows the MLU and total traffic in the DC network[1]. While Duet is able to handle 97% of the total VIP traffic by leveraging HMuxes, it also inflates the MLU to 0.98 (or 98%). This high MLU can be explained by two design decisions of Duet.

First, Duet assigns a VIP only to a single HMux. But the traffic sources and DIPs for individual VIPs are spread in a large number of ToRs (Figure 1). The diverse location of traffic sources and DIPs per VIP suggests *no matter where the single Mux for a VIP is positioned in the network, it will be far away from most of the traffic sources and DIPs for that VIP*, and hence most VIP traffic will traverse through the network to reach the HMuxes and then the DIPs, which inflates the path length between the sources and DIPs.

Table 1 shows that the average number of hops between the sources and the DIPs across all individual VIPs is 5.47 in Duet. Notice that the traffic between two hosts that does not go through the LB would have a maximum of 4 hops (ToR-Agg, Agg-Core, Core-Agg, Agg-ToR). Thus the average path length of 5.47 in Duet indicates that most traffic goes through the core links and further experiences some detour in the DC network. Figure 3 shows an example where the VIP-1 traffic originated at S_1 has to travel 6 hops to reach DIP D_1 – 3 hops to reach the HMux at switch A_3, and 3 more hops to reach D_1.

To dissect the impact of the redirection, we measure the MLU and total traffic in the DC network in a hypothetical case where the HMuxes are located on a direct path between the sources and DIPs, labeled as "Direct". Figure 4 shows that in this case the MLU is reduced to 0.46 (from 0.98 in Duet), and the bandwidth used is lowered by 1.36x, compared to Duet. Also, the average path length in "Direct" is lowered to 3.94 (1.38x improvement). This means the redirection design in Duet inflates the MLU by 2.13x and bandwidth used by 1.36x.

The second cause for the high link utilization is location-oblivious DIP selection in Duet. The HMux splits the VIP's traffic by hashing on the 5-tuples in the IP header, and chooses the DIP based on the hash. Thus, even if there is a DIP located under the same ToR as the HMux and has the capacity to handle all the local traffic for the VIP, the HMux will spread the local traffic among all DIPs, many of which can be far away in the DC.

To measure the impact of location-oblivious DIP selection, we measure the MLU and bandwidth used in a hypothetical case, where the traffic from the individual sources is routed to the closest DIP and assuming the HMuxes lie on the path. This mechanism is labeled as "Closest". Figure 4 shows that the MLU is reduced to just 0.08, and the bandwidth used reduces by 3.19x compared to Duet. Also, the average path length is lowered to just 1.78 hops.

Effective full bisection bandwidth reduced at core. Many DC networks have adopted topologies like FatTree and Clos [15] to achieve full-bisection bandwidth. Such networks guarantee that there is enough aggregate capacity between Core and Agg switches as between Agg and ToR switches, and hence the core links will never become a bottleneck for any traffic between the hosts.

However, traffic indirection can break this assumption, if the HMuxes reside in Agg or ToR switches. This happens to Duet, as Duet considers all the switches while assigning VIP-to-DIP mappings. This is illustrated in Figure 3. When VIP_1 is assigned to an Agg switch (A_3), the traffic from source S_1 travels the core links twice enroute to DIP D_1 – first to get to HMux A_3, and then to D_1. In contrast, direct host-to-host traffic only has to traverse core links at most once. As a result, the effective bandwidth in the core links is reduced – in Figure 3, the available bandwidth to container-2 (servers S_3-S_6) is reduced due to the LB traffic among other containers.

Our evaluation in §10.3 shows the traffic overhead in Duet, *i.e.,* the ratio of the additional traffic due to redirection to the total traffic without redirection is 44% in core links and 16% in containers. This means the remaining bisection bandwidth of the Agg-Core links is lower than the remaining bisection bandwidth in the ToR-Agg links. This breaks the full-bisection guarantee provided by the FatTree or Clos, which jeopardizes other applications that co-exist in the DC and assume full-bisection bandwidth is available (*e.g.,* [12, 23]).

4 RUBIK Overview

In the previous section, we saw that the traffic indirection in Duet incurs substantial overhead in the DC net-

[1] Absolute values for "total traffic" are omitted for confidentiality.

work bandwidth usage. In this paper, we propose a new LB design, RUBIK, that significantly reduces the bandwidth usage in the DC network while providing low cost, high performance and high availability benefits.

RUBIK is based on two key ideas motivated by the observations in the last section. First, it exploits *locality*, *i.e.,* it tries to load balance traffic generated in individual ToRs across the DIPs present in the same ToRs. In this way, a substantial fraction of the load balanced traffic will not enter the links beyond ToRs which reduces the DC network bandwidth usage and MLU.

The second key idea of RUBIK is to exploit *DIP placement flexibility* to place DIPs closer to the sources. In RUBIK online services specify the number of DIPs for individual VIPs, and RUBIK decides the location of the servers to be assigned to individual VIPs. This idea is synergistic with the first idea, as it facilitates exploiting locality in load balancing within ToRs.

Realizing the two ideas is challenging, because (1) there are a limited number of servers in each ToR where DIPs can be assigned, (2) switches have limited memory for storing VIP-to-DIP mappings, (3) a VIP may have traffic sources in more ToRs than the total number of DIPs for that VIP. In such a case, a DIP cannot be assigned in every ToR that has traffic sources, (4) dependencies between the VIPs make it even harder, as the sources to some of the VIPs are not known until DIPs for other VIPs are placed.

RUBIK addresses the above capacity limitations (switch memory and DIPs in a ToR) using two complimentary ideas. First, RUBIK uses a new LB architecture that splits the VIP-to-DIP mapping for a VIP across multiple HMuxes to *enable* efficient use of switch memory while containing local traffic. Second, it employs a novel algorithm that *calculates* the most efficient use of switch memory for containing the most local traffic.

5 RUBIK Architecture

RUBIK uses a new LB design that splits the VIP-to-DIP mapping for each VIP into *multiple local and a single residual VIP-to-DIP mappings*. This idea is inspired by the observation that the traffic for individual VIPs is skewed (§2.2) – some ToRs generate more traffic than other ToRs for a given VIP. In RUBIK, we assign local mappings to the ToRs generating large fractions of the traffic and also assign enough DIPs to handle those traffic. The local mapping for a VIP load balances traffic for that VIP across the DIPs present under the same ToR (called local DIPs). We then assign a single residual mapping for that VIP to handle the traffic from all the remaining ToRs, where no local mapping is assigned.

Effectively, the VIP-to-DIP mapping for a VIP is split across the local and residual mappings such that a single DIP appears in only one mapping. Assigning a DIP only once makes the most efficient use of the limited tunneling table space of HMuxes so the total VIP traffic handled by the HMuxes can be maximized. The assignment module (§6) then calculates the actual assignment that maximizes the VIP traffic handled locally.

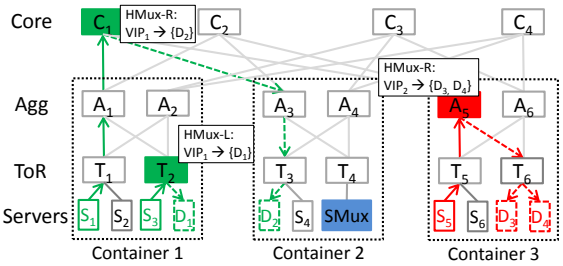


Figure 5: RUBIK Architecture. DIPs for VIP_1 are split in local (HMux-L) and residual mappings (HMux-R).

We now explain the RUBIK architecture in detail.

Local mapping. If some of the sources and DIPs for a VIP already reside in the same ToR, RUBIK exploits this locality by load balancing the source traffic across those local DIPs. To ensure that the traffic does not flow outside the ToR in detouring through the HMux, RUBIK stores a subset of the VIP-to-DIP mapping, *i.e.,* containing only the local DIPs, at the ToR itself (*e.g.,* HMux T_2 in Fig. 5). We denote such a mapping containing the subset of local DIPs as a *local mapping*.

Residual mapping. For an individual VIP, we assign a single residual mapping to handle the remaining traffic not handled by the local mappings (called residual traffic). We pool all the remaining DIPs for a VIP together in a single DIP-set, called the *residual mapping* for that VIP (*e.g.,* HMux on C_1 and A_5 in Fig. 5). The residual mapping for each VIP announces the VIP using BGP so that other routers (or switches) route the VIP traffic to the HMux where its residual mapping is assigned.

In principle, we can replicate the residual mapping at all the ToRs containing any remaining traffic sources. Such replication can reduce the number of hops between the sources and HMux, but it can also consume a significant amount of the limited tunneling table space. Therefore, we only assign the residual mapping for a VIP to a single HMux, and the optimal choice of HMux to store the residual mapping of a VIP depends on the location of the remaining traffic sources and residual DIPs.

VIP routing. The above DIP-set splitting design has one potential problem. If the HMuxes storing either the local mappings or the residual mapping of a VIP all announce the VIP via BGP to the network, some of the residual source traffic may be routed towards the HMuxes storing local mappings if they are closer than the HMux storing the residual mapping. This would significantly complicate the DIP placement, and DIP-set splitting and placement problem. We avoid this complication by making the

HMuxes storing local mappings *not* announce the VIP via BGP. In this way, only local source traffic within a ToR sees the local mapping and is split to the local DIPs.

SMuxes. Because of the limited switch memory, the numbers of VIPs and DIPs supported by HMuxes remain limited. Current HMuxes can support up to 16K VIPs [14], and our DC has 30K+ VIPs. Also, it remains challenging to provide high availability during HMux failures. We address both problems by deploying a small number of SMuxes as a backstop, to handle the VIP traffic that could not be handled using HMuxes. We also announce all the VIPs from all SMuxes. We use Longest prefix matching (LPM) to: (1) preferentially route the VIP traffic to the HMuxes for the VIPs assigned to both HMuxes and SMuxes, (2) route the traffic to the remaining VIPs not assigned to HMuxes to the SMuxes.

The use of SMuxes in this way also provides high availability during residual mapping failure. §7 gives details on how RUBIK recovers from a variety of failures.

Summary. The benefits of this architecture can only be realized by carefully calculating the DIP placement, and local and residual mappings for individual VIPs subject to a variety of constraints, which we describe next.

6 Joint VIP and DIP Assignment

RUBIK's objective is to maximize the traffic handled by the HMuxes, while maximizing the traffic handled locally within ToRs. The assignment algorithm determines for each VIP, (1) the location of its DIPs; (2) the number of DIPs in each ToR in the local VIP-to-DIP mapping; and (3) the number of DIPs in the residual mapping, and the HMux assigned to store the mapping.

RUBIK needs to calculate this assignment such that the capacity of all resources (switch tables, links, and servers per ToR) is not be exceeded. Also, RUBIK needs to ensure that it assigns DIPs in the failure domains (*i.e.,* ToRs) specified by the online services. The placement calculated at a given time may lose effectiveness over time as the VIP traffic changes, and VIPs and DIPs are added and removed. To adapt to such cloud dynamics, RUBIK reruns the placement algorithm from time to time. While calculating a new assignment, RUBIK has to ensure that the number of machines migrated from the old assignment is under the limit.

The assignment problem is a variant of the bin-packing problem (NP-hard [11]), where the resources are the bins, and the VIPs are the objects. It is further compounded because the VIP traffic exhibits hierarchical dependencies (§2.2).

To reduce the complexity, RUBIK decomposes the joint assignment problem into two independent modules, (1) *DIP and local mapping placement*, (2) *residual mapping placement*, as shown in Algorithm 1. The first module places the DIPs and local mappings for all the VIPs to maximize the total traffic load-balanced locally on individual ToRs. We calculate DIP and local mapping simultaneously, because the problem of DIP and local mapping placement are intertwined, as the traffic for a VIP is contained within a ToR only if the ToR has (1) enough DIPs to handle the traffic, and (2) enough memory to store the corresponding VIP-to-DIP mapping.

Algorithm 1: RUBIK Assignment Algorithm

```
1 Input: V, M, N_v, f_v, S, L, b_{t,v}, C_{t,v}
2 Output: x^D_{t,v}, x^M_{t,v}
3 topological_sort(V in DAG)
4 for l = 1, depth of DAG do
5 |   local_mapping_and_dip_placement(VIPs in
  |   DAG_level(l))
6 end
7 residual_mapping_placement()
```

Notation	Explanation
Input	
S, L, V	Sets of switches, links, and VIPs
M_t	# servers under t-th ToR
T_s	Table capacity of s-th switch
L_e	Link traffic capacity of link e
N_v, f_v	#DIPs and failure-domain for v-th VIP
$b_{t,v}$	Traffic sent to v-th VIP from t-th ToR
$C_{t,v}$	Traffic capacity of server in t-th ToR when assigned to v-th VIP
Variables	
$x^D_{t,v}$	Number of servers (DIPs) in t-th ToR assigned to v-th VIP
$x^M_{t,v}$	Number of table entries in t-th ToR assigned to v-th VIP

Table 2: Notations used in the algorithm.

Since the VIP traffic exhibits hierarchical dependencies (§2.2), we create a DAG that captures the traffic flow and hence the dependency between the VIPs, and then perform a topological sort on the DAG to divide the VIPs into different levels. We then place the DIPs and local mappings for the VIPs level-by-level (lines 3:6 in Algo. 1). As we place DIPs for VIPs in one level, the sources in the next level become known.

The second module places the residual mappings of all the VIPs to maximize the total traffic handled by HMuxes. The residual mapping placement subproblem remains NP-hard. But, the residual VIP traffic is typically only a small portion of the total traffic and hence we can apply heuristics to solve it without significantly affecting the quality of the overall solution (line 7).

6.1 DIP and Local Mapping Placement

The first module places the DIPs and the local mappings of all the VIPs for which the sources are known such that the total VIP traffic load balanced within the

ToRs is maximized. We formulate the joint DIP and local mapping placement problem as ILP using notations shown in Table 2 as follows.

Input: The input includes (1) the network topology and resource information (capacity of switch tables, links, and servers in the ToRs), (2) for every VIP in current level, the number of DIPs and number of failure domain and traffic, and (3) max. number of DIPs to migrate (δ).

Output/Variables: The output includes the local VIP-to-DIP mappings on individual ToRs, and placement of all the DIPs (including residual DIPs), for all VIPs.

Let $x^D_{t,v}$ denote the number of machines in the t-th ToR assigned as the DIPs for the v-th VIP, and $x^M_{t,v}$ denote the number of machines out of these $x^D_{t,v}$ machines that are used in the local mapping for the VIP, *i.e.,* they will appear in the local VIP-to-DIP mapping of the t-th ToR.

Objective:

maximize Locality $L = \sum_{v\in V}\sum_{t\in T} y^M_{t,v}\cdot b_{t,v}$

where $y^D_{t,v}$ is set if there are any DIPs in the t-th ToR assigned to the v-th VIP, and $y^M_{t,v}$ is set if the t-th ToR switch (HMux) contains local VIP-to-DIP mapping for the v-th VIP. This way, $y^M_{t,v}\cdot b_{t,v}$ denotes if traffic for v-th VIP in t-th ToR is handled locally, and we maximize traffic handled locally across all VIPs and ToRs.

$$y^M_{t,v} = \begin{cases} 1 & x^M_{t,v} \geq 1 \\ 0 & \text{Otherwise} \end{cases} \qquad y^D_{t,v} = \begin{cases} 1 & x^D_{t,v} \geq 1 \\ 0 & \text{Otherwise} \end{cases}$$

Constraints:

(1,2) Switch table size and number of servers not exceeded on every ToR

$$\forall t \in T, \sum_{v\in V} x^M_{t,v} \leq T_t, \quad \sum_{v\in V} x^D_{t,v} \leq M_t$$

(3,4) Specified number of DIPs assigned for every VIP; failure domain constraints

$$\forall v \in V, \sum_{t\in T} x^D_{t,v} = N_v, \quad \sum_{t\in T} y^D_{t,v} \geq f_v$$

(5a, 5b) DIPs are not overloaded (no hot-spots)

$$\forall t \in T, \forall v \in V, y^M_{t,v}\cdot b_{t,v} \leq x^M_{t,v}\cdot C_{t,v}$$

$$\forall v \in V, \sum_{t\in T}(1-y^M_{t,v})\cdot b_{t,v} \leq \sum_{t\in T}(x^D_{t,v}-x^M_{t,v})\cdot C_{t,v}$$

Constraint (5a) ensures the DIPs mapped in the local mapping are not overloaded. Constraint (5b) ensures the DIPs in the residual mapping are not overloaded.

(6) Limiting the number of DIP moves

$$\sum_{v\in V, t\in T} |x^M_{t,v} - x^{M,old}_{t,v}| \leq \delta$$

where $x^{M,old}_{t,v}$ denotes the number of DIPs in the ToR in the previous assignment, and δ is the threshold on the maximum number of DIPs to be moved. We convert constraint (6) into the linear form as:

$$\sum_{v\in V, t\in T} z_{t,v} \leq \delta$$

$$\forall t \in T, \forall v \in V, z_{t,v} \geq x^M_{t,v} - x^{M,old}_{t,v}, z_{t,v} \geq x^{M,old}_{t,v} - x^M_{t,v}$$

(7) ToRs have more DIPs than in local mappings

$$\forall v \in V, t \in T, x^D_{t,v} \geq x^M_{t,v}$$

(8a,8b) Writing $y^M_{t,v}$, $y^D_{t,v}$ in linear form

$$\forall t \in T, \forall v \in V, 0 \leq y^M_{t,v}, y^D_{t,v} \leq 1, y^M_{t,v} \leq x^M_{t,v}, y^D_{t,v} \leq x^D_{t,v}$$

6.2 Residual Mapping Placement

The second module places the residual mappings for the VIPs among the switches while maximizing the total VIP traffic load balanced by the residual mapping HMuxes (traffic not handled by local mappings), subject to switch memory and link capacity constraints.

This assignment problem is the same as that in Duet, and we solve it using the same heuristic algorithm as in Duet. Briefly, to assign the VIPs, we first sort the VIPs in decreasing traffic volume, and attempt to assign them one by one. We define the notion of maximum resource utilization (MRU). MRU represents the maximum utilization across all resources – switches and links. To assign a given VIP, we consider all switches as candidates. We calculate the MRU for each assignment, and pick the one that results in the smallest MRU, breaking ties at random. If the smallest MRU exceeds 100%, *i.e.,* no assignment can accommodate the traffic of the VIP, the algorithm terminates. The remaining VIPs are not assigned to any switch – their traffic will be handled by the SMuxes.

7 Failure Recovery

A key requirement of the LB design is to maintain high availability during failure: (1) the traffic to any VIP should not be dropped, (2) existing connections should not be broken. As in Duet, RUBIK relies on SMuxes to load balance the traffic during various failures. In addition to storing VIP-to-DIP mapping for all the VIPs, we use the ample memory on individual SMuxes to provide connection affinity by maintaining per-connection state.

Residual mapping HMux failure: Failure of the HMux storing the residual mapping of a VIP only affects the traffic going to that HMux; the traffic handled by other local and residual mappings is unaffected. The routing entries for the VIPs assigned to the failed HMux are removed from all other switches via BGP withdraw messages. After routing convergence, traffic to these VIPs is routed to the SMuxes, which announce all VIPs. Since each SMux stores the same residual DIPs and uses the same hash function as the residual mapping HMux to select a DIP, existing connections are not broken.

Local mapping failure: When a ToR switch fails, all the sources and DIPs for a VIP under it are also discon-

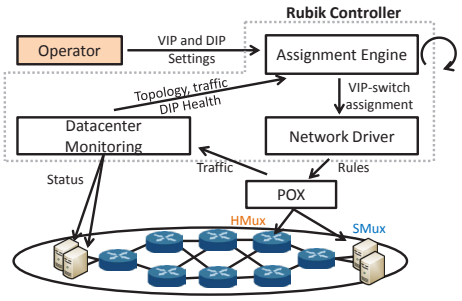


Figure 6: RUBIK implementation.

nected. As a result, the traffic the local mapping was handling also disappears. Further, the rest of the traffic for that VIP continues to be routed to the residual mapping or other local mappings, and are not affected.

SMux failure: On an SMux failure, traffic going to that SMux is rerouted to the remaining SMuxes using ECMP. The connections are not broken as all the SMuxes use the same hash function.

DIP failure: Existing connections to the failed DIP would necessarily be terminated. For VIPs whose mapping are assigned to SMuxes, connection to the remaining DIPs are maintained as SMuxes use consistent hashing in DIP selection [21]. For VIPs assigned to HMuxes, the connections are maintained using smart hashing [2].

8 Implementation

We briefly describe the implementation of the three building blocks of RUBIK, (1) RUBIK controller, (2) network driver, (3) HMux and SMux, as shown in Figure 6.

RUBIK **controller:** The controller orchestrates all control activities in RUBIK. It consists of three key modules: (1) DC monitor, (2) Assignment engine, (3) Network driver. The DC monitor periodically captures the traffic and DIP health information from the DC network and sends it to the assignment engine. The assignment engine calculates the DIP placement, local and residual VIP-to-DIP mappings for all the VIPs, and pushes these new assignment to the network driver. We use CPLEX [7] to solve the LP (§6.1).

Network driver: This module is responsible for maintaining VIP and DIP traffic routing in the LB. Specifically, when the VIP-to-DIP assignment changes, the network driver announces or withdraws routes for the changed VIPs according to BGP.

HMux and SMux: We implement HMuxes and SMuxes using Open vSwitches that split the VIP traffic among its DIPs using ECMP based on the source addresses [5]. We implement smart hashing [2] using OpenFlow rules. The replies from the DIPs directly go to the sources using DSR [21].

Lastly, we use POX to push the rules and poll the traffic statistics. We developed a separate module to monitor the DIP health. The code for all the modules consists of

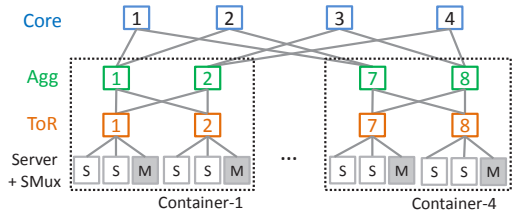


Figure 7: Our testbed. FatTree with 4 containers connected to 4 Core switches.

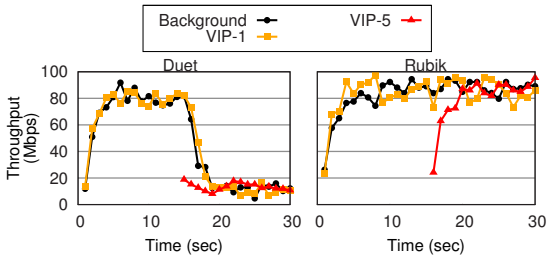


Figure 8: RUBIK reduces congestion.

3.4K LOC in C++ and Python.

9 Testbed

Setup: We evaluate RUBIK prototype using Open vSwitches and Mininet. Our testbed (Fig. 7) consists of 20 switches (HMuxes) in 4 containers connected in a FatTree topology. Each ToR contains an SMux (marked "M") and 2 hosts that can be set as DIPs (marked "S").

Services: We evaluate the performance of RUBIK using two services that require load balancing: (1) HTTP web service, (2) Bulk data transfer service. The web service serves static web pages of size 1KB and generates a large number of short-lived TCP flows. The Bulk data transfer service receives a large amount of data using a small number of long-lived TCP flows. All the servers and clients for these services reside in the same DC.

Experiments: Our testbed evaluation shows: (1) RUBIK lowers congestion in the network; (2) RUBIK achieves high availability during a variety of failures – local mapping, residual mapping, and DIP failure.

9.1 Reduction in Congestion

First we show that RUBIK reduces congestion in the DC network by using local mappings. In this experiment, initially 4 VIPs (each with 1 source and 1 DIP) are assigned to 4 different HMuxes. Additionally, there is background traffic between 2 hosts. Figure 8 shows the per-second throughput measured across 2 flows. "VIP-1" denotes the throughput for one of the 4 VIPs added initially. "Background" denotes the throughput for the background flow (not going through the LB). Initially, there is no congestion in the network and as a result all flows experience high throughput. At time 15 sec, we add a new VIP (VIP-5) that has 2 DIPs and 2 sources sending equal volume of traffic, and assign it using Duet.

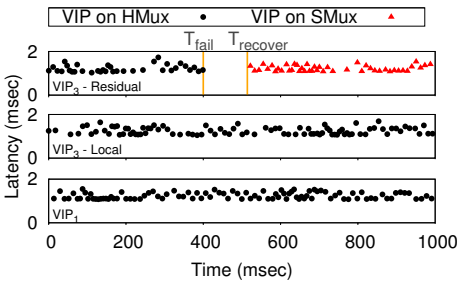


Figure 9: VIP availability when residual mapping fails.

However, assigning the new VIP causes congestion as the new flows compete with the old flows. As a result, the throughput for all the flows drop by almost 5-6x.

We repeat the same experiment with RUBIK. At time 15 sec, we assign the VIP-5 using RUBIK. RUBIK assigns local mappings to handle the VIP-5 traffic. As a result, adding VIP-5 does not cause congestion (no drop in throughput), as shown in Figure 8. This experiment shows that by exploiting locality, RUBIK reduces the congestion and improves the throughput by 5-6x.

9.2 Failure Mitigation

Next we show how RUBIK maintains high availability during various failures.

Residual mapping failure: Fig. 9 shows the availability of the VIP, measured using ping latency, when its residual mapping fails. In this experiment, we have 3 VIPs (VIP-1, 2, 3) assigned to the data-transfer service. VIP-1 and VIP-2 have one source and one DIP each in different ToRs, and their traffic is handled by residual mappings (no local mapping). VIP-3 has two sources and two DIPs. One source and one DIP are in the same ToR – the local mapping on that ToR handles their traffic. The remaining source and DIP are in two different ToRs, and their traffic is handled by the residual mapping.

At 400 msec, we fail the HMux storing the residual mapping for VIP-3. We make four observations: (1) On HMux failure, VIP-3 traffic handled by it is lost for 114 msec. (2) After 114 msec, VIP-3 is 100% available, *i.e.,* all of the pings are successful again. During this time, the routing converges, and the traffic that used to go to the HMux is rerouted to the SMuxes. (3) The traffic for VIP-3 handled by the local mapping (shown as VIP-3-Local) is not affected – no ping message is dropped. (4) Other VIPs (only VIP-1 is shown) are not affected – their ping messages are not dropped.

This shows that RUBIK provides high availability during residual mapping failure.

Local mapping failure: Figure 10 shows the impact of local mapping failure on the availability of the VIPs. We use the same setup as before, and fail the HMux where VIP-3's local mapping was assigned. We measure the ping message latency from 2 sources for VIP-3 (denoted as Client-1, 2). The traffic from Client-2 is handled locally, whereas Client-1 traffic is handled by the residual mapping. When local mapping fails (at 500 msec), all the sources and DIPs under it disappear. Therefore, ping messages for Client-2 are lost as Client-2 itself is down. Figure 10 shows that the traffic from Client-1, which is handled by the residual mapping, is not affected.

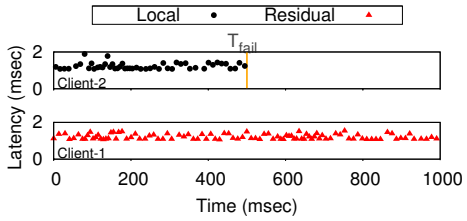


Figure 10: VIP availability during local mapping failure.

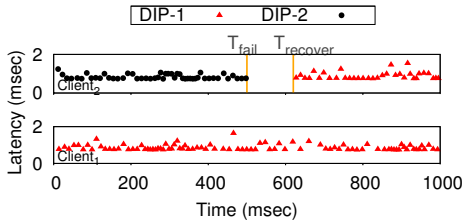


Figure 11: VIP availability during DIP failure.

DIP failure: Lastly, we evaluate the impact of DIP failure on service availability. In this experiment, we use a single VIP with 2 sources (Client-1, 2) and 2 DIPs (DIP-1, 2), located in different ToRs. Therefore, both DIPs are assigned to the residual mapping. Initially, the traffic from Client-1 is served by DIP-1 and that of Client-2 is served by DIP-2. We fail DIP-2 at 500 msec.

Figure 11 shows the latency for the ping messages from Client-1 and Client-2. When DIP-2 fails, the ping messages for Client-2 are lost for about 120 msec. After 120 msec, Client-2 traffic is served by DIP-1. This is because when DIP-2 fails, the residual mapping is adjusted using smart-hashing, *i.e.,* the traffic going to the failed DIP is split across the remaining DIPs. As a result, the traffic going to DIP-2 is now served by DIP-1. It can also be seen that Client-1 traffic is not affected – there is no drop in the ping messages. This shows that a DIP failure does not affect the traffic going to other DIPs, and traffic going to the failed DIP is spread across remaining DIPs.

10 Simulation

In this section, we use large-scale simulations of RUBIK and Duet to show: (1) RUBIK handles a large percentage of traffic in HMuxes as in Duet but incurs significantly lower maximum link utilization (MLU); (2) RUBIK reduces the traffic in the core by 3.68x and in the container by 3.47x; (3) RUBIK contains 63% of VIP traffic within ToRs; (4) RUBIK does not create hotspots.

Network: Our simulated network closely resembles

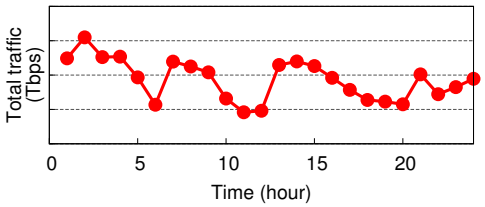


Figure 12: Total traffic variation over 24 hours.

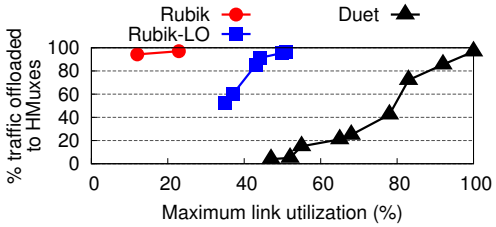


Figure 13: Traffic handled by HMuxes vs. MLU.

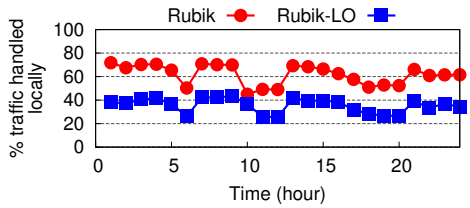


Figure 14: Traffic handled using local mappings.

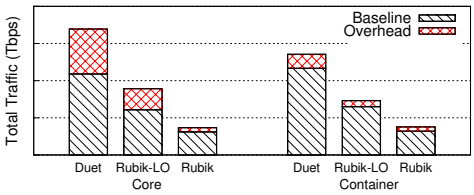


Figure 15: Total traffic in core and container.

that of a production DC, with a FatTree topology connecting 500K VMs under 1600 ToRs in 40 containers. Each container has 40 ToRs and 4 Agg switches, and the 40 containers are connected with 40 Core switches. The link and switch memory capacity were set with values observed in the production DC.

Workload: We run the experiments using traffic trace collected from the production DC over a 24-hour duration. The trace consists of the number of bytes sent between all sources and all VIPs. Figure 12 shows the total traffic per hour fluctuates over the 24-hour period[2].

Comparison: We compare the performance of Duet, RUBIK-LO and RUBIK. Duet exploits neither locality nor DIP placement. RUBIK-LO is a version of RUBIK that only exploits locality without moving the DIPs; it assumes DIP placement is fixed and given, and only calculates the local and residual mappings. RUBIK exploits both locality and flexibility in moving the DIPs. RUBIK performs stage-by-stage VIP-to-DIP mapping assignment following the VIP dependency.

10.1 MLU Reduction

We first compare the trade-off between the MLU and fraction of the traffic handled by the HMuxes under the three schemes. Note that all three schemes try to maximize the total traffic handled by HMuxes. The traffic not handled by HMuxes is handled by SMuxes.

Figure 13 shows the fraction of traffic handled by HMuxes under the three schemes. The MLU shown is the total MLU which resulted from load balancing all VIP traffic, handled by HMuxes and by SMuxes. We see that Duet can handle 97% traffic using HMuxes, but incurs a high MLU of 98%. But when MLU is restricted to 47%, Duet can only handle 4% traffic using HMuxes.

In contrast, RUBIK-LO handles 97% VIP traffic using HMuxes at MLU of 51%. It handles 52% of VIP traffic using HMuxes at MLU of 35%. This improvement over Duet comes purely from exploiting locality.

Lastly, RUBIK significantly outperforms both Duet and RUBIK-LO. It handles 97% traffic with a low MLU of 22.9%, a 4.3x reduction from Duet. Also, at a MLU of 12%, RUBIK handles 94% traffic using HMuxes.

10.2 Traffic Localized

RUBIK significantly reduces the MLU by containing significant amount of traffic within individual ToRs. Figure 14 shows the fraction of the total traffic contained within ToRs in RUBIK and RUBIK-LO over the 24-hour period, where these mechanisms calculate new assignment every hour. In RUBIK, we limit the machine moves to 1% based on the trade-off detailed in §10.5.

We see that RUBIK-LO localizes 25.5-43.4% (average 34.8%) of the total traffic within ToRs, and RUBIK localizes 46-71.8% (average 63%) of the total traffic within ToRs. Additionally, for the VIPs generating 90% of the total VIP traffic, we find that, the local mappings handle traffic from 37.8-48.6% (average 41.8%) sources, and 50.2-57.7% (average 53%) of the total DIPs are assigned to their local mappings.

10.3 Traffic Reduction

Figure 15 shows the total bandwidth usage across all the links caused by the VIP traffic under the three mechanisms. We separately show the total traffic on the core links (between Core and Agg switches) and containers links (between ToR and Agg switches). The total traffic shown is the average over 24 hours. Furthermore, we break down the total traffic into baseline and overhead due to redirection. The baseline traffic shows the amount of traffic generated if the HMuxes were on the direct path between source and DIPs, which would cause no redirection. The remaining traffic is the extra traffic due to the

[2] Absolute values are omitted for confidentiality.

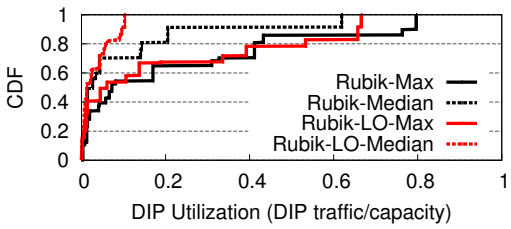


Figure 16: DIP utilization distribution across VIPs.

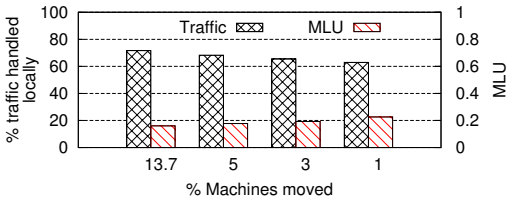


Figure 17: Impact of machine moves.

redirection to route traffic to and from HMuxes.

RUBIK and RUBIK-LO significantly reduce the total traffic in the core network and containers. Compared to Duet, on average RUBIK-LO reduces the total traffic by 1.94x and 1.88x, respectively. RUBIK reduces the total traffic by 3.68x and 3.47x, respectively.

Secondly, RUBIK-LO and RUBIK reduce the traffic overhead due to traffic redirection by 2.1x and 10.9x compared to Duet. It should be noted that both RUBIK and RUBIK-LO cannot eliminate the traffic overhead, because they cannot localize 100% of the VIP traffic. As a result, the traffic not localized is handled by the HMuxes storing residual mappings, which causes traffic detour.

10.4 DIP Load Balance

To exploit locality, RUBIK partitions the DIPs for a VIP into local and residual DIP-sets, which can potentially overload some of the DIPs (hotspots). We calculate the average and peak DIP utilization (DIP traffic/capacity) across all DIPs for every VIP. Figure 16 shows the CDF across all VIPs in RUBIK and RUBIK-LO. It shows that both schemes ensure that the peak utilization for all the DIPs is well under 80%, which is the constraint given to the assignment algorithm. Furthermore, for 80% VIPs, the peak utilization is under 40%. This shows RUBIK does not create hotspots.

10.5 Impact of Limiting Machine Moves

Lastly, we evaluate the impact of limiting machine moves in RUBIK's assignment LP formulation (§6.1) on the fraction of traffic localized and MLU. Figure 17 shows the two metrics as we reduce the percentage machine moves allowed. Without any restriction, RUBIK assignment results in moving 13.7% of the DIPs. When the percentage machine moves is 1%, the fraction of traffic localized decreases by 8.7% whereas the MLU increases by 6.6%, and the execution time to find the solution increases by 2.3x compared to unrestricted machine moves. This shows that most of the benefits of RUBIK are maintained after restricting the machine moves to just 1%. We therefore used this threshold in all the previous simulations and testbed experiments.

11 Related work

To our best knowledge, RUBIK is the first LB design that exploits locality and end-point flexibility. Below we review work related to DC LB design which has received much attention in recent years.

LB: Traditional hardware load balancers [4, 1] are expensive and typically only provide 1+1 availability. We have already discussed Duet [14] and Ananta [21] load balancers extensively. Other software-based load balancers [6, 8, 9, 3] have also been proposed, but they lack the scalability and availability of Ananta [21]. In contrast to these previous designs, RUBIK substantially reduces the DC network bandwidth usage due to traffic indirection while providing low cost, high performance benefits.

OpenFlow based LB: Several recent proposals focus on using OpenFlow switches for load balancing. In [24], the authors present a preliminary LB design using OpenFlow switches. They focus on minimizing the number of wildcard rules. In [18], the authors propose a hybrid hardware-software design and propose algorithms to calculate the weights for splitting the VIP traffic. Plug-n-Serve [16] is another preliminary design that uses OpenFlow switches to load balance web servers deployed in unstructured, enterprise networks. In contrast, RUBIK is designed for DC networks and efficiently load balances the traffic by exploiting locality and end-point flexibility.

SDN architecture and middleboxes: Researchers have leveraged the SDN designs in the context of middleboxes for policy enforcement and verification [22, 13], which is a different goal from RUBIK. Researchers have also proposed using OpenFlow switches for a variety of other purposes. *e.g.,* DIFANE [25] and vCRIB [19] use switches to cache rules and act as authoritative switches. Again their main focus is quite different from RUBIK.

12 Conclusion

RUBIK is a new load balancer design that drastically reduces the bandwidth usage while providing low cost, high performance and reliability benefits. RUBIK achieves this by exploiting two design principles: (1) locality: it load balances traffic generated in individual ToRs across DIPs present in the same ToRs, (2) end-point flexibility: it places the DIPs closer to the traffic sources. We evaluate RUBIK using a prototype implementation and extensive simulations using traces from our production DC. Our evaluation shows together these two principles reduce the bandwidth usage by the load balanced traffic by over 3x compared to prior art Duet.

References

[1] A10 networks ax series. http://www.a10networks.com.

[2] Broadcom smart hashing. http://http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf.

[3] Embrane. http://www.embrane.com.

[4] F5 load balancer. http://www.f5.com.

[5] Fattree routing using openflow. https://github.com/brandonheller/ripl.

[6] Ha proxy load balancer. http://haproxy.1wt.eu.

[7] Ibm cplex lp solver. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[8] Loadbalancer.org virtual appliance. http://www.load-balancer.org.

[9] Netscalar vpx virtual appliance. http://www.citrix.com.

[10] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *EuroSys 2011*.

[11] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *SODA, 1999*.

[12] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM 2013*.

[13] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. *Proc. HotSDN*, 2013.

[14] R. Gandhi, H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM 2014*.

[15] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM 2009*.

[16] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-serve: Load-balancing web traffic using openflow. *ACM SIGCOMM Demo*, 2009.

[17] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. *SIGCOMM*, 2013.

[18] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Niagara: Scalable load balancing on commodity switches. In *Technical Report (TR-973-14), Princeton, 2014*.

[19] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *NSDI*, 2013.

[20] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al. The case for ramcloud. *Communications of the ACM, 2011*.

[21] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[22] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *SIGCOMM, 2013*.

[23] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI 2012*.

[24] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Usenix HotICE*, 2011.

[25] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.

Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters

Konstantinos Karanasos Sriram Rao Carlo Curino Chris Douglas
Kishore Chaliparambil Giovanni Matteo Fumarola Solom Heddaya
Raghu Ramakrishnan Sarvesh Sakalanaga

Microsoft Corporation

Abstract

Datacenter-scale computing for analytics workloads is increasingly common. High operational costs force heterogeneous applications to share cluster resources for achieving economy of scale. Scheduling such large and diverse workloads is inherently hard, and existing approaches tackle this in two alternative ways: 1) *centralized* solutions offer strict, secure enforcement of scheduling invariants (e.g., fairness, capacity) for heterogeneous applications, 2) *distributed* solutions offer scalable, efficient scheduling for homogeneous applications.

We argue that these solutions are complementary, and advocate a blended approach. Concretely, we propose *Mercury*, a *hybrid resource management framework* that supports the full spectrum of scheduling, from centralized to distributed. Mercury exposes a programmatic interface that allows applications to trade-off between scheduling overhead and execution guarantees. Our framework harnesses this flexibility by opportunistically utilizing resources to improve task throughput. Experimental results on production-derived workloads show gains of over 35% in task throughput. These benefits can be translated by appropriate application and framework *policies* into job throughput or job latency improvements. We have implemented and contributed Mercury as an extension of Apache Hadoop / YARN.[1]

1 Introduction

Over the past decade, applications such as web search led to the development of datacenter-scale computing, on clusters with thousands of machines. A broad class of data analytics is now routinely carried out on such large clusters over large heterogeneous datasets. This is often referred to as "Big Data" computing, and the diversity of applications sharing a single cluster is growing dramatically for various reasons: the consolidation of clusters to increase efficiency, the diversity of data (ranging from relations to documents, graphs and logs) and the corresponding diversity of processing required, the range of techniques (from query processing to machine learning) being increasingly used to understand data, the ease of use of cloud-based services, and the growing adoption of Big Data technologies among traditional organizations.

This diversity is addressed by modern frameworks such as YARN [27], Mesos [16], Omega [24] and Borg [28], by exposing cluster resources via a well-defined set of APIs. This facilitates concurrent sharing between applications with vastly differing characteristics, ranging from batch jobs to long running services. These frameworks, while differing on the exact solution (monolithic, two-level or shared-state) are built around the notion of centralized coordination to schedule cluster resources. For ease of exposition, we will loosely refer to all such approaches as *centralized scheduler* solutions. In this setting, individual per-job (or per-application framework) managers petition the centralized scheduler for resources via the resource management APIs, and then coordinate application execution by launching tasks within such resources.

Ostensibly, these centralized designs simplify cluster management in that there is a single place where scheduling invariants (e.g., fairness, capacity) are specified and enforced. Furthermore, the central scheduler has cluster-wide visibility and can optimize task placement along multiple dimensions (locality [31], packing [15], etc.).

However, the centralized scheduler is, by design, in the critical path of *all* allocation decisions. This poses scalability and latency concerns. Centralized designs rely on heartbeats which are used for both liveness and for triggering allocation decisions. As the cluster size scales, to minimize heartbeat processing overheads, operators are forced to lower the heartbeat rate (i.e., less frequent heartbeats). In turn, this increases the scheduler's

[1] The open-sourcing effort is ongoing at the moment of writing the paper. Progress can be tracked in Apache JIRA [6].

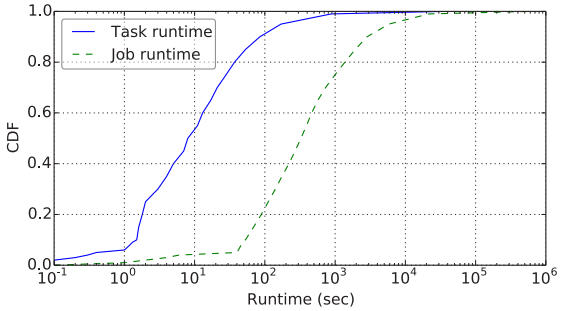


Figure 1: Task and job runtime distribution.

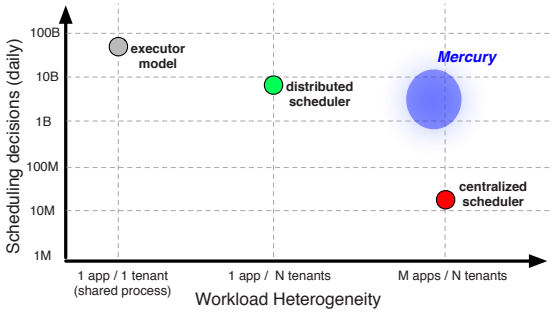


Figure 2: Ideal operational point of alternative scheduling approaches.

allocation latency. This compromise becomes problematic if typical tasks are short [22]. A workload analysis from one of the production clusters at Microsoft also suggests that shorter tasks are dominant. This is shown as a CDF of task duration in Figure 1. Note that almost 60% of the tasks complete execution under 10 seconds. Therefore, the negative effects of centralized heartbeat-based solutions range from poor latency for interactive workloads to utilization issues (slow allocation decisions means resources are fallow for longer periods of time).

To amortize the high scheduling cost of centralized approaches, the "executor" model has been proposed [29, 19, 21, 22]. This hierarchical approach consists in reusing containers assigned by the central scheduler to an application framework that multiplexes them across tasks/queries.[2] Reusing containers assumes that submitted tasks have similar characteristics (to fit in existing containers). Moreover, since the same system-level process is shared across tasks, the executor model has limited applicability to *within* a single application type. It is, thus, orthogonal to our work.

Fully distributed scheduling is the leading alternative to obtain high scheduling throughput. A practical system leveraging this design is Apollo [9]. Apollo allows each running job to perform independent scheduling choices and to queue its tasks directly at worker nodes. Unfortunately, this approach relies on a uniform workload (in terms of application type), as all job managers need to run the same scheduling algorithm. In this context, allowing arbitrary applications, while preventing abuses and strictly enforcing capacity/fairness guarantees, is non-trivial. Furthermore, due to lack of global view of the cluster, distributed schedulers make local scheduling decisions that are often not globally optimal.

In Figure 2, we pictorially depict the ideal operational point of these three approaches: centralized [16, 27], distributed [9], and executor-model [29, 22], as well as the target operational point for our design. A detailed discussion of related work is deferred to § 8.

[2]By containers we refer to the allocation units that may comprise multiple resources, such as memory and CPU.

The key technical challenge we explore in this paper is the design of a resource management infrastructure that allows us to simultaneously: (1) support diverse (possibly untrusted) application frameworks, (2) provide high cluster throughput with low-latency allocation decisions, and (3) enforce strict scheduling invariants (§ 2).

Below we present the main contributions of this paper.

First: we propose a novel *hybrid* resource management architecture. Our key insight is to offload work from the centralized scheduler by *augmenting* the resource management framework to include an auxiliary set of schedulers that make fast/distributed decisions (see Fig. 3). The resource management framework comprising these schedulers is now collectively responsible for all scheduling decisions (§ 3).

Second: we expose this flexibility to the applications by associating semantics with the type of requested containers (§ 3.2). Applications may now choose to accept high scheduling costs to obtain strong execution guarantees from the centralized scheduler, or *trade strict guarantees for sub-second distributed allocations*. Intuitively, opportunistic jobs or applications with short tasks can benefit from fast allocations the most.

Third: we leverage the newly found scheduling flexibility to explore the associated policy space. Careful policy selection allows us to translate the faster scheduling decisions into job throughput or latency gains (§ 4 and § 5).

Fourth: we implement, validate and open-source this overall design in a YARN-based system called *Mercury* (§ 6). We compare Mercury with stock YARN by running synthetic and production-derived workloads on a 256-machine cluster. We show 15 to 45% task throughput improvement, while maintaining strong invariants for the applications that need them. We also show that by tuning our policies we can translate these task throughput gains to improvements of either job latency or throughput (§ 7).

The open-source nature [6] and architectural generality of our effort makes Mercury an ideal substrate for other researchers to explore centralized, distributed and hybrid scheduling solutions, along with a rich policy space. We describe ongoing work in § 9.

2 Requirements

Given a careful analysis of production workloads at Microsoft, and conversations with cluster operators and users, we derive the following set of requirements we set out to address with Mercury:

R1 **Diverse application frameworks**: Allow arbitrary user code (as opposed to a homogeneous, single-app workload).

R2 **Strict enforcement of scheduling invariants**: Example invariants include fairness and capacity; this includes policing/security to prevent abuses.

R3 **Maximize cluster utilization and throughput:** Higher cluster utilization and throughput lead to higher return on investment (ROI).

R4 **Fine-grained resource sharing**: Tasks from different jobs can concurrently share a single node.

R5 **Efficiency and scalability of scheduling**: Support high rate of scheduling decisions.

Note that classical centralized approaches target R1-R4, while distributed approaches focus on R3-R5. We acknowledge the tension between conflicting requirements (R2 and R5), each emerging from a subset of the applications we aim to support. In Mercury, we balance this tension by blending centralized and distributed decision-making in a request-specific manner.

Non-goals *Low latency for sub-second interactive queries is outside the scope of our investigation*. This is the target of executor-model approaches [29, 19, 21, 22], which achieve millisecond start times by sharing processes. This is at odds with requirements R1-R2.

3 Mercury Design

We first provide an overview of the Mercury architecture (§ 3.1). Next, we describe the programming interface that Job Managers use for requesting resources (§ 3.2), and how the framework allocates them (§ 3.3). Then we provide details about task execution (§ 3.4).

3.1 Overview

Mercury comprises two subsystems, as shown in Fig. 3:

Mercury Runtime This is a daemon running on every worker node in the cluster. It is responsible for all interactions with applications, and for the enforcement of execution policies on each node.

Mercury Resource Management Framework This is a subsystem that includes a *central scheduler* running on a dedicated node, and a set of *distributed schedulers* running on (possibly a subset of) the worker nodes, which loosely coordinate through a *Mercury Coordinator*. This combination of schedulers performs cluster-wide resource allocation to applications for the same pool of resources. The allocation unit, referred to as a *container*, consists of a combination of CPU and RAM resources on an individual machine.

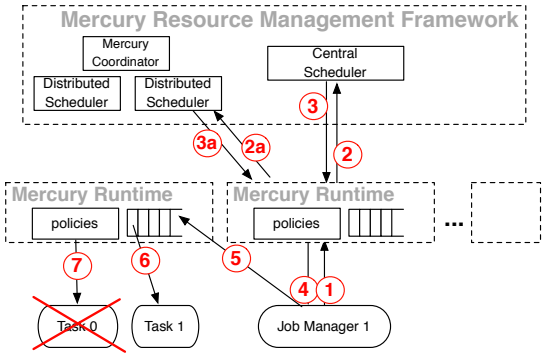


Figure 3: Mercury resource management lifecycle.

Note that we do not dedicate specific part of the cluster resources to each scheduler. This is done dynamically, based on the resource requests from the running applications and the condition of the cluster, as described in § 4 and § 5. Conflicts emerging from schedulers assigning resources over the same pool of machines are resolved optimistically by the Mercury Runtime.

Next we present the resource management lifecycle, following the steps of Fig. 3.

Resource request Consider an application running in the cluster (Job Manager 1) that wants to obtain resources. To this end, it petitions the local Mercury Runtime through an API that abstracts the complex scheduling infrastructure (step 1). The API allows applications to specify whether they need containers with strict execution guarantees or not (§ 3.2). Based on this information and on framework policies (§ 4), the runtime delegates the handling of a request to the central scheduler (step 2) or to one of the distributed schedulers (step 2a).

Container allocation The schedulers assign resources to the application according to their scheduling invariants, and signal this by returning to the Mercury Runtime *containers* that grant access to such resources (steps 3 and 3a). The Mercury Runtime forwards back to the Job Manager all the granted containers (step 4).

Task execution The application submits each allocated container for execution to the Mercury Runtime on the associated node (step 5).[3] Depending on scheduling priorities among containers and the resource utilization on the node, the runtime decides whether the container should be executed immediately or get enqueued for later execution (more details in § 3.4). To execute a container, the remote Mercury Runtime spawns a process on that node and runs the application task (step 6). To ensure

[3] Containers are bound to a single machine to prevent abuses [27].

that priorities are enforced, the runtime can also decide to kill or preempt running tasks (step 7), to allow immediate execution of higher priority tasks.

3.2 Resource Request

When requesting containers, a Job Manager uses Mercury's programming interface to specify the type of containers it needs. This specification is based on the container's allocation/execution semantics. Our design defines the following two container classes:

`GUARANTEED` containers incur no queuing delay, i.e., they are spawned by the Mercury Runtime as soon as they arrive to a worker node. Moreover, these containers run to completion bar failures, i.e., they are never preempted or killed by the infrastructure.

`QUEUEABLE` containers enable the Job Manager to "queue" a task for execution on a specific node. No guarantees are provided on the queuing delay, or on whether the container will run to completion or be preempted.

3.3 Container Allocation

In our design (see Figure 3), `GUARANTEED` containers are allocated by the *central scheduler* and `QUEUEABLE` containers are allocated by one of the *distributed schedulers*. Requests for either containers are routed appropriately by the Mercury Runtime. Furthermore, both schedulers are free to allocate containers on *any* node in the cluster. In what follows, we describe the design rationale.

The *central scheduler* has knowledge about container execution as well as resource availability on individual machines. This information is part of the periodic heartbeat messages that are exchanged between the framework components. Consequently, the *central scheduler* can perform careful placement of `GUARANTEED` containers without causing resource contention.

To support fast container allocation, a *distributed scheduler* restricts itself to allocating `QUEUEABLE` containers, which can be placed on *any* machine in the cluster. The *distributed scheduler* uses lightweight cluster load information, provided by the Mercury Coordinator, for making placement decisions.

The path not taken: We considered and discarded two alternative designs. First the central scheduler could make all scheduling decisions, including `QUEUEABLE`. Such design would overload the central scheduler. This would be coped with by limiting the rate at which Job Managers can petition the framework for resources (e.g., every few seconds instead of in the millisecond range as we enable with Mercury). This is akin to forfeiting R5. The second alternative sees the framework-level distributed scheduler making all decisions, including `GUARANTEED`. This would require costly consensus building among schedulers to enforce strict invariants, or relax our guarantees, thus forfeiting R2.

The *hybrid* approach of Mercury allows us to meet requirements R1- R5 of § 2, as we validate experimentally.

3.4 Task Execution

As described above, Mercury's centralized and distributed schedulers independently allocate containers on a single shared pool of machines. This in turn means that conflicting allocations can be made by the schedulers, potentially causing resource contention. Mercury Runtime resolves such conflicts as follows:

`GUARANTEED` - `GUARANTEED` By design the central scheduler prevents this type of conflicts by linearizing allocations. This is done by allocating a `GUARANTEED` container only when it is certain that the target node has sufficient resources.

`GUARANTEED` - `QUEUEABLE` This occurs when a central scheduler and the distributed scheduler(s) allocate containers on the same node, causing the node's capacity to be exceeded. Following the semantics of § 3.2, any cross-type conflict is resolved in favor of `GUARANTEED` containers. In the presence of contention, (potentially all) running `QUEUEABLE` containers are terminated to make room for any newly arrived `GUARANTEED`. If `GUARANTEED` containers are consuming all the node resources, the start of `QUEUEABLE` ones is delayed until resources become available.

`QUEUEABLE` - `QUEUEABLE` This occurs when multiple distributed schedulers allocate containers on the same target node in excess of available resources. Mercury Runtime on the node enqueues the requests (see Figure 3) and thereby prevents conflicts. To improve job-level latency, we explore a notion of priority among `QUEUEABLE` containers in § 4.

When a `QUEUEABLE` container is killed there is potentially wasted computation. To avoid this, Mercury supports promoting a running `QUEUEABLE` container to a `GUARANTEED` one. A Job Manager can submit a promotion request to the Mercury Runtime, which forwards it to the *central scheduler* for validation. The promotion request will succeed only if the *central scheduler* determines that the scheduling invariants would not be violated.

4 Framework Policies

In the previous section we presented our architecture and the lifecycle of a resource request. We now turn to the policies that govern all scheduling decisions in our system. For ease of exposition we group the policies in three groups: *Invariants enforcement*, *Placement*, and *Load shaping*, as described in the following subsections.

4.1 Invariants Enforcement Policies

These policies describe how scheduling invariants are enforced throughout the system.

Invariants for `GUARANTEED` containers Supporting scheduling invariants for centralized scheduler designs is well studied [1, 2, 14]. Furthermore, widely deployed Hadoop/YARN frameworks contain robust implementations of cluster sharing policies based on capacity [1] and fairness [2]. Hence, Mercury's *central scheduler* leverages this work, and can enforce any of these policies when allocating `GUARANTEED` containers.

Enforcing quotas for `QUEUEABLE` containers The enforcement of invariants for distributed schedulers is inherently more complex. Recall that applications have very limited expectations when it comes to `QUEUEABLE` containers. However, cluster operators need to enforce invariants nonetheless to prevent abuses. We focus on one important class of invariants: *application-level quotas*. Our Mercury Runtime currently provides operators with two options: (1) an absolute limit on the number of concurrently running `QUEUEABLE` containers for each application (e.g., a job can have at most 100 outstanding `QUEUEABLE` containers), and (2) a limit relative to the number of `GUARANTEED` containers provided by the central scheduler (e.g., a job can have `QUEUEABLE` containers up to $2\times$ the number of `GUARANTEED` containers).

4.2 Placement Policies

These policies determine how requests are mapped to available resources by our scheduling framework.

Placement of `GUARANTEED` containers Again, for central scheduling we leverage existing solutions [1, 2]. The central scheduler allocates a `GUARANTEED` container on a node, if and only if that node has sufficient resources to meet the container's demands. By tracking when `GUARANTEED` containers are allocated/released on a per-node basis, the scheduler can accurately determine cluster-wide resource availability. This allows the central scheduler to suitably delay allocations until resources become available. Furthermore, the scheduler may also delay allocations to enforce capacity/fairness invariants.

Distributed placement of `QUEUEABLE` containers Our objective when initially placing `QUEUEABLE` containers is to minimize their *queuing* delay. This is dependent on two factors. First, the head-of-line blocking at a node is estimated based on: (1) the cumulative execution times for `QUEUEABLE` containers that are currently enqueued (denoted T_q), (2) the remaining estimated execution time for running containers (denoted T_r). To enable this estimation, individual Job Managers provide task run-time estimates when submitting containers for execution.[4] Second, we use the elapsed time since a `QUEUEABLE` container was last executed successfully on a node, denoted T_l, as a broad indicator of resource availability for `QUEUEABLE` containers on that node. The Mercury Runtime determines at regular intervals the ranking order R of a node as follows:

$$R = T_q + T_r + T_l$$

Then it pushes this information to the Mercury Coordinator that disseminates it to the whole cluster through the heartbeat mechanism. Subsequently, each distributed scheduler uses this information for load balancing purposes during container placement. We build around a pseudo-random approach in which a distributed scheduler allocates containers by arbitrarily choosing amongst the "top-k" nodes that have minimal queuing delays, while respecting locality constraints.

4.3 Load Shaping Policies

Finally, we discuss key policies related to maximizing cluster efficiency. We proceed from dynamically (re)-balancing load across nodes, to imposing an execution order to `QUEUEABLE` containers, to node resource policing.

Dynamically (re)-balancing load across nodes To account for occasionally poor placement choices for `QUEUEABLE` containers, we perform *load shedding*.[5] This has the effect of dynamically re-balancing the queues across machines. We do so in a lightweight manner using the Mercury Coordinator. In particular, while aggregating the queuing time estimates published by the per-node Mercury Runtime, the Coordinator constructs a distribution to find a targeted maximal value. It then disseminates this value to the Mercury Runtime running on individual machines. Subsequently, using this information, the Mercury Runtime on a node whose queuing time estimate is above the threshold, selectively discards `QUEUEABLE` containers to meet this maximal value. This forces the associated individual Job Managers to requeue those containers elsewhere.

Observe that these policies rely on the task execution estimates provided by the users. Interestingly, even in case of inaccurate estimates, re-balancing policies will restore the load balance in the system. Malicious users that purposely and systematically provide wrong estimates are out of the scope of this paper, although our system design allows us to detect such users.

Queue reordering Reordering policies are responsible for imposing an execution order to the queued tasks. Various such policies can be conceived. In Mercury, we are

[4] Such estimates are currently provided by the users, but can also be derived from previous job executions, and/or be dynamically adjusted as parts of a job get executed.

[5] Other methods, such as work stealing, can also be applied. We use load shedding as it naturally fits into a YARN-based implementation.

currently ordering tasks based on the submission time of the job they belong to. Thus, tasks belonging to jobs submitted earlier in the system will be executed first. This policy improves job tail latency, allowing jobs to finish faster. This in turn allows more jobs to be admitted in the system, leading to higher task throughput, as we also show experimentally in § 7.2.3.

Resource policing: minimizing killing To minimize preemption/killing of running QUEUEABLE containers, the Mercury Runtime has to determine *when* resources can be used for opportunistic execution. In doing so, it maximizes the chances of a QUEUEABLE container actually running to completion. We develop a simple policy that leverages historical information about aggregate cluster utilization to identify such opportunities. Based on current and expected future workload, the Mercury Coordinator notifies the per-node Mercury Runtimes regarding the amount of local resources that will be required for running GUARANTEED containers over a given time window. Subsequently, the Mercury Runtime can opportunistically use the remaining resources in that period for QUEUEABLE containers and thereby minimize preemption.

5 Application-level Policies

As explained in § 3.1, Mercury exposes the API for applications to request both GUARANTEED and QUEUEABLE containers. To take advantage of this flexibility, each Job Manager should implement an application policy that determines the desired type of container for each task. These policies allow users to tune their scheduling needs, going all the way from fully centralized scheduling to fully distributed (and any combination in between).

In this paper, we introduce the following flexible policy, while we discuss more sophisticated options in our technical report [18].

hybrid-GQ is a policy that takes two parameters: a task duration threshold t_d, and a percentage of QUEUEABLE containers p_q. QUEUEABLE containers are requested for tasks with expected duration smaller than t_d, in p_q percent of the cases. All remaining tasks use GUARANTEED containers. In busy clusters, jobs' resource starvation is avoided by setting p_q to values below 100%. Note that fully centralized scheduling corresponds to setting $t_d = 0$, and fully distributed scheduling corresponds to setting $t_d = \infty$ and $p_q = 100\%$. We refer to these policies as only-G and only-Q, respectively.

6 Mercury Implementation

We implemented Mercury by extending Apache Hadoop YARN [3]. We provide a brief overview of YARN before detailing the modifications that support our model.

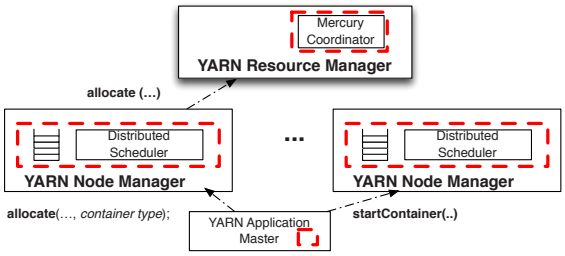


Figure 4: *Mercury implementation*: dashed boxes show Mercury modules and APIs as YARN extensions.

6.1 YARN Overview

Hadoop YARN [27] is a cluster resource management framework that presents a generalized job scheduling interface for running applications on a shared cluster. It is based on a centralized scheduling architecture, consisting of the following three key components.

ResourceManager (RM): This is a central component that handles arbitration of cluster resources amongst jobs. The RM contains a pluggable scheduler module with a few implementations [1, 2]. Based on the sharing policies, the RM allocates containers to jobs. Each allocation includes a token that certifies its authenticity.

NodeManager (NM): This is a per-node daemon that spawns processes locally for executing containers and periodically heartbeats the RM for liveness and for notifying it of container completions. The NM validates the token offered with the container.

ApplicationMaster (AM): This is a per-job component that orchestrates the application workflow. It corresponds to the *Job Manager* we use throughout the paper.

6.2 Mercury Extensions to YARN

We now turn to the implementation of Mercury in YARN. Further details can be found in JIRA (the Apache Hadoop feature and bug tracking system) [6].

Adding container types We introduce our notion of container *type* as a backward-compatible change to the allocation protocol. The semantics of the containers allocated by the YARN RM match GUARANTEED containers. Hence, as shown in Fig. 4, the YARN RM corresponds to the *central* scheduler of our Mercury design. QUEUEABLE containers are allocated by an *ex novo* distributed scheduler component, which we added to the NM.

Interposing Mercury Runtime We have implemented Mercury Runtime as a module inside the YARN NM (see Fig. 4) and thereby simplified its deployment. As part of our implementation, a key architectural change we made to YARN is that the Mercury Runtime is introduced as a

layer of indirection with two objectives. First, the Mercury Runtime proxies container allocation requests between an AM and Mercury's schedulers, thereby controlling how requests are satisfied. This proxying is effected by rewriting configuration variables and does not require modifications to AM. Second, for enforcing execution semantics, the Mercury Runtime intercepts an AM submitted container request to the NM and handles them appropriately. We elaborate on these next.

The AM annotates each request with the weakest guarantee it will accept, then forwards the request using the `allocate()` call in Fig. 4. Mercury directs requests for `GUARANTEED` resources to the central RM, but it may service `QUEUEABLE` requests using the instance of Mercury's distributed scheduler running in the NM. When this happens, since it is essentially a process context switch, the `QUEUEABLE` containers (and tokens) for any node in the cluster are issued with millisecond latency. The authenticity of the container allocations made by a distributed scheduler is validated at the target NM using the same token checking algorithm that YARN uses for verifying `GUARANTEED` containers.

To enforce the guarantees provided by the respective container types, Mercury intercepts container creation commands at the NM. As illustrated in Fig. 4, a `startContainer()` call will be directed to the Mercury Runtime module running in the NM. This module implements the policies described in § 4; based on the container type, the Mercury Runtime will enqueue, kill and create containers.

6.3 Distributed Scheduler(s)

The distributed scheduler is implemented as a module running in each NM. We discuss the changes necessary for enforcing the framework policies described in § 4.

Placement To direct `QUEUEABLE` containers to fallow nodes, Mercury uses estimates of queuing delay as described in § 4.2. For computing this delay, the Mercury Runtime requires computational time estimates for each enqueued container. We modified the Hadoop MapReduce [3] and Tez [4] AMs to provide estimates based on static job information. Furthermore, in our implementation, the AMs continuously refine estimates at runtime based on completed container durations. The Mercury Coordinator is implemented as a module inside the YARN RM (Fig. 4). It collects and propagates queuing delays as well as the "top-*k*" information by suitably piggybacking on the RM/NM heartbeats.

Dynamic load balancing Our implementation leverages the Mercury Coordinator for dynamic load balancing. We modified the YARN RM to aggregate information about the estimated queuing delays, compute outliers (i.e., nodes whose queuing delays are significantly higher than average), and disseminate cluster-wide the targeted queuing delay that individual nodes should converge to. We added this information to YARN protocols and exchange it as part of the RM/NM heartbeats. Upon receiving this information, the Mercury Runtime on an outlier node discards an appropriate number of queued containers so as to fit the target. Containers dropped by a Mercury Runtime instance are marked as `KILLED` by the framework. The signal propagates as a YARN event to the Mercury Runtime, which proxies it to the AM. The AM will forge a new request, which will be requeued at a less-loaded node.

Quotas To prevent `QUEUEABLE` traffic from overwhelming the cluster, Mercury imposes operator-configured quotas on a per-AM basis. A distributed scheduler maintains an accurate count by observing allocations and container start/stop/kill events.

7 Experimental Evaluation

We deployed our YARN-based Mercury implementation on a 256-node cluster and used it to drive our experimental evaluation. § 7.1 provides the details of our setup. In § 7.2, we present results from a set of micro-experiments using short tasks. Then in § 7.3, we describe results for a synthetic workload involving tasks with a range of execution times. Finally, in § 7.4, we give results from workloads based on Microsoft's production clusters.

Our key results are:

1. Our policies can translate task throughput gains into improved job latency for 80% of jobs, and 36.3% higher job throughput (§ 7.2.1).
2. Careful resource policing reduces the preemption of `QUEUEABLE` containers by up to 63% (§ 7.2.3).
3. On production-derived workloads, Mercury achieves 35% task throughput gain over Stock YARN (§ 7.4).

7.1 Experimental Setup

We use a cluster of approximately 256 machines, grouped in racks of at most 40 machines. Each machine has two 8-core Intel Xeon E5-2660 processors with hyper-threading enabled (32 virtual cores), 128 GB of RAM, and 10 x 3-TB data drives configured as a JBOD. The connectivity between any two machines within a rack is 10 Gbps while across racks is 6 Gbps.

We deploy Hadoop/YARN 2.4.1 with our Mercury extensions for managing the cluster's computing resources amongst jobs. We set the heartbeat frequency to 3 sec, which is also the value used in production clusters at Yahoo!, as reported in [27]. For storing job input/output we use HDFS [7] with 3x data replication. We use Gridmix [8], an open-source benchmark that uses workload

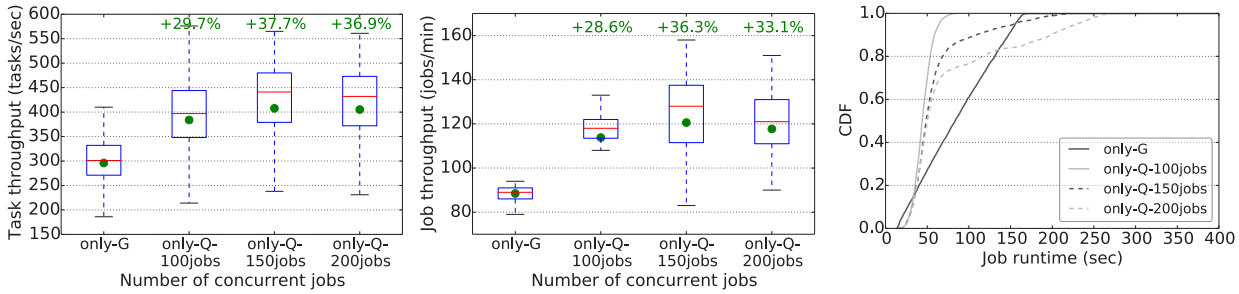


Figure 5: Task throughput, job throughput and job latency for varying number of concurrent jobs.

traces for generating synthetic jobs for Hadoop clusters. We use Tez 0.4.1 [4] as the execution framework for running these jobs.

Metrics reported In all experiments we measure task throughput, job throughput, and job latency for runs of 30 mins. Due to space limitations, we report only task throughput in some cases, however, the full set of results can be found in [18]. Note that for the task and job throughput we are using *box plots* (e.g., see Fig. 5), in which the lower part of the main box represents the 25-percentile, the upper part the 75-percentile, and the red line the median. Moreover, the lower whisker is the 5-percentile, the upper the 95-percentile, and the green bullet the mean.

7.2 Microbenchmarks

In this section we perform a set of micro-experiments that show how Mercury can translate task throughput gains into job throughput/latency gains. For a given workload, we first study how the maximum number of jobs allowed to run concurrently in the cluster affects performance (§ 7.2.1). Then, we experimentally assess various framework policies (as discussed in § 4), including placement (§ 7.2.2) and load shaping policies (§ 7.2.3).

For all experiments of this section we use Gridmix to generate jobs with 200 tasks/job, in which each task executes, on average, for a 1.2 sec duration. We use the `only-G` and `only-Q` policies (§ 5).

7.2.1 Varying Number of Concurrent Jobs

In this experiment, we investigate the performance of the system by altering the number of jobs that the scheduling framework allows to run concurrently. For distributed scheduling (`only-Q`), we set this limit to 100, 150 and 200 jobs. This is compared with the central scheduler (`only-G`) that implements its own admission control [27], dynamically adjusting the number of running jobs based on the cluster load. Fig. 5 shows that `only-Q` dominates across the board, and that, given our cluster configuration, 150 concurrent jobs yield the maximum increase of task throughput, i.e., 38% over `only-G`. This task throughput improvement translates to improvement in both job throughput and latency (higher by 36% and 30%, respectively, when compared to `only-G`). Low job limits (100 jobs) fail to fully utilize cluster resources, while high limits (200 jobs) impact latency negatively.

In the following experiments, we use the 150-job limit, as this gives the best compromise between job throughput and latency, and explore other parameters. At each experiment we adjust the job submission rate, so as to have sufficient jobs at each moment to reach the job limit.

7.2.2 Placement Policies (Varying Top-k)

As discussed in § 4.2, whenever a distributed scheduler needs to place a task on a node, it picks among the k nodes with the smallest estimated queuing delay. Here we experiment with different values for k. Our results are shown in Fig. 6. The biggest gains are achieved for k=50, with 44.5% higher task throughput compared to `only-G`. Lower values (k=20) leave nodes under-utilized, while higher values (k=100) place tasks to already highly-loaded nodes. In both cases, higher load imbalance is created, leading to lower task throughput. Therefore, in the remainder of the experiments we use k=50.

7.2.3 Load Shaping Policies

In this section we study the load shaping policies that were presented in § 4.3.

Balancing node load and queue reordering We experiment with different ways of rebalancing node queues. We synthetically cause imbalance by introducing few straggler nodes that underestimate queuing delay. Our results are given in Fig. 7. Among the presented policies, (1) `only-Q` is a basic approach with no rebalancing; (2) `only-Q/avg+`σ triggers rebalancing actions for any node with a queuing delay which is over mean plus one standard deviation (σ);

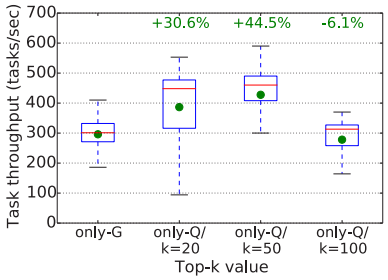


Figure 6: Task throughput for varying top-*k*.

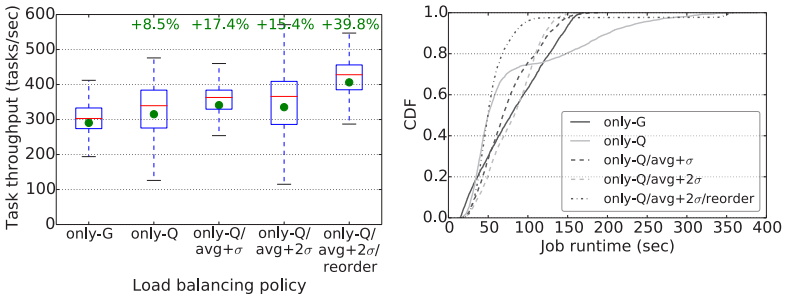


Figure 7: Task throughput and job latency for various load balancing policies.

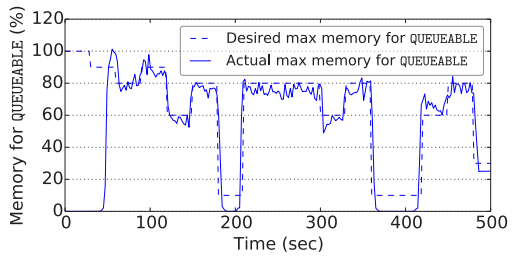


Figure 8: Desired and actual maximum percentage of memory given to QUEUEABLE containers at each node.

Memory limit per node for QUEUEABLE containers	20%	30%	40%	100%
Mean containers killed / node	287	428	536	780
Mean slot utilization for QUEUEABLE	11.4%	16.7%	20.9%	28.4%

Table 1: Effectiveness of maximum memory limit for QUEUEABLE containers.

(3) only-Q/avg+2σ is as above with 2 standard deviations; (4) only-Q/avg+2σ/reorder is as above with reordering of containers in the queue (favoring jobs submitted earlier). Imbalances limit the task throughput gains of only-Q to 8.5% over our baseline only-G. Subsequent refinements improve the resulting gains by up to 39.8%. Note that reordering reduces average job latency: as jobs exit the system, new jobs start, and by imposing fresh demand for resources drive utilization higher. We measured frequency of task dequeuing to be at an acceptable 14% of all tasks.

Resource policing: minimizing container killing To show how resource policing (discussed in § 4.3) can be used to minimize container killing, we create a Gridmix workload that generates a stable load of 70% using GUARANTEED containers, that is, 30% of the slots can be used at each moment for QUEUEABLE containers. At the same time, we are submitting QUEUEABLE containers and observe the average number of such containers killed (due to the GUARANTEED ones). We set the allowed memory limit for QUEUEABLE containers to 20, 30, 40, and 100% (the latter corresponds to no limit). Our results are shown in Table 1. We also report the average utilization due to QUEUEABLE containers. Our implementation is able to opportunistically use resources leading to utilization gains. However, given a steady compute demand, aggressively utilizing those resources without knowing future demand does cause an increase in task kills.

To address this issue we develop a novel policy using historical cluster utilization data to determine the compute demand for current and future workload due to GUARANTEED containers. Any remaining resources can be used for executing QUEUEABLE containers. We input this information to the Mercury Coordinator, which periodically propagates it to the Mercury Runtime on individual machines. This allows the Mercury Runtime on each node to determine how many of the unallocated resources can be used opportunistically. Fig. 8 shows the actual (dashed line) and the observed (solid line) resources used at a node for executing QUEUEABLE containers. The two lines track closely, demonstrating that our implementation adapts to changing cluster conditions. This also shows that there is no need for strict partitioning of resources.

7.3 Impact of Task Duration

We now explore the impact of task duration, by using task run-times of 2, 10, and 20 sec. We compare only-G and only-Q, parameterized at best, given our § 7.2 experiments. Our results are shown in Fig. 9. As expected, the longer the task duration, the smaller the benefit from using distributed scheduling. In particular, when compared to the centralized scheduling, we get approx. 40% gains both in task and job throughput for jobs with 2 sec tasks. This gain drops to about 14% for jobs with 20 sec tasks. Likewise, average job latency for distributed scheduling is comparable with centralized for 2 sec tasks, but is 60% worse for 20 sec tasks.

Note that for short tasks, to fully utilize the cluster resources, more jobs are admitted in the cluster. For dis-

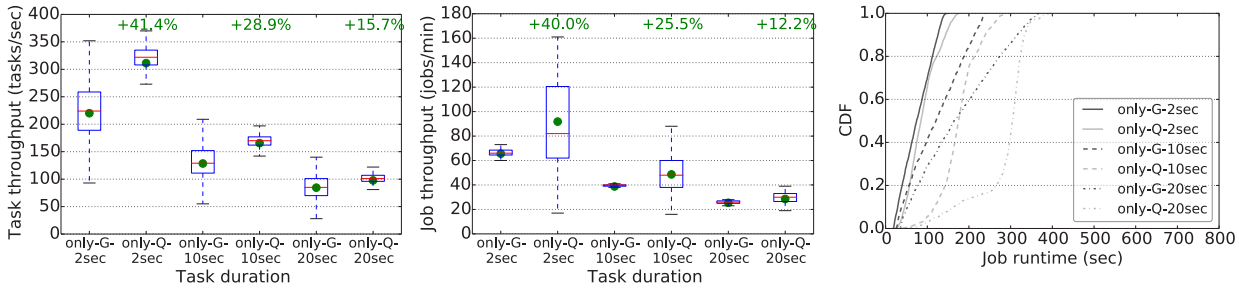


Figure 9: Task throughput, job throughput and job latency for jobs with increasing task duration.

tributed scheduling, this leads to queuing at the same time tasks belonging to a bigger number of jobs, which increases the variance of job duration and thus of job throughput. We are investigating more involved queue reordering techniques to further mitigate this issue.

7.4 Microsoft-based Hybrid Workload

Finally we assess our system against a complex scenario. We create a Gridmix workload that follows the task duration characteristics observed in Microsoft production clusters, as shown in Fig. 1.

We explore several configurations for our `hybrid-GQ` policy (§ 5). Besides `only-G` and `only-Q`, we have:

`50%-Q:` all tasks have a 50% chance of being `QUEUEABLE` ($t_D = \infty$, $p_q = 50\%$);

`<5sec-Q:` all tasks shorter than 5 seconds are `QUEUEABLE` ($t_D = 5sec$, $p_q = 100\%$);

`<10sec-70%-Q:` 70% of tasks shorter than 10 seconds are `QUEUEABLE` ($t_D = 10sec$, $p_q = 70\%$);

In Fig. 10 we report on the task throughput, as well as the job latency for jobs with various task durations from this workload. In this mixed scenario, using `only-Q` gives almost no improvement in task throughput and also leads to worse job latency for jobs of all durations. `50%-Q` gives the best task throughput, but that does not get translated to clear wins in the latency of jobs with short tasks (e.g., jobs with 3 and 11 sec tasks), especially for the higher percentiles, due to the unpredictability of `QUEUEABLE` containers. On the other hand, handing `QUEUEABLE` containers to the short tasks gives a significant improvement in task throughput (`<10sec-70%-Q` achieves a 26% gain compared to `only-G`), and has a performance comparable to the centralized scheduler for the short tasks. What is more, for the longer tasks there is significant job latency improvement. For instance, `<10sec-70%-Q` reduces mean latency by 66% (82%) when compared to `only-G` (`only-Q`) for 11 sec tasks. The intuition behind these gains is that we "sneak" the execution of short tasks using `QUEUEABLE` containers between the execution of long running tasks that use `GUARANTEED` ones.

We also provide results for an additional hybrid workload in [18].

8 Related Work

Mercury relates to several proposed resource management frameworks, which we discuss in this section.

Centralized Cluster resource management frameworks, such as YARN [27], Mesos [16], Omega [24] and Borg [28], are based on a centralized approach. We implemented Mercury as extension to YARN and experimentally demonstrated performance gains of a hybrid approach. Borg is similar to YARN in that it uses a logically centralized component for both resource management and scheduling. On the other hand, Mesos and Omega are geared towards supporting diverse, independent scheduling frameworks on a single shared cluster. They use a two-level scheduling model where each framework (e.g., MPI, MapReduce) pulls resources from a central resource manager, and coordinates multi-tenant job execution over these resources in an idiom isolated to that framework. Omega uses an optimistic concurrency control model for updating shared cluster state about resource allocation. This model works well for clusters that retain their resources for a reasonably long duration; a scheduling framework will almost always obtain the set of nodes it needs, retries are rare, and frameworks reach quick consensus on allocations. In contrast, our approach of dynamic load balancing works well even for heterogeneous workloads that share resources at finer granularity.

A central scheduler can reason globally about soft constraints such as data locality [17, 31], or hard constraints including multi-resource sharing [14], capacity guarantees [1] or fairness [2]. With knowledge of the workload, a central scheduler can also reason about allocations over time to effect reservation-based scheduling [11] and packing [15]. We leverage this rich body

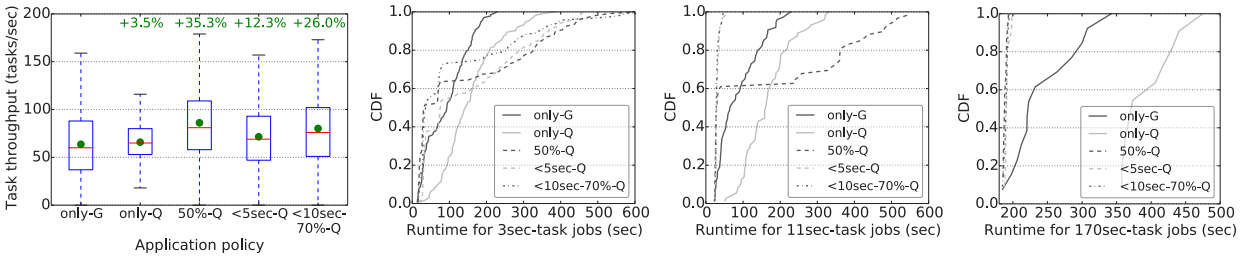


Figure 10: Task throughput and job latency CDF for Microsoft-based workload.

of work for Mercury's central scheduler. Quasar [13] accounts for both resource heterogeneity and interference during task placement, leading to near-optimal scheduling for long jobs but impacting the latency of short jobs.

HPC schedulers (e.g., SLURM [30], TORQUE [26]) are also centralized job scheduling frameworks that support at most a few hundred concurrent running jobs/sec, orders of magnitude lower than what Mercury targets.

Distributed Systems such as Apollo [9], are built using a fully decentralized approach. These schedulers achieve extreme scalability for low-latency allocations by allowing and correcting allocation conflicts. Lacking a chokepoint for throttling or coordinated feedback, fully distributed techniques maintain their invariants on an eventual manner. Worker nodes in distributed architectures maintain a queue of tasks to minimize time the node spends idle and to throttle polling. Similar to Mercury, Apollo estimates wait times at each node and lazily propagates updates to schedulers. In particular, Apollo uses a principled approach that combines optimizer statistics and observed execution behavior to refine task runtime estimates. These techniques can be incorporated by YARN AMs, which can in turn improve Mercury's placement and load balancing policies. Note that, unlike Mercury, the scheduler in Apollo is part of the SCOPE [34] application runtime, so operator policies are not enforced, updated, or deployed by the platform.

Executor model Single-framework distributed schedulers focus on a different class of workloads. Sparrow [22] and Impala [19] schedule tasks in long-running daemons, targeting sub-second latencies. This pattern is also used in YARN deployments, as applications will retain resources to amortize allocation costs [4, 29, 33] or retain data across queries [20, 32]. In contrast, Mercury not only mixes heterogeneous workloads with fine granularity, but its API also enables jobs to suitably choose a combination of guaranteed and opportunistic resources.

Performance enhancement techniques Corrective mechanisms for distributed placement of tasks are essentially designed to mitigate tail latency [12]. Sparrow uses batch sampling and late binding [22], which are demonstrably effective for sub-second queries. Apollo [9] elects to rebalance work by cloning tasks (i.e., duplicate execution), rather than shedding work from longer queues. Resources spent on duplicate work adversely affect cluster goodput and contribute to other tasks' latency. Instead, Mercury uses dynamic load shedding as its corrective mechanism.

Several Big Data schedulers have dynamically adjusted node allocations to relieve bottlenecks and improve throughput [23, 25], but the monitoring is trained on single frameworks and coordinated centrally. Principled oversubscription is another technique often applied to cluster workloads [10] with mixed SLOs. Our current approach with Mercury is intentionally conservative (i.e., no oversubscription) and already demonstrates substantial gains. We can further improve on these gains by enhancing Mercury to judiciously overcommit resources for opportunistic execution.

9 Conclusion

Resource management for large clusters and diverse application workloads is inherently hard. Recent work has addressed subsets of the problem, such as focusing on central enforcement of strict invariants, or increased efficiency through distributed scheduling. Analysis of modern cluster workloads shows that they are not fully served by either approach. In this paper, we present Mercury, a hybrid solution that resolves the inherent dichotomy of centralized-distributed scheduling.

Mercury exposes the trade-off between execution guarantees and scheduling efficiency to applications through a rich resource management API. We demonstrate experimentally how this design allows us to achieve task throughput improvements, while providing strong guarantees to applications that need them. The task throughput gains are then translated to job level performance wins by well tuned policies.

The key architectural shift we introduce, has far greater generality than we discussed in this paper. In particular, the Mercury Runtime provides a level of indirection that is being leveraged to scale YARN clusters to over 50K machines by federating multiple smaller clusters [5].

References

[1] Apache Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[2] Apache Hadoop Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[3] Apache Hadoop Project. http://hadoop.apache.org/.

[4] Apache Tez. https://tez.apache.org.

[5] Enable YARN RM scale out via federation using multiple RM's. https://issues.apache.org/jira/browse/YARN-2915.

[6] Extend YARN to support distributed scheduling. https://issues.apache.org/jira/browse/YARN-2877.

[7] Hadoop Distributed Filesystem. http://hadoop.apache.org/hdfs.

[8] Hadoop Gridmix. http://hadoop.apache.org/docs/r1.2.1/gridmix.html.

[9] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI* (2014).

[10] CARVALHO, M., CIRNE, W., BRASILEIRO, F., AND WILKES, J. Long-term SLOs for reclaimed cloud computing resources. In *SoCC* (2014).

[11] CURINO, C., DIFALLAH, D. E., DOUGLAS, C., KRISHNAN, S., RAMAKRISHNAN, R., AND RAO, S. Reservation-based scheduling: If youre late dont blame us! In *SoCC* (2014).

[12] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56* (2013), 74–80.

[13] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: resource-efficient and qos-aware cluster management. In *ASPLOS* (2014).

[14] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI* (2011).

[15] GRANDL, R., ANANTHANARAYANAN, G., KANDULA, S., RAO, S., AND AKELLA, A. Multi-resource packing for cluster schedulers. In *SIGCOMM* (2014).

[16] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

[17] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *SOSP* (2009).

[18] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. Tech. Rep. MSR-TR-2015-6, 2015.

[19] KORNACKER, M., BEHM, A., BITTORF, V., BOBROVYTSKY, T., CHING, C., CHOI, A., ERICKSON, J., GRUND, M., HECHT, D., JACOBS, M., JOSHI, I., KUFF, L., KUMAR, D., LEBLANG, A., LI, N., PANDIS, I., ROBINSON, H., RORKE, D., RUS, S., RUSSELL, J., TSIROGIANNIS, D., WANDERMAN-MILNE, S., AND YODER, M. Impala: A modern, open-source SQL engine for hadoop. In *CIDR* (2015).

[20] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC* (2014).

[21] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *PVLDB* (2010).

[22] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *SOSP* (2013).

[23] SANDHOLM, T., AND LAI, K. Dynamic proportional share scheduling in hadoop. In *Job scheduling strategies for parallel processing* (2010).

[24] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys* (2013).

[25] SHARMA, B., PRABHAKAR, R., LIM, S., KANDEMIR, M. T., AND DAS, C. R. MROrchestrator: a fine-grained resource orchestration framework for mapreduce clusters. In *Cloud Computing (CLOUD)* (2012).

[26] STAPLES, G. TORQUE resource manager. In *IEEE SC* (2006).

[27] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache hadoop YARN: yet another resource negotiator. In *SoCC* (2013).

[28] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *EuroSys* (2015).

[29] WEIMER, M., CHEN, Y., CHUN, B.-G., CONDIE, T., CURINO, C., DOUGLAS, C., LEE, Y., MAJESTRO, T., MALKHI, D., MATUSEVYCH, S., MYERS, B., NARAYANAMURTHY, S., RAMAKRISHNAN, R., RAO, S., SEARS, R., SEZGIN, B., AND WANG, J. REEF: Retainable evaluator execution framework. In *SIGMOD* (2015).

[30] YOO, A. B., JETTE, M. A., AND GRONDONA, M. SLURM: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing* (2003).

[31] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys* (2010).

[32] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[33] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *HotCloud* (2010).

[34] ZHOU, J., BRUNO, N., WU, M.-C., LARSON, P.-Å., CHAIKEN, R., AND SHAKIB, D. Scope: parallel databases meet mapreduce. *VLDB J. 21*, 5 (2012), 611–636.

Hawk: Hybrid Datacenter Scheduling

Pamela Delgado, EPFL
Florin Dinu, EPFL
Anne-Marie Kermarrec, INRIA, Rennes
Willy Zwaenepoel, EPFL

Abstract

This paper addresses the problem of efficient scheduling of large clusters under high load and heterogeneous workloads. A heterogeneous workload typically consists of many short jobs and a small number of large jobs that consume the bulk of the cluster's resources.

Recent work advocates distributed scheduling to overcome the limitations of centralized schedulers for large clusters with many competing jobs. Such distributed schedulers are inherently scalable, but may make poor scheduling decisions because of limited visibility into the overall resource usage in the cluster. In particular, we demonstrate that under high load, short jobs can fare poorly with such a distributed scheduler.

We propose instead a new hybrid centralized/distributed scheduler, called Hawk. In Hawk, long jobs are scheduled using a centralized scheduler, while short ones are scheduled in a fully distributed way. Moreover, a small portion of the cluster is reserved for the use of short jobs. In order to compensate for the occasional poor decisions made by the distributed scheduler, we propose a novel and efficient randomized work-stealing algorithm.

We evaluate Hawk using a trace-driven simulation and a prototype implementation in Spark. In particular, using a Google trace, we show that under high load, compared to the purely distributed Sparrow scheduler, Hawk improves the 50th and 90th percentile runtimes by 80% and 90% for short jobs and by 35% and 10% for long jobs, respectively. Measurements of a prototype implementation using Spark on a 100-node cluster confirm the results of the simulation.

1 Introduction

Large clusters have to deal with an increasing number of jobs, which can vary significantly in size and have very different requirements with respect to latency [4, 5]. Short jobs, due to their nature are latency sensitive, while longer jobs, such as graph analytics, can tolerate long latencies but suffer more from bad scheduling placement. Efficiently scheduling such heterogeneous workloads in a data center is therefore an increasingly important problem. At the same time, data center operators are seeking higher utilization of their servers to reduce capital expenditures and operational costs. A number of recent works [6, 7] have begun to address scheduling under high load. Obviously, scheduling in high load situations is harder, especially if the goal is to maintain good response times for short jobs.

The first-generation cluster schedulers, such as the one used in Hadoop [22], were centralized: all scheduling decisions were made in a single place. A centralized scheduler has near-perfect visibility into the utilization of each node and the demands in terms of jobs to be scheduled. In practice, however, the very large number of scheduling decisions and status reports from a large number of servers can overwhelm centralized schedulers, and in turn lead to long latencies before scheduling decisions are made. This latency is especially problematic for short jobs that are typically latency-bound, and for which any additional latency constitutes a serious degradation. For many of these reasons, there is a recent movement towards distributed schedulers [8, 14, 17]. The pros and cons of distributed schedulers are exactly the opposite of centralized ones: scheduling decisions can be made quickly, but by construction they rely on partial information and may therefore lead to inferior scheduling decisions.

In this paper, we propose Hawk, a hybrid scheduler, staking a middle ground between centralized and distributed schedulers. Attempting to achieve the best of both worlds, Hawk centralizes the scheduling of long jobs and schedules the short jobs in a distributed fashion. To compensate for the occasional poor choices made by distributed job scheduling, Hawk allows task stealing for short jobs. In addition, to prevent long jobs from monopolizing the cluster, Hawk reserves a (small) portion of the servers to run exclusively short jobs.

The rationale for our hybrid approach is as follows. First, the relatively small number of long jobs does not overwhelm a centralized scheduler. Hence, scheduling latencies remain modest, and even a moderate amount of scheduling latency does not significantly degrade the performance of long jobs, which are not latency-bound. Conversely, the large number of short jobs would overwhelm a centralized scheduler, and the scheduling latency added by a centralized scheduler would add to what is already a latency-bound job. Second, by scheduling long jobs centrally, and by the fact that these long jobs take up a large fraction of the cluster resources, the centralized scheduler has a good approximation of the occupancy of nodes in the cluster, even though it

does not know where the large number of short jobs are scheduled. This accurate albeit imperfect knowledge allows the scheduler to make well-informed scheduling decisions for the long jobs. There is, of course, the question of where to draw the line between short and long jobs, but we found that benefits result for a large range of cutoff values.

The rationale for using randomized work stealing is based on the observation that, in a highly loaded cluster, choosing uniformly at random a loaded node from which to steal a task is very likely to succeed, while finding at random an idle node, as distributed schedulers attempt to do, is increasingly less likely to succeed as the slack in the cluster decreases.

We evaluate Hawk through trace-driven simulations with a Google trace [15] and workloads derived from [4, 5]. We compare our approach to a state-of-the-art fully distributed scheduler, namely Sparrow [14] and to a centralized one. Our experiments demonstrate that, in highly loaded clusters, Hawk significantly improves the performance of short jobs over Sparrow, while also improving or matching long job performance. Hawk is also competitive against the centralized scheduler.

Using the Google trace, we show that Hawk performs up to 80% better than Sparrow for the 50th percentile runtime for short jobs, and up to 90% for the 90th percentile. For long jobs, the improvements are up to 35% for the 50th percentile and up to 10% for the 90th percentile. The differences are most pronounced under high load but before saturation sets in. Under low load or overload, the results are similar to Sparrow. The results are similar for the other traces: Hawk sees the most improvements under high load, and in some cases the improvements are even higher than those seen for the Google trace.

We break down the benefits of the different components in Hawk. We show that both reserving a small part of the cluster and work stealing are essential to good performance for short jobs, with work stealing contributing the most to the overall improvement, especially for the 90th percentile runtimes. The centralized scheduler is a key component for obtaining good performance for the long jobs.

We implement Hawk as a scheduler plug-in for Spark [23], by augmenting the Sparrow plug-in with a centralized scheduler and work stealing. We evaluate the implementation on a cluster of 100 nodes, using a small sample of the Google trace. We demonstrate that the general trends seen in the simulation hold for the implementation.

In summary, in this paper we make the following contributions:

1. We propose a novel hybrid scheduler, Hawk, combining centralized and distributed schedulers, in which the centralized entity is responsible for scheduling long jobs, and short jobs are scheduled in a distributed fashion.
2. We introduce the notion of randomized task stealing as part of scheduling data-parallel jobs on large clusters to "rescue" short tasks queued behind long ones.
3. Using extensive simulations and implementation measurements we evaluate Hawk's benefits on a variety of workloads and parameter settings.

2 Motivation

2.1 Prevalent workload heterogeneity

Workload heterogeneity is the norm in current data centers [4, 15]. Typical workloads are dominated by short jobs. Long jobs are considerably fewer, but dominate in terms of resource usage. In this paper, we precisely address scheduling for such heterogeneous workloads.

To showcase the degree of heterogeneity in real workloads, we analyze the publicly available Google trace [1, 15]. We order the jobs by average task duration. The top 10% jobs account for 83.65% of the task-seconds (i.e., the product of the number of tasks and the average task duration). Moreover, they are responsible for 28% of the total number of tasks, and their average task duration is 7.34 times larger than the average task duration of the remaining 90% of jobs.

Workload	% Long Jobs	% Task-Seconds
Google 2011	10.00%	83.65%
Cloudera-b 2011	7.67%	99.65%
Cloudera-c 2011	5.02%	92.79%
Cloudera-d 2011	4.12%	89.72%
Facebook 2010	2.01%	99.79%
Yahoo 2011	9.41%	98.31%

Table 1: Long jobs in heterogeneous workloads form a small fraction of the total number of jobs, but use a large amount of resources.

We also analyzed additional workloads described in [4, 5]. Table 1 shows the percentage of long jobs among all jobs, and the percentage of task-seconds contributed by the long jobs. The same pattern emerges in all cases, even for different providers: the long jobs account for a disproportionate amount of resource usage.

The numbers we provided also corroborate previous findings from several other researchers [2, 16, 22].

2.2 High utilization in data centers

Understanding how to run data centers at high utilization is becoming increasingly important. Resource-efficiency reduces provisioning and operational costs as

the same amount of work can be performed with fewer resources [12]. Moreover, data center operators need to be ready to maintain acceptable levels of performance even during peak request rates, which may overwhelm the data center.

Related work has approached the problem from the point of view of a single data center server [6, 7]. For a single server, the challenge is to maximize resource utilization by collocating workloads without the danger of decreased performance due to contention. As a result, several isolation and resource allocation mechanisms have been proposed, ensuring that resources on servers are well and safely utilized [19, 20].

Running highly utilized data centers presents additional, orthogonal challenges beyond a single server. The problem we are targeting consists of scheduling jobs to servers in a scalable fashion such that all resources in the cluster are efficiently used.

2.3 Challenges in performing distributed scheduling at high load

We next highlight by means of simulation why a heterogeneous workload in a loaded cluster is a challenge for a distributed scheduler. The main insight is that with few idle servers available at high load, distributed schedulers may not have enough information to match incoming jobs to the idle servers. As a result, unnecessary queueing will occur. The impact of the unnecessary queueing increases dramatically for heterogeneous workloads.

We illustrate this insight in more detail using the Sparrow scheduler, a state-of-the-art distributed cluster scheduler [14]. In Sparrow, each job has its own scheduler. To schedule a job with t tasks, the scheduler sends probes to $2t$ servers. When a probe comes to the head of the queue at a server, the server requests a task from the scheduler. If the scheduler has not given out the t tasks to other servers, it responds to the server with a task. This technique is called "batch probing". More details can be found in the Sparrow paper [14], but the above suffices for our purposes. Sparrow is extremely scalable and efficient in lightly and moderately loaded clusters, but under high load, few servers are idle, and $2t$ probes are unlikely to find them. More probes could be sent, but the paper found that this is counterproductive because of messaging overhead.

We use the same simulator employed by the Sparrow paper [14] to investigate the following scenario: 1000 jobs need to be scheduled in a cluster of 15000 servers. 95% of the jobs are considered short. Each short job has 100 tasks, and each task takes 100s to complete. 5% of the jobs are long. Each has 1000 tasks, and each task takes 20000s. The job submission times are derived from a Poisson distribution with a mean of 50s. We measure the cluster utilization (i.e., percentage of used servers) every 100s. The median utilization is 86%, and the maximum is 97.8%. This suggests that at least 300 servers (2%) are free at any time, enough to accommodate all tasks of any incoming short job.

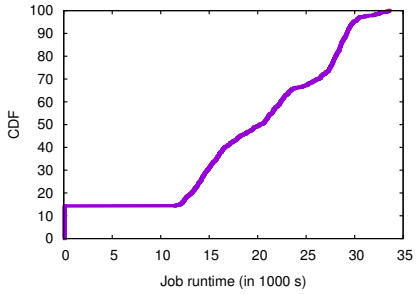


Figure 1: CDF of runtime for short jobs, in a loaded cluster, using Sparrow.

Figure 1 presents the cumulative frequency distribution (CDF) of the runtimes of short jobs. A large fraction of short jobs exhibit runtimes of more than 15000 seconds, far in excess of their execution time, which clearly indicates a large amount of queuing, mostly behind long jobs. Given that enough servers are free, an omniscient scheduler would yield job runtimes of 100s for the majority of the short jobs. With Sparrow, if all tasks are 100s long, the impact of queueing is less severe. However, a heterogeneous workload coupled with high cluster load has a strong negative impact on the performance of short jobs.

3 The Hawk Scheduler

3.1 System model

We consider a cluster composed of server (worker) nodes. A job is composed of a set of tasks that can run in parallel on different servers. Scheduling a job consists of assigning every task of that job to some server. We use the terms long task and short tasks to refer to tasks belonging to long jobs or short jobs respectively. A job completes only after all its tasks finish. Each server has one queue of tasks. When a new task is scheduled on a server that is already running a task, the task is added to the end of the queue. The server queue management policy is FIFO.

3.2 Hawk in a nutshell

The previous section demonstrated that *(i)* many cluster workloads consist of a short number of long jobs that take up the bulk of the resources and a large number of short jobs that take up only a small amount of the total resources, and *(ii)* existing distributed cluster scheduling systems, exemplified by Sparrow, do not provide good

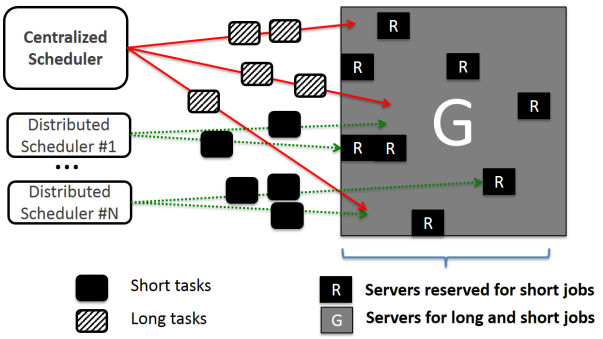


Figure 2: Overview of job scheduling in Hawk.

performance for short jobs in such an environment, due to head-of-line blocking.

In this context, Hawk's goals are:

1. to run the cluster at high utilization,
2. to improve performance for short jobs, which are the most penalized ones in highly loaded clusters,
3. to sustain or improve the performance for long jobs.

To meet these challenges, Hawk relies on the following mechanisms. To improve performance for short jobs, head-of-line blocking must be avoided. To this end, Hawk uses a combination of three techniques. First, it reserves a small part of the cluster for short jobs. In other words, short jobs can run anywhere in the cluster, but long jobs can only run on a (large) subset of the cluster. Second, to maintain low latency scheduling decisions, Hawk uses distributed scheduling of short jobs, similar to Sparrow. Third, Hawk uses randomized work stealing, allowing idle nodes to steal short tasks that are queued behind long tasks.

Finally, Hawk uses centralized scheduling for long jobs to maintain good performance for them, even in the face of reserving a part of the cluster for short jobs. The rationale for this choice is to obtain better scheduling decisions for long jobs. Since there are few long jobs, they do not overwhelm a centralized scheduler, and since they use a large fraction of the cluster resources, this centralized scheduler has an accurate view of the resource utilization at various nodes in the cluster, even if it does not know the location of the many short jobs. Figure 2 presents an overview of the Hawk scheduler.

3.3 Differentiating long and short jobs

The main idea behind Hawk is to process long jobs and short jobs differently. Two important questions are 1) how to compute a per-job runtime estimate, and 2) where to draw the line between the two categories.

Hawk uses an estimated task runtime for a job and computes it as the average task runtime for all the tasks in that job. This allows Hawk to easily classify jobs with variations in task runtime [13] without having to deal with per-task estimates. Moreover, the average task runtime is relatively robust in the face of a few outlier tasks.

Hawk compares the estimated task runtime against a cutoff (threshold). The value of the cutoff is based on statistics about past jobs because the relative proportion of short and long jobs in a cluster is expected to remain stable over time. Jobs for which the estimated task runtime is smaller than the cutoff are scheduled in a distributed fashion. This estimation-based approach is grounded in the fact that many jobs are recurring [9] and compute on similar input data. Thus, task runtimes from a previous execution of a job can inform a future run of the same job [9].

3.4 Splitting the cluster

Hawk reserves a portion of the servers to run exclusively short tasks. Long tasks are scheduled on the remaining (large) part. Short tasks may be scheduled on the whole set of servers. This allows short tasks to take advantage of any idle servers in the entire cluster. Henceforth we use the term *short partition* to refer to the set of servers reserved for short jobs and the term *general partition* to refer to the set of servers that can run both types of tasks.

If long tasks were scheduled on any server in the cluster, this may severely impact short jobs when short tasks end up queued after long tasks. A particularly detrimental case occurs when a long job has more tasks than servers or when several long jobs are being scheduled in rapid succession. In this case, every server in the cluster ends up executing a long task, and short tasks have no choice but to queue after them.

Hawk sizes the general partition based on the proportion of time that cluster resources are used by long jobs. For example, from Table 1 Hawk uses the percentage of task-seconds.

3.5 Scheduling short jobs

Hawk maintains low-latency scheduling for short jobs by relying on a distributed approach. Typically, each short job is scheduled by a different scheduler. For scalability reasons, these distributed schedulers have no knowledge of the current cluster state and do not interact with other schedulers or with the centralized component.

Distributed schedulers schedule tasks on the entire cluster. The first scheduling step is achieved as in Sparrow. To schedule a job with t tasks, a distributed scheduler sends probes to $2t$ servers. When a probe comes to the head of a server's queue, the server requests a task from the scheduler. If the scheduler has not given out the t tasks to other servers, it responds to the server with a task. Otherwise, a cancel is sent.

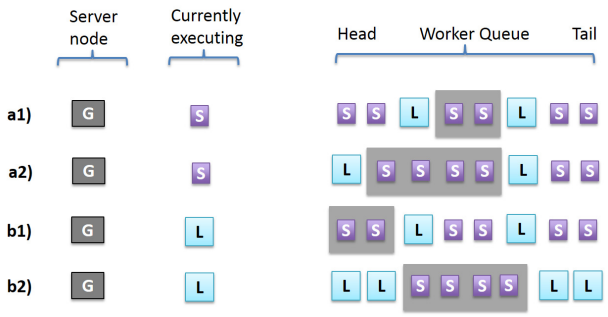


Figure 3: Task stealing in Hawk. L = Long task, S = Short task. Stolen tasks are on the dark background.

3.6 Randomized task stealing

Hawk uses task stealing as a run-time mechanism aimed at mitigating some of the delays caused by the occasionally suboptimal, distributed scheduling decisions. Since the distributed schedulers are not aware of the content of the server queues, they may end up scheduling short tasks behind long tasks. In a highly loaded cluster, the probability of this event happening is fairly high. Even if a short job is scheduled using twice as many probes as tasks, if more than half of the probes experience head-of-line blocking, then the completion time of the short job takes a big hit.

Hawk implements a randomized task stealing mechanism, that leverages the fact that the benefit of stealing arises in highly loaded clusters. In such a cluster a random selection very likely returns an overloaded server. Indeed, if 90% of the servers are overloaded, a uniform random probe has 90% probability of returning an overloaded server from which tasks are stolen.

The cluster might reach a point where many servers in the general partition are occupied by long tasks and also have short tasks in their queues, while other servers lie idle. Hawk allows such idle servers to steal tasks from the over-subscribed ones. This works as follows: whenever a server is out of tasks to execute, it randomly contacts a number of other servers to select one from which to steal short tasks. Both the servers from the general partition and the servers from the short partition can steal, but they can only steal from servers in the general partition, because that is where the head-of-line blocking is caused by long jobs.

Task stealing in Hawk proceeds as follow: The first consecutive group of short tasks that come after a long task is stolen. To see this in more detail, consider Figure 3. In cases a1) and a2) a server currently is executing a short job. The short tasks that it provides for stealing come after the first long job in the queue. In cases b1) and b2) the server is executing a long task. The short tasks stolen come immediately after that long task. Even though that long task is being executed already and has made some progress to completion, it is still likely that it will delay the short tasks queued behind it.

With our design we want to increase the chance that stealing actually leads not only to an improvement in task runtime but also in job runtime. Consider a job that has completed all but two of its tasks. Stealing just one of these tasks improves that task's runtime, but the job runtime is still determined by the completion time of the last task (the one not stolen). As shown in Figure 3, Hawk steals a limited number of tasks and starts from the head of the queue when deciding what to steal. Thus, stealing focuses on a few short jobs, increasing the chance that the runtime of those jobs benefits. If short tasks were stolen from random positions in server queues that would likely end up focusing on too many jobs at the same time while failing to improve most.

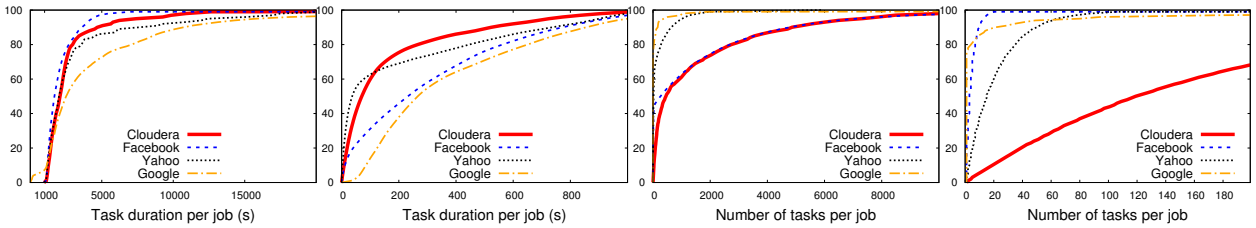
3.7 Scheduling long jobs

The final technique used in Hawk is to schedule long jobs in a centralized manner. Long jobs are only scheduled in the general partition, and the centralized component has no knowledge of where the short tasks are scheduled. This centralized approach ensures good performance for long jobs for three reasons. First, the number of long jobs is small, so the centralized component is unlikely to become a bottleneck. Second, long jobs are not latency-bound, so they are largely unaffected even if a moderate amount of scheduling latency occurs. Third, by scheduling long jobs centrally and by the fact that these long jobs take up a large fraction of the cluster resources, the centralized component has a timely and fairly accurate view of the per-node queueing times regardless of the presence of short tasks.

The centralized component keeps a priority queue of tuples of the form $< server, waiting\ time >$. The priority queue is kept sorted according to the waiting time. The waiting time is the sum of the estimated execution time for all long tasks in that server's queue plus the remaining estimated execution time of any long task that currently may be executing. When a new job is scheduled, for every task, the centralized allocation algorithm puts the task on the node that is at the head of the priority queue (the one with the smallest waiting time). After every task assignment, the priority queue is updated to reflect the waiting time increase caused by the job that is being scheduled. The goal of this algorithm is to minimize the job completion time for long jobs.

3.8 Implementation

We implement Hawk as a scheduler plug-in for Spark [23], by augmenting the Sparrow scheduler with a centralized scheduler and work stealing. To realize work stealing we enable the Sparrow node monitors to com-



(a) Long jobs. (b) Short jobs. (c) Long jobs. (d) Short jobs.

Figure 4: Workload properties. CDFs of average task duration and number of tasks per job.

municate and send tasks to each other. The node monitors communicate via the Thrift RPC library.

4 Evaluation

We compare Hawk with Sparrow, a state-of-the-art fully distributed scheduler. We show that in loaded clusters Hawk outperforms Sparrow for both long and short jobs. The benefits hold across all workloads. We also show that Hawk compares well to a centralized scheduler.

4.1 Methodology

Workloads We use the publicly available Google trace [1, 15]. After removing invalid or failed jobs and tasks we are left with 506460 jobs. Task durations vary within a given job. The estimated task execution time for a job is the average of its task durations.

We create additional traces using the description of the Cloudera C and Facebook 2010 workloads from [4] and Yahoo 2011 workload from [5]. We only consider the mapper tasks from these workloads, since many jobs do not have reducers. In [4, 5] the workloads are described as k-means clusters, and the first cluster is deemed composed of short jobs. We consider the rest of the clusters to be long jobs. For each cluster we derive the centroid values for the average number of tasks per job and the duration of the tasks by combining the information on task-seconds from [4, 5] with the job to mapper duration ratios in [22]. We then use the derived centroid values as the scale parameter in an exponential distribution in order to obtain the number of tasks and the mean task duration for each job. Given the mean task duration we derive task runtimes using a Gaussian distribution with standard deviation twice the mean, excluding negative values.

Figures 4a, 4b, 4c and 4d show the CDFs of the duration of tasks and the number of tasks per job for both long and short jobs. Table 2 shows additional trace properties. The trace properties differ from trace to trace. This is expected, as workload properties are known to vary depending on the provider [2, 4, 5].

Simulator We augment the event-based simulator used to evaluate Sparrow [14]. The input traces contain tuples of the form: (jobID, job submission time, number of tasks in the job, duration of each task). Network delay is assumed to be 0.5ms. The scheduling decisions and the task stealing do not incur additional costs.

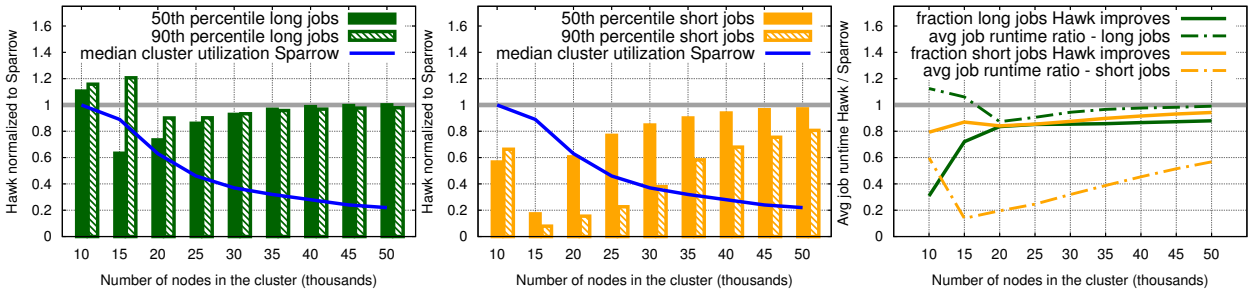
Workload	% Long Jobs	Total number jobs
Google 2011	10.00%	506460
Cloudera-c 2011	5.02%	21030
Facebook 2010	2.01%	1169184
Yahoo 2011	9.41%	24262

Table 2: Number of long jobs and total number of jobs.

Real cluster run We use a 100-node cluster with 1 centralized and 10 distributed schedulers. We use a subset of 3300 jobs from the Google trace. To obtain task runtimes proportional to the ones in the Google trace, we scale down task duration by 1000x (i.e., sec. to msec.) and use these durations in a sleep task. We also scale down the number of tasks per job by keeping constant the ratio between the cluster size and the largest number of tasks in a job. When we scale down the number of tasks in a job, we compensate by proportionally increasing the duration of the remaining tasks in order to keep the same task-seconds ratio as the original trace. We vary the cluster load by varying the mean job inter-arrival rate as a multiple of the mean task runtime. We use this mean to generate job inter-arrival times according to a Poisson distribution.

Parameters By default, in Hawk, a node performs task stealing by randomly contacting 10 other nodes and stealing from the first node that has short tasks eligible for stealing. We compare against Sparrow configured to send two probes per task because the authors of Sparrow [14] have found two to be the best probe ratio. Each simulated cluster node has 1 slot (i.e., can execute only one task at a time). This is analogous to having multi-slot nodes with each slot served by a different queue. Following the task-second proportion between long and short jobs, the short partition comprises 17% of the nodes for the Google trace and 9%, 2% and 2% for the Cloudera, Facebook and Yahoo traces, respectively.

Metrics When comparing Hawk to another approach X, we mostly take the ratio between the 50th (or



(a) Google trace long jobs. (b) Google trace short jobs. (c) Google trace additional metrics.

Figure 5: Google trace. Hawk normalized to Sparrow. Figure (c) shows two additional metrics: (1) percentage of jobs for which Hawk is equal or better to Sparrow and (2) average job runtime.

90th) percentile job runtime for Hawk and the 50th (or 90th) percentile job runtime time for X. Consequently, our results are normalized to 1. We do this separately for short and long jobs. Additional metrics are explained with the corresponding results. In all figures lower values are better.

Repeatability of results The results for the 50th and 90th percentiles are stable across multiple runs, and for this reason we do not show confidence intervals. We have seen variations in the maximum job runtime for short tasks. This is expected, as failing to steal one task can make a big difference in job runtime.

4.2 Overall results on the Google trace

We take the Google trace and vary the number of server nodes in order to vary cluster utilization. We find that Hawk consistently outperforms Sparrow, especially in a highly loaded cluster. Figures 5a and 5b illustrate the improvements in job runtime for long jobs and short jobs, respectively as a function of the number of machines in the cluster. The cluster utilization is based on snapshots taken every 100s.

Hawk shows significant improvements when the cluster is highly loaded but not overloaded (i.e., 15000 - 25000 nodes), since both the centralized scheduler and the task stealing algorithm make efficient use of any idle slots. In the best cases, Hawk improves the 50th and 90th percentile runtimes by 80% and 90% for short jobs and by 35% and 10% for long jobs. Hawk improves short job runtime at the 90th percentile more than at the 50th percentile, because these jobs are more affected by queueing. Stealing a few (even one) short tasks experiencing head-of-line blocking can greatly improve short job completion time.

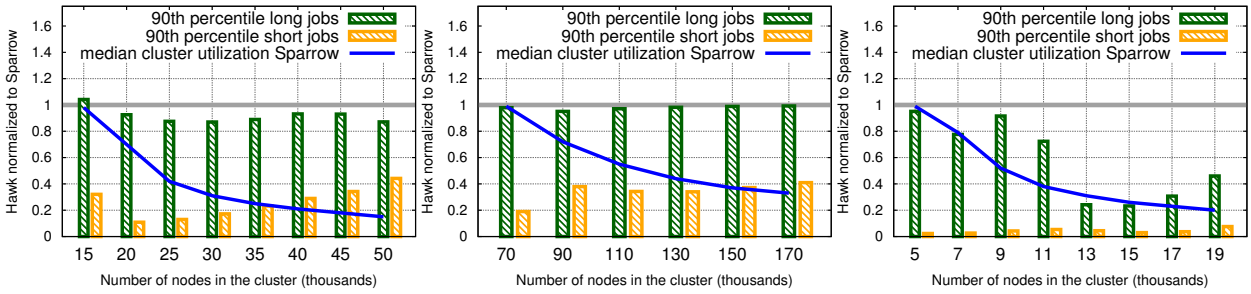
Figure 5c presents additional metrics: the percentage of jobs for which Hawk provides performance better than or equal to Sparrow and the average job runtime for Hawk vs. Sparrow. The average job runtime for short jobs is significantly better for Hawk and is as low as a factor of 7. For 15000 nodes we present additional details, not all pictured: Hawk improves the runtime of 68% of short jobs, while for 59% of short jobs the improvement is more than 50%. Overall, for 86% of short jobs, Hawk is better or equal to Sparrow. For long jobs, Hawk improves 51% of jobs and is better or equal to Sparrow for 72% of jobs.

Small clusters (10000 nodes) tend to be overwhelmed by the high job submission rate in the trace. As a result, the node queues become progressively longer and waiting times keep increasing. We do not believe that any cluster should be run at this overload, but the case is nevertheless interesting to understand. Hawk is just slightly worse for long jobs, as the long jobs in Hawk are scheduled only in the general partition, while in Sparrow they can be scheduled across the entire cluster. Conversely, Hawk is better for short jobs because of the randomized stealing, but the improvement is small. The short partition is overloaded, and its nodes have few opportunities to steal short tasks experiencing head-of-line blocking in the general partition. As the cluster size increases (40000+ nodes), the benefits of Hawk decrease as the cluster becomes mostly idle. Any scheduler is likely to do well in that case.

4.3 Overall results on additional traces

Figures 6a, 6b and 6c show the results for the workloads derived from Facebook, Cloudera and Yahoo data. Hawk's benefits hold across all traces. At the median (not pictured), Hawk also improves on Sparrow across all simulated cluster sizes.

The most important difference compared to the Google trace is the larger improvement for short jobs. This can be traced back to the utilization of the short partition. In the Facebook, Cloudera and Yahoo traces the short partition is less utilized compared to the Google trace so there are more chances for stealing.



(a) Cloudera trace. (b) Facebook trace. (c) Yahoo trace.

Figure 6: Cloudera, Facebook and Yahoo traces. Long and short jobs. Hawk normalized to Sparrow.

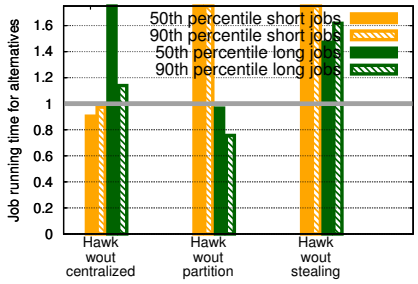


Figure 7: Break-down of Hawk's benefits normalized to Hawk. 15000 nodes. Google trace.

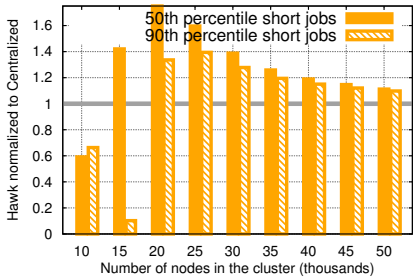


Figure 8: Hawk normalized to centralized approach, short jobs. Google trace.

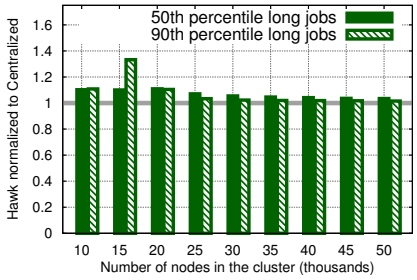


Figure 9: Hawk normalized to centralized approach, long jobs. Google trace.

4.4 Breaking down Hawk's benefits

This subsection analyzes the impact of each of the major components of Hawk: work stealing, reserving cluster space for short jobs and using centralized scheduling for the long jobs. We find that the absence of any of the components reduces the performance of Hawk for either long or short jobs.

Figure 7 shows the results of the Google trace normalized to Hawk with all components enabled. Without centralized scheduling for long jobs the performance of long jobs takes a significant hit, as tasks of different long jobs queue one after the other. The performance of short jobs improves due to the decrease in the performance for long jobs. As the placement of long jobs is not optimized in the general partition, fewer short tasks encounter queueing there.

Without partitioning the cluster, the short jobs are impacted, because they can be stuck behind long tasks on any node. For long jobs, the performance slightly increases, because they can be scheduled on more nodes. Without task stealing both short and long jobs suffer. The short jobs are greatly penalized, because some of their tasks are stuck behind long tasks. The long tasks are penalized, because they share the queues with more short tasks.

4.5 Hawk vs. a fully centralized approach

We next look at the performance of Hawk compared to an approach that schedules all jobs (long and short) in a centralized manner. We find that Hawk is competitive, while not suffering from the scalability concerns that plague centralized schedulers.

This centralized scheduler does not reserve part of the cluster for short jobs and does not use work stealing. It uses the algorithm we presented in subsection 3.7 for all jobs. Figures 8 and 9 show Hawk normalized to the centralized scheduler's performance using the Google trace.

The centralized scheduler penalizes short jobs (Figure 8), when the cluster is heavily loaded (10000-15000 nodes). This is because in periods of overload the centralized scheduler does not have many options and

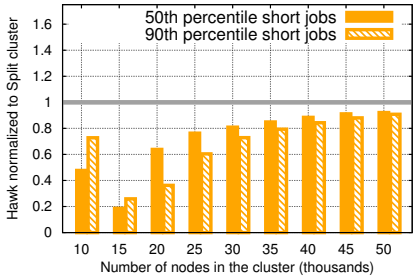


Figure 10: Hawk normalized to split cluster, short jobs. Google trace.

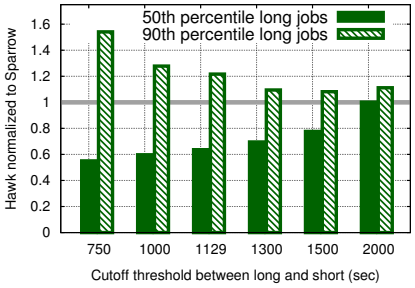


Figure 12: Effect of varying cutoff, Hawk normalized to Sparrow, long jobs. 15000 nodes. Google trace.

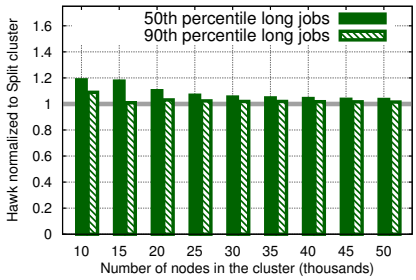


Figure 11: Hawk normalized to split cluster, long jobs. Google trace.

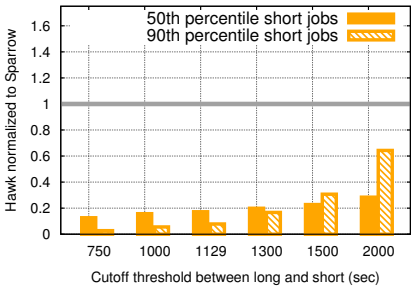


Figure 13: Effect of varying cutoff, Hawk normalized to Sparrow, short jobs. 15000 nodes. Google trace.

queues short tasks behind long ones. This is especially the case when long jobs are present in every node in the cluster. In Hawk short tasks benefit from stealing and from running on reserved nodes. As the cluster utilization decreases, the centralized scheduler does an increasingly better job for short jobs. When the cluster becomes lightly loaded (50000 nodes), the results for both approaches begin to converge.

For long jobs the centralized approach performs slightly better (Figure 9), because they can use the entire cluster. In Hawk they only use the general partition.

4.6 Hawk compared to a split cluster

We now compare Hawk to a split cluster, in which a long partition only runs long jobs and a short partition only runs short jobs. In other words, there is no general partition, in which both short and long jobs can execute. Hawk fares significantly better for short jobs, while being competitive for long jobs.

We use the Google trace. The split cluster uses 17% of the cluster for the short partition, and the remaining 83% is reserved for long jobs (long partition). The split cluster uses centralized scheduling for the long partition and distributed scheduling for the small one.

Figures 10 and 11 show the results. For long jobs, the split cluster performs slightly better, because the short jobs do not take up the space in the general partition. However, this comes at the cost of greatly increasing runtime for short jobs. For short jobs, for small cluster sizes, the relative degradation for the split cluster is smaller, because both approaches suffer from significant queueing delays. In the other extreme, for a large cluster, both approaches do well. In between, the split cluster shows extreme degradation, because short tasks cannot leverage the general partition nodes.

4.7 Sensitivity to the cutoff threshold

Next we vary the cutoff point between short and long jobs. Hawk yields benefits for a range of cutoff values, showing that it does not depend on the precise cutoff chosen.

The cluster size is 15000 nodes in this experiment, and we use the Google trace. Figures 12 and 13 show the results for long and short jobs, respectively. The percentage of short jobs increases as the cutoff increases. Thus, for the smaller cutoffs, Hawk improves the most on Sparrow because the short partition is underloaded and can steal more tasks. The percentage of long jobs increases as the cutoff decreases. For the smaller cutoffs the 90th percentile long job runtime is affected more for Hawk compared to Sparrow, because Sparrow is able to relieve some of the queueing among long jobs by scheduling them over the entire cluster.

4.8 Sensitivity to task runtime estimation

Hawk's centralized component schedules long jobs according to an estimate of the average task runtime for that job. We next analyze how inaccuracies in estimat-

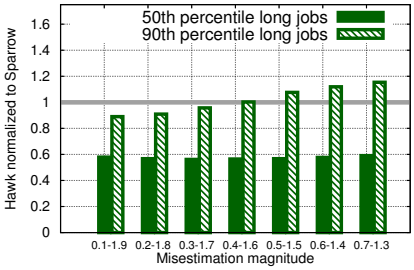


Figure 14: Hawk with varying mis-estimation magnitude normalized to Sparrow, long jobs. 15000 nodes. Google trace.

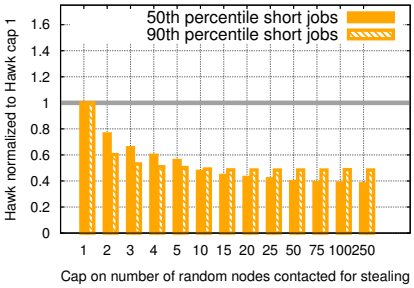


Figure 15: Hawk with varying number of stealing attempts normalized to Hawk capped at 1 attempt, short jobs. 15000 nodes. Google trace.

ing the average affect the results. For each job, to obtain the inaccurate estimate, we multiply the correct estimate with a random value, chosen uniformly within a range given as a parameter (e.g., 0.1-1.9). Figure 14 shows the job runtimes normalized to Sparrow for the set of jobs classified as long when no mis-estimations are present. These results are averaged over ten runs.

Hawk is robust to mis-estimations. The mis-estimation results in some long jobs being classified as short and vice-versa. This is more likely to happen for long and short jobs for which the estimation is comparable to the cutoff. Since these jobs are fairly similar in nature, the two opposing mis-classifications (long as short and short as long) tend to cancel each other. Moreover, most jobs are not mis-classified, because their estimation significantly differs compared to the cutoff. In Figure 14, long jobs perform better at the 90th percentile as the mis-estimation magnitude increases because more long jobs are classified as short. At 15000 nodes the short partition is less loaded than the general partition so the long jobs classified as short benefit from the additional, less-loaded nodes in the short partition.

Short jobs are not directly impacted by mis-estimations, since their scheduling does not rely on estimations. Short jobs can be indirectly impacted by the changes in the scheduling of the long jobs. In the experiments, we only see minute variations for the results for short jobs (not pictured).

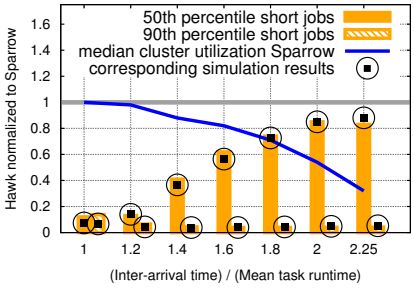


Figure 16: Implementation vs simulation, short jobs. 3300 job sample from the Google trace.

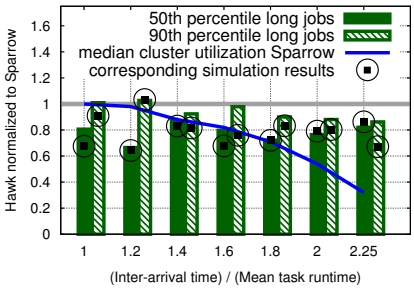


Figure 17: Implementation vs simulation, long jobs. 3300 job sample from the Google trace.

4.9 Sensitivity to stealing attempts

We now vary the maximum number of nodes that an idle node can contact for stealing. We find that performance increases with an increase in the cap value, but even a low value (e.g., 10) gives significant benefit.

Figure 15 shows the results normalized to Hawk using a cap of 1. As expected, increasing the cap also increases performance, as it increases the chance for successful stealing. At high cap values there is also a slight increase in the performance of long jobs (not pictured), because they wait behind fewer short tasks. The improvement for long jobs is small, because of the large relative difference between the resource usage of long jobs compared to short jobs.

4.10 Implementation vs. simulation

Figures 16 and 17 show the results for a 3300-job sample of the Google trace. In the implementation, Hawk schedules 3000 short jobs in a distributed way (300 per each of the 10 distributed schedulers) and 300 long jobs in a centralized fashion. The simulation and implementation experiments agree and show similar trends. Hawk is best at high loads, when it significantly improves on Sparrow for short jobs, while maintaining good performance for long jobs. As load decreases, the 50th percentiles for Hawk and Sparrow become similar, as fewer

jobs suffer from queueing. Even at medium load, the 90th percentile is still considerably better for Hawk for short jobs, since those jobs suffer from queueing in Sparrow but not in Hawk.

The simulation and implementation results do not perfectly match, because the simulation does not model overheads for scheduling or stealing. Moreover, some Spark tasks sleep very little (a few msec) and are sensitive to slight inaccuracies in sleeping time and to various system overheads (message exchanges, network delays).

5 Related Work

The first data center schedulers had a monolithic design [22], which lead to scalability concerns [20]. Second generation schedulers (YARN [20], Mesos [10]) use a two-level architecture, which decouple resource allocation from application-specific logic such as task scheduling, speculative execution or failure handling. However, the two-level architecture relies on a centralized resource allocator, which can still become a scalability bottleneck in large clusters. In contrast, Hawk schedules most jobs in a distributed manner minimizing the scalability concerns.

We compared against Sparrow [14] in this paper. Sparrow is a fully distributed scheduler that performs well for lightly and medium loaded clusters. However, it is challenged in highly loaded clusters, especially for heterogeneous workloads, because tasks experience unnecessary queueing. This is due to Sparrow's design, which is geared at extreme scalability and cannot fully benefit from load information when making scheduling decisions. Moreover, Sparrow does not have runtime mechanisms to compensate in case the initial assignment of tasks to nodes is suboptimal.

In Apollo [3], distributed schedulers utilize global cluster information via a loosely coordinated mechanism. Apollo does not differentiate between long and short jobs and uses the same mechanisms to schedule both types of jobs. Apollo has built-in, node-level correction mechanisms to compensate for inaccurate scheduling decisions. If a task is queued longer than estimated at scheduling time, then Apollo starts duplicate copies of the task on other nodes. In contrast, work stealing in Hawk works at the level of the entire cluster. Even if the queueing time for a task has been correctly predicted, the task can be stolen by another server that becomes idle.

Mercury [11] is parallel work on designing a hybrid scheduler. In Mercury, jobs can choose between guaranteed (non-preemtable, non-queueable, centrally-allocated) containers and queueable containers (preemptable, allocated in a distributed way). However, it is not clear whether jobs have the information necessary to make an informed choice with respect to the appropriate container type. In Mercury, distributed schedulers loosely coordinate with a coordinator to obtain per-node load information. In Hawk, the distributed schedulers make completely independent decisions.

Omega [17] supports multiple concurrent schedulers which have full access to the entire cluster. The schedulers compete in a free-for-all manner, and use optimistic concurrency control to handle conflicts when they update the cluster state. Omega is designed to support at most tens of schedulers and this may prove insufficient to ensure low latency scheduling for very short jobs. Borg [21] uses a logically centralized controller but employs replication to improve availability and scalability. Borg's scheduling design is similar to Omega's optimistic concurrency control.

HPC and Grid schedulers [18] use centralized scheduling and do not have the same latency requirements. The jobs they schedule are usually compute-intensive and often long running. These jobs come with several constraints as they are tightly coupled in nature, requiring periodic message passing and barriers.

6 Conclusions and Future Work

In this paper we address the problem of efficient scheduling in the context of highly loaded clusters and heterogeneous workloads composed of a majority of short jobs and a minority of long jobs that use the bulk of the resources. We propose Hawk, a hybrid scheduling architecture. Hawk schedules only the long jobs in a centralized manner, while performing distributed scheduling for the short jobs. To compensate for the occasional poor choices made by distributed job scheduling, Hawk uses a novel randomized task stealing approach. With a Spark-based implementation and with large scale simulations using realistic workloads we show that Hawk outperforms Sparrow, a state-of-the-art fully distributed scheduler, especially in the challenging scenario of highly loaded clusters.

Acknowledgement

We thank the anonymous reviewers and our shepherd David Shue for their feedback. We also thank Laurent Bindschaedler, Christos Gkantsidis, Calin Iorgulescu, Konstantinos Karanasos, Sergey Legtchenko, Amitabha Roy, Malte Schwartzkopf and John Wilkes for the discussions, feedback and help. Additional thanks go to the Sparrow authors for making their simulator available. This research has been supported in part by a grant from Microsoft Research Cambridge.

References

[1] J. Wilkes - More Google cluster data. http://googleresearch.blogspot.ch/2011/11/more-google-cluster-data.html.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *Proc. NSDI 2012*.

[3] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. OSDI 2014*.

[4] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. In *Proc. VLDB 2012*.

[5] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proc. MASCOTS 2011*.

[6] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS 2013*.

[7] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proc. ASPLOS 2014*.

[8] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. In *Proc. SIGCOMM 2014*.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. EuroSys 2012*.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proc. NSDI 2011*.

[11] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. Usenix ATC 2015*.

[12] C. Kozyrakis. Resource efficient computing for warehouse-scale datacenters. In *Proc. DATE 2013*.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. In *Proc. OpenCirrus Summit 2011*.

[14] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. SOSP 2013*.

[15] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. SoCC 2012*.

[16] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. In *Proc. VLDB 2013*.

[17] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. EuroSys 2013*.

[18] G. Staples. Torque resource manager. In *Proc. SuperComputing 2006*.

[19] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: a software-defined storage architecture. In *Proc. SOSP 2013*.

[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOCC 2013*.

[21] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. EuroSys 2015*.

[22] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proc. EuroSys 2010*.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI 2012*.

Bolt: Faster Reconfiguration in Operating Systems

Sankaralingam Panneerselvam[1], Michael M. Swift[1], and Nam Sung Kim[2]

[1]Department of Computer Sciences
[2]Department of Electrical and Computer Engineering
University of Wisconsin-Madison
[1]{sankarp,swift}@cs.wisc.edu
[2]nskim3@wisc.edu

Abstract

Dynamic resource scaling enables provisioning extra resources during peak loads and saving energy by reclaiming those resources during off-peak times. Scaling the number of CPU cores is particularly valuable as it allows power savings during low-usage periods. Current systems perform scaling with a slow *hotplug* mechanism, which was primarily designed to remove or replace faulty cores. The high cost of scaling is reflected in power management policies that perform scaling at coarser time scales to amortize the high reconfiguration latency.

We describe *Bolt*, a new mechanism built on existing hotplug infrastructure to reduce scaling latency. Bolt also supports a new bulk interface to add or remove multiple cores at once. We implemented Bolt for x86 and ARM architectures. Our evaluation shows that Bolt can achieve over 20x speedup for entering offline state. While turning on CPUs, Bolt achieve speedups of 1.3x and 21x for x86 and ARM. The speedup is limited by high latency hardware intialization. On an ideal processor with zero-latency initialization, the speedup on x86 rises to 10x.

1 Introduction

Most operating system policies focus on improving performance given a *fixed* set of resources. For example, schedulers assume a fixed set of processors to which assign threads, and memory managers assume a fixed pool of memory. However, this assumption is increasingly violated in the current computing landscape, where resources can be added or removed dynamically during runtime for reasons of energy reduction, cost savings, virtual-machine scaling or hardware heterogeneity [11].

We refer to changing the set of resources as *resource scaling* and our work focuses on scaling the set of processors available to the operating system. This scaling may be helpful in virtualized settings such as cloud computing [7] and disaggregated servers [14, 2] where it is possible to add or remove CPUs at anytime within a system. Scaling can improve performance during peak loads and minimize energy during off-peak loads [8]. Dynamically reconfigurable processors (e.g., [16]), which allow processing resources to be reconfigured at runtime to meet application demands, can also benefit from scaling the set of available CPUs [21].

Current operating systems assume that the processor cores available to them are essentially static and almost never change over runtime of the system. These systems support scaling through a hotplug mechanism that was primarily designed to remove faulty cores from the system [10]. This mechanism is slow, bulky and halts the entire machine for a few milliseconds while the OS reconfigures. The current Linux hotplug mechanism takes tens of milliseconds to reconfigure the OS [21]. In contrast, processor vendors are aiming for transitions to and from sleep states in the order of microseconds [25, 23], making OS the bottleneck in scaling. In spite of these drawbacks, hotplug is being widely used by mobile vendors as a means to scale the number of processor cores to save energy [22] due to the lack of better alternatives.

We propose a new mechanism, Bolt, that builds on the current Linux hotplug infrastructure with the assumption that scaling events are frequent and a goal of low latency scaling mechanism. Bolt classifies every operation carried out during hotplug as critical or non-critical operations. The critical operations are performed immediately for proper functioning of the system whereas non-critical operations are done lazily. Bolt also supports a *new bulk interface* to turn on/off multiple cores simultaneously, which is not supported by the current hotplug mechanism.

We implemented Bolt for both x86 and ARM processors, and find that Bolt can achieve over 20x better latency over native hotplug offline mechanism. For getting a CPU to online state, speedup is limited to 1.3x for x86 due to hardware overhead whereas the software overhead is reduced by 10x. The concurrency allowed through the bulk interface achieves speedup of 4x-67x when adding or removing 3 cores.

2 Motivation

2.1 Processor Scaling

Many current and future systems will support a dynamic changing set of cores for three major reasons.

Energy Proportionality. This property dictates that energy consumption should be proportional to the amount of work done and is getting a major focus in all forms on computing from cloud to mobile devices. Low power (P-states) and sleep state (C-states) support from processors help achieve energy proportionality by reducing energy consumption when the processor is not fully utilized. However, in many processors further power savings could be achieved by turning off cores to a deeper sleep state, turning off an entire socket or allowing to enter package level sleep state. These deeper states require OS intervention, as the core is logically turned off and not available for scheduling threads or processing interrupts. For example, most mobile systems turn off cores during low system utilization to conserve battery capacity by reducing static power consumption. Some processors (e.g. Exynos [18]) provide a package-level deep sleep state that is enabled when all but one core is switched off. Operating systems may use core parking [12] to consolidate tasks in a single socket to switch off other sockets completely. Quick scaling support by the OS can allow rapid transitions into and out of deep sleep states.

Heterogeneity. Many processors support heterogeneity either statically [1] via different core designs or dynamically [16] through reconfiguration. On dark silicon systems [5], not all processors could be used at full performance together, and hence the OS must decide which processors to enable based on the application characteristics. Processors like Exynos [3] and Tegra [20] employ ARM's big.LITTLE architecture, with a mix of high performance and high efficiency cores. In systems with these processors, the OS must choose the type of processor core based on the performance need. The OS must change the processor set when switching between different CPU types.

VM Scaling. Virtual machines are widely used in cloud environment, and many hypervisors provide support for scaling the number of virtual CPUs in a virtual machine (VM) [19]. IBM supports VM scaling through DLPAR (dynamic logical partitioning [17]) and VMware supports them through hot add/remove interfaces. Some application like databases benefit from scaling up of virtual machines by provisioning more resources rather than scaling out where more virtual machines are spawned. These techniques can be used at finer scale if the guest OS provides quick processor scaling.

2.2 OS Support

There are several mechanisms an OS can use to scale the number of CPUs in use.

Virtualization. An extra layer of indirection through virtualization decouples the physical execution layer from rest of the operating system and exposes only virtual CPUs to the OS. To scale down, multiple virtual CPUs (VCPU) can be multiplexed on a single physical CPU (PCPU). In terms of latency, virtualization could provide an ideal support where it could switch VCPUs from a PCPU that is being switched off to a different physical CPU instantly by saving and restoring context of those virtual CPUs.

However, the drawbacks of using virtualization-based techniques are two-fold. First, virtualization adds overhead in the common case, particularly for memory access [6]. Second, multiplexing VCPUs on a physical CPU can hurt performance due to context switches during critical sections [26].

Power Management. OS support for idle sleep states (C-states) can be used to move unused cores to a sleep state and wake up when needed. The latency of entering and exiting such sleep state is very low when compared to the hotplug mechanism. However, OS power management support requires that cores can still respond to interrupts, which is not the case for all deep sleep states or for non-power uses of scaling. Furthermore, the OS may accidentally wake up a sleeping core unnecessarily to involve them in regular activities like scheduling or TLB shootdowns [24].

Processor Proxies. Chameleon [21] proposed a new alternative to hotplug called processor proxies that is several times faster than hotplug. A proxy represents an offline CPU and runs on an active CPU making the system believe that the offline CPU is still active. However, proxies can only be used for a short period because they handle interrupts and Read-Copy-Update (RCU) operations only, and do not reschedule threads from a CPU in offline state.

Scalability-Aware Kernel. Ideally, an OS kernel could natively support changing the set of CPUs at low latency and with low overhead. Rather than assuming that scaling events are rare, a scalability-aware kernel would spend little time freeing resources during a scale-down event when they are likely to be re-allocated during an upcoming scale-up event.

2.3 Hotplug

Hotplug is a widely used mechanism available in Linux to support processor set scaling. It offers interfaces for any kernel subsystem to subscribe to notifications for processor set changes. However, handling of notifications by every subsystem follow the assumption that hotplug events are rare. The shortcomings of the mechanism are discussed below.

Repeat Execution. A direct implication of the above assumption is that most kernel subsystems free or reinitialize the software structures during hotplug event. Out of the 50 subscriptions to the hotplug from various subsys-

tems, 8 of them remove or initialize `sysfs` structures and 14 of them to free and create software structures needed for the subsystem. However, all these operations become redundant if the CPU set changes frequently.

Synchronous. All operations performed in response to the notifications are synchronous. For example, subsystems like slab allocator frees the slab memory from its per-CPU queue when the CPU is moved to offline state, per-CPU statistics values are aggregated into a global structure, and the hotplug operation is blocked until system threads move into sleep state. However, these operations need not be synchronous for the correct execution of the system.

Hotplug Prevention. Hotplug events can be prevented by disabling preemption or interrupts on any CPU, similar to grabbing a lock. So, the hotplug mechanism ensures that preemption and interrupts are enabled on all CPUs by scheduling a special kernel thread on *every CPU* in the system. This special form of locking (through preemption) avoids the overhead of acquiring a lock and releasing during normal execution.

As a result of these properties, Linux's current hotplug mechanism is too slow for rapid scaling. As we show in Section 4, it takes orders of milliseconds to reconfigure, while current hardware can transition from sleep states in the order of *microseconds* [25].

3 Bolt

Bolt is a reconfiguration mechanism that can be used as a replacement for hotplug. The functionality of Bolt is similar to that of hotplug in getting the system from one stable state to another after processor scaling. However, Bolt aims to offer stability at very low latency and is built by refactoring the existing hotplug infrastructure.

Bolt achieves low latency by separating hotplug notifications into *critical* and *non-critical* operations. The former needs to be handled synchronously to ensure correctness of the system whereas the latter could be removed from the critical path and performed after the CPU goes online/offline.

3.1 Critical Operations

Every action taken by hotplug, including handling of notifications by kernel subsystems is classified based on its criticality. Bolt defines critical operations as those that need to be executed immediately for correct running of the system.

State Migration. In the event of CPU removal, important software state associated with that CPU has to be migrated to another active CPU. Such software states include softirq or bottom halves and threads in the CPU's runqueue. The softirqs are queued in a per-CPU structure that are moved to a different active CPU for them to be processed. Similarly, threads from the runqueue are migrated synchronously to avoid performance degradation for running programs.

Hardware Management Functions. Certain hardware dependent features need to be disabled or enabled during scaling (hotplug) events. For example, machine check has to be disabled before the CPU is put to offline state since any fault in the offline CPU should not affect the remaining system. Similarly when a CPU is started, it's microcode need to be updated and MTRR registers need to be initialized for proper functioning of the system.

Bitmask Updates. Linux maintains a few important global bitmasks of CPU state. These include `cpu_online_mask` and `cpu_active_mask`, which are accessed frequently across the system. These masks should be updated immediately during scaling events for correct functioning of the system.

We consider subscriptions from a few subsystems like workqueue and perf as critical since we are still in the process of adapting those subsystems to Bolt.

3.2 Non-Critical Operations

Bolt defines non-critical operations as those that can either be performed lazily or not performed at all. Bolt makes a best effort to push many of the non-critical operations out of the critical path and thus reduce latency.

Interrupt Migration. Handling of interrupts is a time critical event and it might be surprising to see interrupt handling as a non-critical operation. Interrupts affinitized to a CPU are moved to a different CPU when the CPU is removed. However, from our observation on a Nexus mobile device, all I/O interrupts are always delivered to the base CPU (CPU 0). This was not the case on a desktop machine where the network interrupts were distributed across multiple CPUs.

Bolt currently affinitizes all interrupts to the base CPU to avoid interrupt migration during processor scaling event. However, this will result in performance degradation for servers receiving high number of interrupts. On such systems, the high-traffic interrupts (e.g., network or SSD) can by distributed across multiple CPUs through the `irq_migration` daemon, while lightly loaded interrupts can be handled by the base CPU to avoid migration during scaling. But Bolt does not support this optimization and implementing it is part of the future work.

Memory. Many kernel subsystems, upon receiving notifications of scaling events, free or allocate memory structures such as per-CPU structures or buffer queues. Bolt elides these operations and instead saves memory to be re-used if and when the CPU comes back online. Most kernel subsystems support a master thread that is invoked during memory pressure to release all per-CPU structures. Bolt uses this master thread to avoid any memory leak if a CPU stays offline for an extended time period.

Sysfs. Volatile filesystems like `sysfs` and `procfs` expose kernel settings and metrics through virtual file system interface in Linux. Processor-core based file or directory nodes are removed or created during processor set scaling. However, Bolt does not perform these operations but instead it prevents access to CPU dependent files by verifying if the CPU is online during file open.

Thread Operations. Earlier versions of Linux destroyed and spawned system threads during hotplug. More recent versions of Linux use thread parking [9], which suspends threads indefinitely. System threads like watchdog threads are moved to a parked state during processor scaling. Parking the thread involves waking the thread, even if it is in sleep state, invoking a registered function pointer to perform cleanup, and then moving the thread to the sleep state synchronously. Bolt instead employs an asynchronous approach and it does *not* wait for the thread to enter sleep state after sending the parking message. The benefits are that it improves latency and parking could be avoided if the CPU returns back online before the thread wakes up and sees the parking message.

3.3 Bulk Interface

Bolt adds a new bulk interface support to allow scaling multiple CPUs simultaneously; existing APIs only support adding or removing a single CPU. The API takes a cpumask argument, indicating which CPUs to add or remove rather than the index of an individual CPU. To accomplish a bulk offline with native hotplug, each CPU is moved to offline state sequentially and with no overlap; the online case is similar. Bolt leverages the fact that certain operations can be done once even if multiple CPUs change state. Bolt makes two optimizations. First, updates to global structures are made as a single operation. For example, Bolt reorganizes the scheduling domain related structures once for all CPUs. Second, Bolt performs some operations in parallel. For example, during CPU online it clears caches and register sets concurrently on all CPUs.

4 Evaluation

Our evaluation focuses on the performance of Bolt, but we speculate on the potential energy benefits as well.

Experimental Platform. We performed our experiments on two different processor architectures. First, an x86 based machine with an Intel i5-2500K (Sandy bridge) processor running Linux kernel 3.17.1. Second, an Odroid development board [13] with Exynos 5410 processor running Android 4.4 with Linux kernel 3.4. The Exynos is a big.LITTLE architecture provisioned with A15 and A7 4-core clusters. We disable Turbo Boost in the x86 machine to avoid any performance variability. All experiments were performed in an idle system without any active workloads. For all the experiments, we ran the processors at highest frequency: x86 at 3.3 GHz and A15 at 1.6 GHz.

End-End Latency. The CPU state (online/offline) is accessed through the sysfs file `/sys/devices/system/cpu/cpu*/online`— writing '0' initiates the offline process and '1' brings the cpu to an online state. We measure latency as the time taken from the write to when it returns, at which point the CPU becomes invisible to the OS or it is actively available to the OS.

Figure 1 show the end-to-end latency for an offline operation. The legend represents the individual components of the hotplug operation. (a) Down prepare, dead and post dead are different notifications sent to the kernel subsystems. The down prepare is costly due to a synchronous thread creation by workqueue subsystem in the critical path. Bolt avoids this behavior by reusing threads. (b) Park refers to the thread parking operation that is classified by Bolt as non-critical and handled appropriately. (c) Reduction in the latency of `take_cpu_down` is achieved by avoiding interrupt migration and thus, saving 0.7ms. The remaining overhead is caused by `stop_machine` interface, which is not optimized by Bolt. (d) RCU denotes the protection of `cpu_active_mask` bitmask through RCU synchronization and this is costly since read-copy-update (RCU) has to wait till all CPUs undergo a context switch. Bolt replaces RCU-based synchronization with regular locks. The impact of this change is limited to very few interfaces as can be seen in this commit log [27].

Figure 2 show the latency breakdown for an online operation. Interestingly, the major source of latency in Exynos is software, and in x86, hardware. In the Exynos system, thread creation causes overhead similar to the offline case and Bolt avoids this by parking threads and re-using them during the online operation. However, the x86 incurs substantial delay when the init IPI (to start the core) is sent. Intel documentation [15] specifies that OS should wait for a period of 10ms after the init message is sent to perform hardware initialization. The speedup of Bolt over the native system is thus limited to 1.3x.

Software Entry/Exit Latency. The entry latency refers to the time taken for the CPU core to be switched off during offline operation. This gives an idea on how soon the energy savings begin. The native system takes around 12.5ms (x86) and 6.7ms (Exynos) whereas Bolt takes 0.45ms (x86) with a speedup of 27.8x and 0.38ms Exynos) with a speedup over native being 17.6x. The exit latency is the time taken for the core to schedule the first thread after it is woken up. This gives an idea of the interactivity of the system. The native system takes around 13.8ms (x86) and 12ms (Exynos) whereas Bolt takes 10.27ms (x86) and 0.22ms (Exynos).

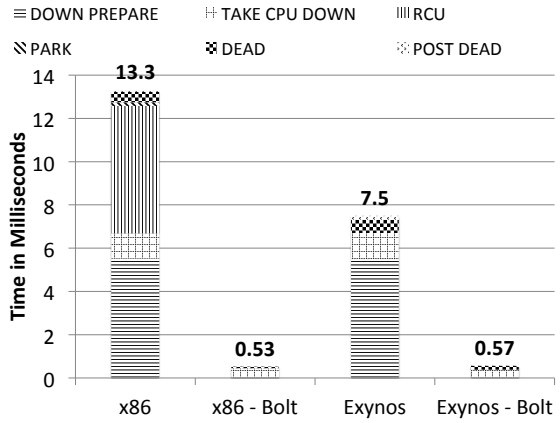


Figure 1: Native hotplug vs. Bolt during offline.

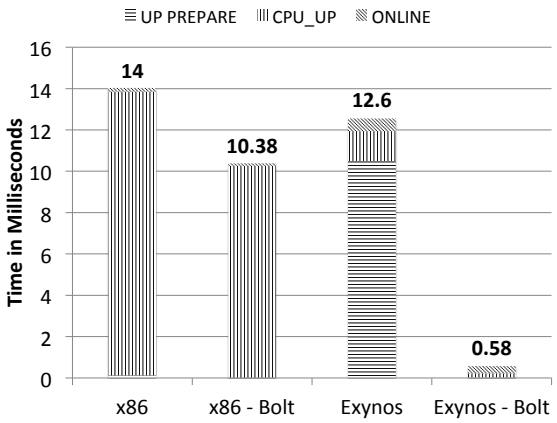


Figure 2: Native hotplug vs. Bolt during online.

Energy Savings. Bolt does not offer a new power management policy but it relies on the processor sleep states for any energy savings. In the Exynos 5410, the offline state and deep sleep state—C2— consume almost same power: 97mW when all cores are in sleep state. However, the processor cluster is allowed to enter C3 (package-level deep sleep state), it consumes 55mW, which is a 43% reduction in power compared to C2. The constraint is that all on-chip cores but one should be in hotplug offline state to enable C3 state.

The default power management policy used for Exynos employs a conservative approach that ensures a minimum of two CPU cores is online for better interactivity. We believe that Bolt could help in implementing a more aggressive policy for more energy savings while preserving interactivity by retaining only one online core and entering C3 in the remainder. On the other hand, hotplug offline on an Intel i5-2500K puts the CPU core to deep sleep state (C6). In this case, hotplug does not result in additional power savings than Linux's cpuidle subsystem [4] but hotplug can still be beneficial by avoiding interrupt handling and scheduling threads on idle cores and extending their idle period.

Bulk Interface. The current bulk interface implementation is available only for x86 architecture and not for Exynos. We specify an input cpumask marked with three CPUs. On the native system, we simulate bulk operations by performing scaling operations sequentially. The native system took 39.2ms and 43.1ms for offline and online operations. Bolt using the same simulation technique took 1.6ms and 31.2ms, while the bulk interface in Bolt took 0.58ms and 10.84ms. The new interface executes the `take_cpu_down` concurrently and avoids multiple time re-organization of schedule domain structures during offline. Overlapping hardware initialization and processing notification messages while waiting for the hardware initialization speed the online operation.

Speculated Hardware. To appreciate the benefits provided by Bolt during an online operation in x86, we emulated a hardware that can perform instant initialization without incurring the 10ms hardware delay. We achieve this by prematurely sending init IPI and removing hardware initialization from critical path. Though this moves the core out of sleep state to running state at increased power, this is acceptable since we are trying to model only the performance without the hardware overhead. Bolt finishes online operation in 0.38ms compared to 4ms for native providing a speedup of 10.5x, and the online bulk interface takes 0.71ms compared to 13.1ms providing a speedup of 18x. The native latency values – 4ms and 13.1ms – are achieved by removing the hardware initialization delay of 10ms from original latency values. These numbers show that hotplug latency will be dominated by software overhead in future processors and Bolt makes significant reduction in the software overhead.

5 Conclusion

Processor set scaling is important for energy efficiency, throughput improvement, cost savings and VM scaling. However, the current hotplug mechanism is slow and cannot support frequent changes. We propose Bolt, which classifies hotplug operations as critical and non-critical. This separation helps Bolt achieve low latency by removing non-critical operations from the critical path. Bolt also supports a bulk interface that allows scaling at granularity of multiple cores. The low latency mechanism and the new interface support through Bolt enable future systems to scale at much finer time-scales.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grant CNS-1302260. We would like to thank our shepherd, Dan Tsafrir, and the anonymous reviewers for their invaluable feedback. We would also like to thank Venkat, Sanketh and Thanu for their comments on the earlier draft of the paper. Swift has a significant financial interest in Microsoft, and Nam Sung Kim has financial interests in Samsung Electronics and AMD.

References

[1] ARM big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[2] AMD - SeaMicro, Inc. Seamicro sm15000 fabric compute systems. http://www.seamicro.com/sites/default/files/SM_DS06_v2.1.pdf, 2012.

[3] H. Chung, M. Kang, and H.-D. Cho. Heterogeneous multi-processing solution of exynos 5 octa with arm® big.little technology. http://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf.

[4] J. Corbet. The cpuidle subsystem. http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/cpuquiet.pd://lwn.net/Articles/384146/, April 2010.

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. ISCA*, June 2011.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *technology*, 28:32, 2014.

[7] Gartner. Case studies in cloud computing. https://www.gartner.com/doc/1761616/case-studies-cloud-computing.

[8] H. R. Ghasemi and N. S. Kim. Rcs: Runtime resource and core scaling for power-constrained multi-core processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 251–262, 2014.

[9] T. Gleixner. Kthread: Implement park/unpark facility. https://lwn.net/Articles/500338/, 2012.

[10] T. Gleixner, P. E. McKenney, and V. Guittot. Cleaning up linux's cpu hotplug for real time and energy management. *SIGBED Rev.*, pages 49–52, Nov. 2012.

[11] A. Gupta, E. Ababneh, R. Han, and E. Keller. Towards elastic operating systems. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, 2013.

[12] C. Hameed. Windows 7 and windows server 2008 r2: Core parking, intelligent timer tick and timer coalescing. http://blogs.technet.com/b/askperf/archive/2009/10/03/windows-7-windows-server-2008-r2-core-parking-intelligent-timer-tick-timer-coalescing.aspx, 2009.

[13] HardKernel co., Ltd. Odroid-xu+e. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079, 2013.

[14] Hewlett-Packard Development Company. Hp moonshot system. http://www8.hp.com/us/en/products/servers/moonshot/index.html.

[15] Intel Corporation. Intel multiprocessor specification, v1.4. http://www.intel.com/design/pentium/datashts/24201606.pdf, 1997.

[16] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accomodating software diversity in chip multiprocessors. In *Proc. of the 34th ISCA*, June 2007.

[17] J. Jann. Dynamic logical partitioning for power systems. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 587–592. Springer US, 2011.

[18] B. Klug. Samsung announces exynos 5 octa soc. http://www.anandtech.com/show/6602/samsung-announces-exynos-5-octa-soc-4-cortex-a7s-4-cortex-a15s, 2013.

[19] M. Nishikiori. Server virtualization with vmware vsphere 4. *Fujitsu Scientific and Technical Journal*, pages 356–361, 2011.

[20] NVIDIA. Nvidia tegra x1 nvidia's new mobile superchip. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, 2015.

[21] S. Panneerselvam and M. M. Swift. Chameleon: operating system support for dynamic processors. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 99–110, 2012.

[22] Schrijver, Peter De and Miettinen, Antti P. Cpuquiet: A framework to manage cpus. http://www.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/cpuquiet.pdf.

[23] A. L. Shimpi. Intel's haswell architecture analyzed: Building a new pc and a new intel. http://www.anandtech.com/show/6355/intels-haswell-architecture/3, 2012.

[24] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, and V. Pallipadi. Energy-aware task and interrupt management in linux. In *Ottawa Linux Symposium*, 2008.

[25] A. V. D. Ven. Absolute power. https://software.intel.com/sites/default/files/absolute_power.pdf.

[26] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *ACM SIGOPS Operating Systems Review*, 43(2), Apr. 2009.

[27] P. Zijlstra. Sched: Remove get_online_cpus usage. https://github.com/torvalds/linux/commit/6acce3ef84520537f8a09a12c9ddbe814a584dd2, 2013.

Boosting GPU Virtualization Performance with Hybrid Shadow Page Tables

Yaozu Dong[1, 2], Mochi Xue[1,2], Xiao Zheng[2], Jiajun Wang[1,2], Zhengwei Qi[1], Haibing Guan[1]
{*eddie.dong, xiao.zheng*}*@intel.com* {*xuemochi, jiajunwang, qizhenwei, hbguan*}*@sjtu.edu.cn*
[1]*Shanghai Jiao Tong University,* [2]*Intel Corporation*

Abstract

The increasing adoption of Graphic Process Unit (GPU) to computation-intensive workloads has stimulated a new computing paradigm called GPU cloud (e.g., Amazon's GPU Cloud), which necessitates the sharing of GPU resources to multiple tenants in a cloud. However, state-of-the-art GPU virtualization techniques such as gVirt still suffer from non-trivial performance overhead for graphics memory-intensive workloads involving frequent page table updates.

To understand such overhead, this paper first presents GMedia, a media benchmark, and uses it to analyze the causes of such overhead. Our analysis shows that frequent updates to guest VM's page tables causes excessive updates to the shadow page table in the hypervisor, due to the need to guarantee the consistency between guest page table and shadow page table. To this end, this paper proposes gHyvi[1], an optimized GPU virtualization scheme based on gVirt, which uses adaptive hybrid page table shadowing that combines strict and relaxed page table schemes. By significantly reducing trap-and-emulation due to page table updates, gHyvi significantly improves gVirt's performance for memory-intensive GPU workloads. Evaluation using GMedia shows that gHyvi can achieve up to 13x performance improvement compared to gVirt, and up to 85% native performance for multi-thread media transcoding.

1 Introduction

The emergence of HPC cloud [30] has shifted many computation-intensive workloads such as machine learning [24], molecular dynamics simulations [31] and media transcoding to cloud environments. This necessitates the use of GPU to boost the performance of such computation-hungry applications, resulting in a new computing paradigm called GPU cloud (such as Amazon's GPU cloud [2]). Hence, it is now vitally important to provide efficient GPU virtualization to provision elastic GPU resources to multiple users.

To address this challenge, two recent full GPU virtualization techniques, gVirt [29] and GPUvm [28], are proposed respectively. gVirt is the first open-source product-level full GPU virtualization approach based on Xen hypervisor [11] for Intel GPUs, while GPUvm provides a Graphic Process Unit (GPU) virtualization approach on the NVIDIA card. This paper mainly focuses on gVirt due to its open-source availability. Specifically, gVirt presents a vGPU instance to each VM to run native graphics driver, which achieves high performance and good scalability for GPU-intensive workloads.

While gVirt has made an important first step to provide full GPU virtualization, our measurement shows that it still incurs non-trivial overhead for media transcoding workloads. Specifically, we build GMedia using Intel's MSDK (Media Software Development Kit) to characterize the performance of gVirt. Our analysis uncovers that gVirt still suffers from non-trivial performance slowdown due to an issue called *Massive Update Issue*. This is caused by frequent updates on guest page tables, which lead to excessive VM-exits to the hypervisor to synchronize the shadow page table with the guest page table.

To address the Massive Update Issue, this paper introduces gHyvi, which provides a hybrid page table shadowing scheme to provide optimized full GPU virtualization based on Xen hypervisor for Intel GPUs. Inspired by the GPU programming model, we introduce a new asynchronous mechanism, namely relaxed page table shadowing, which removes trap-and-emulation and thus reduces the overhead of massive page table's modifications. To minimize the overhead of making guest and shadow page tables consistent, we combine the two mechanisms into a adaptive hybrid page table shadowing scheme, which take advantage of both the traditional strict and the new relaxed page table shadowing. When

[1]The source code of gHyvi will be available at `https://01.org/igvt-g`.

there are infrequent page table accesses, gHyvi works in strict page table shadowing; once the gHyvi detects the guest VM is frequently updating the page table, it will switch to the relaxed page table shadowing.

One critical issue of using the relaxed page table shadowing scheme is to reconstruct the shadow pages when shadow pages are inconsistent with guest pages. To better understand the tradeoff of different reconstruction policies, we implement and evaluate four page table reconstruction policies: full reconstruction, static partial reconstruction, dynamic partial reconstruction and dynamic segmented partial reconstruction. Our analysis shows that the last one usually has better performance than the others, which is thus used as the default policy for gHyvi.

We have implemented gHyvi based on gVirt, which comprises 600 LoCs. Experiments using GMedia on an Intel GPU card show that gHyvi can achieve up to 13x performance improvement compared to gVirt, and up to 85% native performance for multi-thread media transcoding. Our analysis shows that gHyvi wins due to the reduction of up to 69% VM-exits.

In summary, this paper makes the following contributions:

- A GPU-enabled benchmark for media transcoding performance (GMedia), by invoking functions from Intel MSDK to evaluate and collect the performance data on Intel's GPU platforms.
- A relaxed page table shadowing mechanism as well as a hybrid shadow page table scheme, which combines the strict page table shadowing with the relaxed page table shadowing.
- Four reconstruction policies: the full reconstruction policy, static partial reconstruction policy, dynamic partial reconstruction policy, and the dynamic segmented partial reconstruction policy for relaxed page table shadowing mechanism.
- An evaluation showing that gHyvi achieves up to 85% native performance for multi-thread media transcoding and a 13x speedup over gVirt.

The rest of the paper is organized as follows: Section 2 describes some background information on gVirt and GPU programming model. Section 3 presents our benchmark for media transcoding and discusses the Massive Update Issue in detail, followed by the design and implementation of gHyvi In section 4. Then, section 5 evaluates the gHyvi and section 6 discusses the related work. Finally, section 7 concludes with a brief discussion on future work.

2 Background

2.1 GPU for Computing

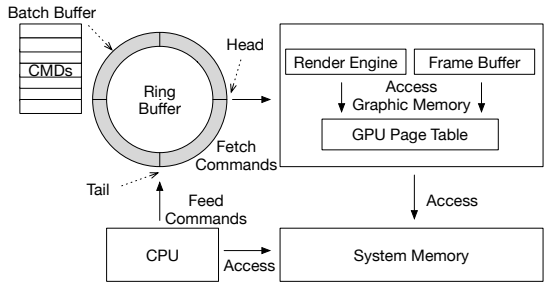


Figure 1: GPU Programming Model

GPU programming model: Figure 1 illustrates the GPU programming model. The graphics driver produces GPU commands into primary buffer and batch buffer, which is driven by the high level programming APIs like OpenGL and DirectX. GPU consumes the commands and fulfills the acceleration work accordingly. The primary buffer is a ring structure (ring buffer), which is designed to deliver the primary commands. Due to the limited space in the ring buffer, the majority (up to 98%) of commands are in the batch buffer chained to the ring buffer.

A register tuple, which includes a head register and a tail register, is implemented in the ring buffer. CPU fills commands from tail to head, and GPU fetches commands from head to tail, all within the ring buffer. The driver notifies GPU the submission and completion of the commands through the tail, while GPU updates the head. Once the CPU completes the placement of commands in the ring buffer and batch buffer, it informs GPU to fetch the commands. In general, GPU will not fetch the commands placed by the CPU in the ring buffer until the CPU updates the tail register [29].

GPU Cloud: Due to the massive computing power, GPU has been expanded from the original graphic computing to general purpose computing. The rising of GPU cloud, which extends today's elastic resource management capability from CPU to GPU, further enables efficient hosting of GPU workload in cloud and datacenter environments. The strong demand of hosting GPU applications calls for GPU clouds that offer full GPU virtualization solutions with good performance, full features and sharing capability.

2.2 GPU Benchmarks

While there are many GPU benchmarks evaluating the performance of GPU cards, they mainly focus on graphics ability of cards [1, 8] either for OpenGL or DirectX commands. Though there are a few benchmarks for general purpose computing (GPGPU) such as Rodinia [12] and Parboil [27], they are not available for Intel's GPU. Besides, existing benchmarks neglect the media processing workloads, which is a key to boost the performance of media applications in cloud.

To this end, this paper presents GMedia, a media transcoding benchmark shown in Figure 4, based on Intel's MSDK (Media Software Development Kit). Intel's MSDK grants media application developers access to hardware acceleration through a unified API. As a result, developers can take advantage of the media acceleration capabilities of future graphics-processing solutions without rewriting the code.

GMedia is a wrapper, which directly invokes the media functions of Intel's MSDK to generate common media transcoding workloads. By modifying the configuration files, we can assign source media file and target media file's settings like resolution, bitrate, FPS, etc. Besides, test cases can be run with assigned threads, which is quite helpful in order to evaluate multi-task performance. After running the benchmark, a report will be provided, which shows the average FPS (frame per second) for each thread and total average FPS. The FPS results intuitively reflect the performance.

3 gVirt and Massive Update Issue

3.1 Intel gVirt

gVirt [29], a product-level full GPU virtualization for Intel Graphics, achieves both good performance and scalability. In full GPU virtualization, a virtual machine monitor (VMM) traps and emulates the guest access to the privilege GPU resources for security and multiplexing, while passing through access to the performance critical resources, such as the access of CPU to graphic memory. For GPU commands, once the CPU submits them, they will be parsed and audited to ensure the safety. Most of the GPU commands will be executed in GPU without VMM intervention, resulting in the nearly native performance being achieved.

gVirt applies virtualization to the GPU page tables. The shared shadow global page table is implemented for all VMs in order to achieve resource partition and address space ballooning. Here, ballooning is the technique gVirt uses to isolate the address spaces of different VMs in shared shadow global page table. The shared shadow global page table is accessible for every VM. However,

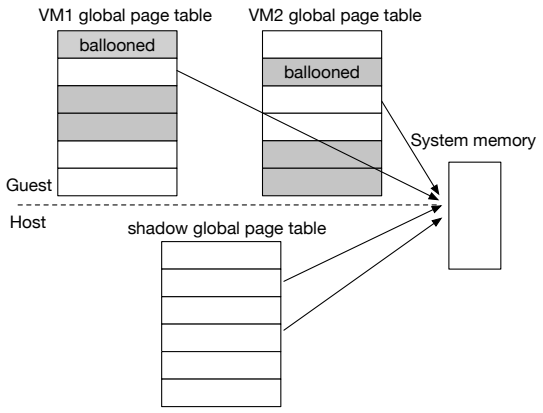


Figure 2: Shared shadow global page table

only part of the shared global page table can be accessed for one VM to guarantee the isolation, and the ballooning technique hides the rest part of shared shadow page table from this VM. As shown in Figure 2, each VM contains its own guest global page table to translate from the graphics memory frame number to the guest memory frame number. The shared shadow global page table maintains the translations from graphics memory frame number to the host memory frame number for all VMs.

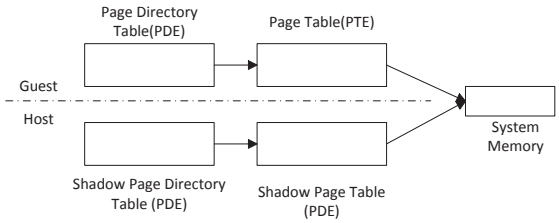


Figure 3: per-VM shadow local page table

Per-VM shadow local page table is implemented to achieve pass-through of local graphics memory access. As shown in Figure 3, the local page tables are with two-level paging structures, the first level being the Page Directory Entries (PDEs), which is located in the global page table. This, in turn, points to the second level Page Table Entries (PTEs), which is in the system memory.

The generic solution for keeping shadow page table consistent with guest page table is to write-protect the shadow page table at all points in time. When a write-protection page fault happens, VMM can potentially trap and emulate updates to the guest page table. In gVirt, shadow page tables are implemented in this *strict page table shadowing*, which is a mechanism that synchronously keeps the page table consistent with the corresponding guest page table all the time.

3.2 Massive Update Issue

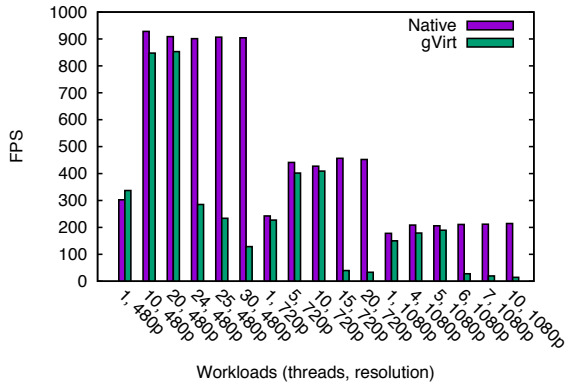


Figure 4: GMedia results of Native and gVirt

While gVirt achieves good performance in many cases, where the guest modifications of page table are infrequent, it suffers from poor performance when dealing with workloads such as media transcoding.

By observing the pattern of guest page table modifications, we find that the guest VM is frequently swapping graphics memory pages, i.e., dropping the previous pages or contents and re-construct the contents later on when needed. Once the guest VM starts to construct the memory pages, it modifies the entries of page table contiguously, until the operation is complete. In turn, this causes a huge amount of page table entry modifications, and the excessive modifications result in busy trap-and-emulate, which eventually leads to low FPS media transcoding with multiple threads. When taking this into account, it is safe to conclude that the strict shadow page table shadowing mechanism is the root cause of the performance issue.

To confirm this, we used GMedia to investigate the media transcoding performance of gVirt under various workloads. Figure 4 shows the results of media transcoding on our test platform (detailed setting in section 5) with multiple threads normalized to one thread. We run 30 cases for each resolution to get a full coverage while selectively presenting the representative cases. For many cases, the performance discrepancy between gVirt and native is not obvious. For the 480p media file transcoding, the native machine works fine in each case with small performance degradation, yet the performance on DomU (the production VM in Xen) degrades very clearly with thread multiplies over 20. For high-resolution media file transcoding, the native machine still works adequately in each case, while DomU's performance degrades with multiple threads, with over 90% in the worst cases.

Transcoding a media file requires a large amount of graphic memory in order to read the file in and process it. Once the memory is limited, Intel's GPU driver [4] [5] allocates a new memory page and modifies the page table entry to point to the new memory page. In gVirt, the write-protection page faults of the shadow page table happen massively when the thread number becomes higher or when the video resolution is high, resulting in the low FPS. Because the guest VM frequently allocates new graphic memory from system memory and massively modifies the page table entries. Therefore, we define this performance overhead problem caused by frequent page table updates as the Massive Update Issue.

3.3 PTE Update Pattern

To further analyze the Massive Update Issue, we profile 6 media transcoding cases from GMedia: 5-thread 720p, 7-thread 720p, 15-thread 720p, 3-thread 1080p, 4-thread 1080p and 10-thread 1080p, to count the VM-exits happen during the workload running. We categorize the VM-exit reasons and find that the EPT-violation dominates in cases with the Massive Update Issue. By breaking down the EPT-violation we find that the guest VM frequently modifies the PTE pages when running issued cases. Furthermore, we analyze the PTE updates to find the pattern of workloads with the Massive Update Issue, which motivates the design of gHyvi.

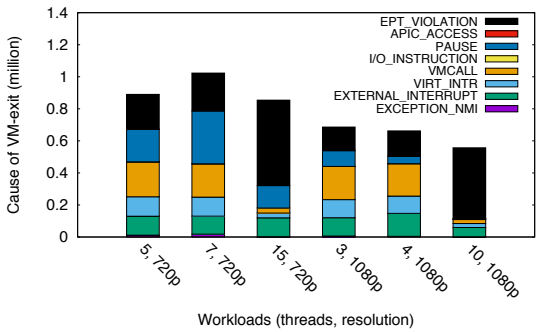


Figure 5: Break-down of VM-exit

Figure 5 shows the break-down of 6 media transcoding cases' VM-exits in the duration of 10s. Among these 6 cases, 15-thread 720p and 10-thread 1080p transcoding have much higher rates of Extended Page Tables violation (EPT-violation), which is caused by a page fault in the extended page table. As shown in Table 1, the percentages of EPT-violation are usually under 25% in other cases but dramatically increase to 62.40% in the case of 15-thread 720p and 79.45% in the case of 10-thread 1080p.

Threads	Resolution	EPT-violation Percentage
5	720p	24.43%
7	720p	23.06%
15	720p	62.40%
3	1080p	21.43%
4	1080p	23.82%
10	1080p	79.45%

Table 1: EPT-violation percentage in the 6 cases

Interestingly, when a VM guest graphics driver accesses CPU pages to prepare PTE pages for GPU, it triggers EPT-violation as well. We further provide a breakdown of the EPT-violations. PTE updates trigger 82.97% and 78.82% of VM-exit caused by EPT-violation for the cases of 15-thread 720p transcoding and 10-thread 1080p transcoding accordingly. The PTE page updates excessively expand the percentage of VM-exits caused by EPT-violation.

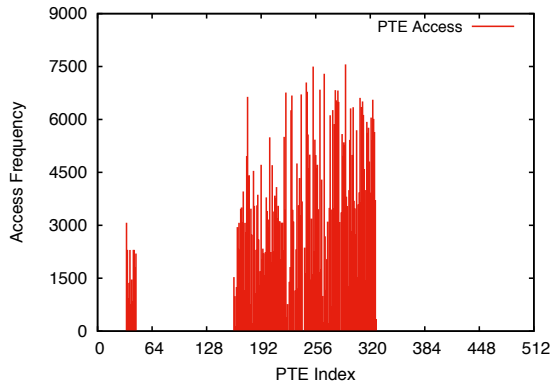


Figure 6: PTE update frequency

Furthermore, Figure 6 demonstrates the update frequency on 512 pages within 10s for 15-thread 720p transcoding case. The pages whose index lie between 150 and 320 are massively modified, and the frequency can be up to 7.5k times. Each PTE updates trigger the VM-exit, then the VMM traps and emulates the corresponding writes. However, there are some pages that are never accessed, like the pages whose index is between 320 and 512. This pattern encourages us to implement the partial reconstruction policies aside from reconstructing the whole page table, because part of the page table may stay unchanged.

We also collected the timestamp and page index to each PTE update to see the overall pattern. Figure 7 demonstrates all 627k PTE updates occurring within the 10s of 15-thread 720p transcoding case. This pattern is

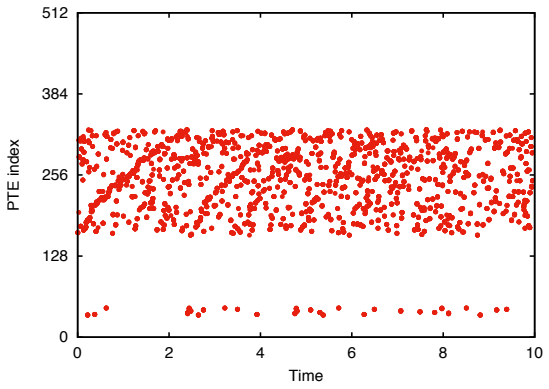


Figure 7: PTE update pattern (in 10s)

in correspondence with Figure 6. Updates on the same page repeat throughout the entire progress.

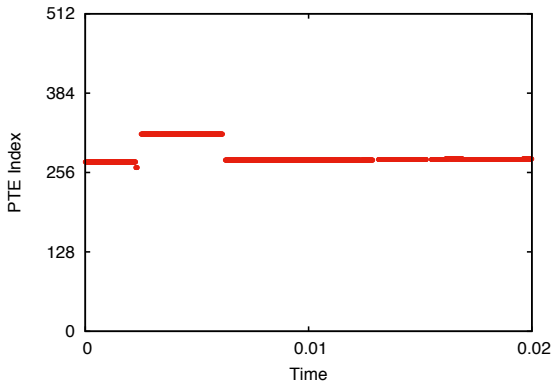


Figure 8: PTE update pattern (in 0.02s)

A small part is split from the 10s to see the detailed pattern of this case's PTE updates. Figure 8 demonstrates the PTE page update pattern in 0.2s, within the same case. The updates on one PTE page are continuous, i.e., once a PTE page is modified, there will be following updates on the same page. This pattern inspires us to remove the write-protection of PTE page once the page is modified for the first time.

4 Design and Implementation

To address the Massive Update Issue for media transcoding workload, this paper describes, gHyvi, a hybrid page table shadowing scheme for gVirt, as shown in Figure 9. gHyvi introduces a new page table shadowing mechanism for shadow page tables in gVirt, namely relaxed page table shadowing, which relaxes the constraints of write-protection to the guest page table. gHyvi switches between two different page table shadowing mechanisms, based on the pattern of GPU's current workload.

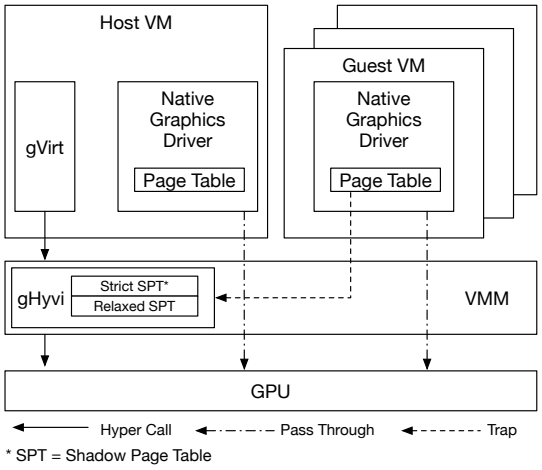


Figure 9: High level architecture of gHyvi

By combining traditional strict page table shadowing and relaxed page table shadowing mechanism, gHyvi takes advantage of both. For workloads with the Massive Update Issue like multi-thread media transcoding, gHyvi could efficiently improve the gVirt's performance.

4.1 Workflow of gHyvi

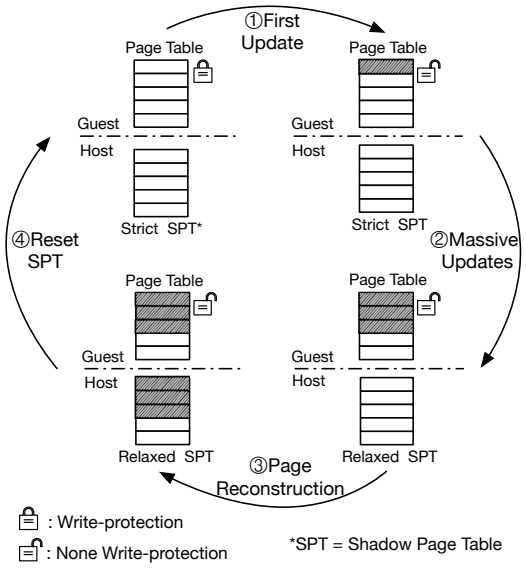


Figure 10: Workflow of gHyvi

Figure 10 illustrates the basic workflow of gHyvi:

(1) gHyvi initiates the shadow page table which is consistent with the guest page table, and it makes all the page table write-protected.

(2) If a page table entry is modified by the guest, it triggers page fault which will be trapped into gHyvi. gHyvi takes a snapshot of this page and removes the write-protection of this page. The corresponding page table entry of the shadow page table will be switched into the relaxed shadowing mechanism. Afterwards, the modifications on the guest page will not be updated to the shadow page table immediately.

(3) When the guest VM is scheduled in, the shadow page table has been already inconsistent with the guest page table. gHyvi will re-construct the shadow page table according to the previous snapshot to promote coherence with the guest page table again, so that it could guarantee the hardware engines use the correct translations.

(4) After the reconstruction of the shadow page table, gHyvi sets the page table entries in the relaxed page table shadowing back to the strict page table shadowing. Then, this workflow circle would be repeated again.

4.2 Relaxed Page Table Shadowing

From GPU's programming model, we observe that the guest VM's modifications of page table entries will not take effect until the GPU commands are submitted to physical engine by VMM. Inspired by this, we implement a new page table shadowing mechanism for page table called relaxed page table shadowing. This mechanism is applied to the guest VM's shadow page table when gHyvi detects that the guest VM modifies the page table entries massively, i.e., the trap-and-emulation of the guest page table frequently happens. In contrast to strict page table shadowing, the relaxed page table shadowing removes the write-protection of page tables to avoid the cost from trapping and emulating the modifications of page table.

For gHyvi, the relaxed page table shadowing will reduce the overhead of trapping and emulating due to continuous and massive modifications on the guest page table. After the shadow page table has been switched to the relaxed page table shadowing mechanism, modifications within the guest page table will not be updated to shadow page table temporarily. The latency is acceptable because of the GPU programming model in which GPU may fetch the commands and cache the page table translations internally at the time of command submission. At the time the commands are submitted to the physical engine, the shadow page table would be consistent with guest page table again to ensure correct translations by reconstructing the page table.

4.3 Hybrid Page Table Shadowing

As we discussed before, for many workloads there are infrequent modifications to the guest page table, where the strict page table shadowing mechanism fits well in this situation. In such cases, relaxed page table shadowing is not suitable, because reconstructing a page takes a longer period than trapping and emulating modifications on that page. To make gHyvi enjoy good performance for both cases and minimize the cost of updating shadow page table, we combine the two mechanisms into one hybrid page table shadowing, where gHyvi's shadow page tables adaptively switch between the strict shadowing and the relaxed shadowing mechanisms, based on the current workload's access pattern.

Since infrequent page table access pattern is ubiquitous, gHyvi will keep guest page table mostly working with the strict shadowing mechanism. Once the gHyvi detects the guest VM is frequently modifying the page table, it will automatically switch the guest page table into a relaxed mechanism. When the guest VM no longer frequently modifies page table, gHyvi may switch guest page table back to the strict shadowing mechanism. gHyvi can also selectively apply the relaxed shadowing mechanism to certain portions of the page table, instead of the whole page table.

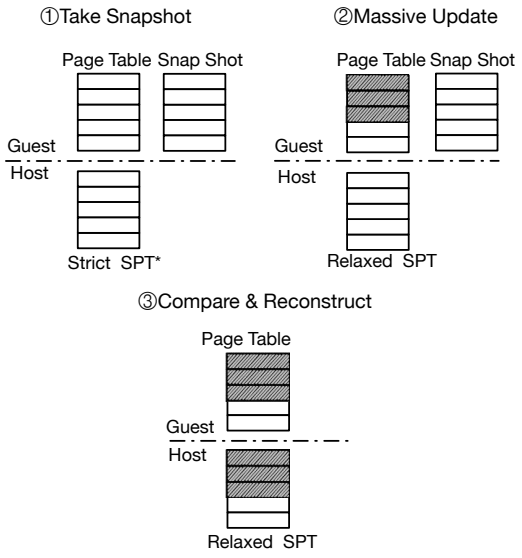


Figure 11: Page reconstruction with snapshot

4.4 Page Reconstruction

Page reconstruction is necessary when the shadow pages are not consistent with the guest pages. There are 1024 page entries in one page, and in order to reconstruct the shadow page, generally we need to re-write all the entries and make sure each entry is consistent with the corresponding entry of the guest page. However, when part of a page is modified, we do not necessarily need to rewrite all its entries when we reconstruct it, because rewriting the unmodified part of the page is costly. Hence, we introduce *snapshot* to accelerate the page reconstruction.

As shown in Figure 11, when a shadow page is consistent with the guest page after the reconstruction or initiation, we take a snapshot of the guest page and store it. When reconstructing a page, we will compare the current page with the snapshot and get the different entries. The different section is the modified part of the page. Hence, we just need to reconstruct this part to make the shadow page consistent with the guest page table. Although the cost of reconstructing a page is expensive, it is worthwhile compared to the efforts needed to trap and emulate the modification multiple times.

4.5 Reconstruction Policies

We implement four reconstruction policies for gHyvi and evaluate them to choose a final policy which delivers the best performance. When gHyvi switches a page into the relaxed shadowing mechanism, the write-protection of this page is removed. Moreover, *relaxed page table shadowing* is an asynchronous mechanism which allows the shadow page table to be inconsistent when it is not needed for delivering translations. Hence, the following modifications on it will not be updated to the shadow page immediately. Before the commands are submitted to the physical engine, gHyvi will reconstruct the page's corresponding shadow page to ensure the correct translation. The profiling of cases with Massive Update Issue in section 3.3 demonstrates that when the workload is accessing the page table massively, only certain pages are being accessed repeatedly, and the majority of the guest page table still remains untouched. Hence, it is essential for gHyvi to switch certain pages into relaxed shadowing mechanism and reconstruct them when necessary.

The full reconstruction policy is to switch all pages into the relaxed shadowing mechanism, and reconstruct them all before the commands are submitted to the physical engine. When a VM is created, it allocates 512 pages in total, and we will remove the write-protection of all 512 pages. After that, there will no longer be any trapping and emulating to update the shadow pages, and all the shadow pages will be reconstructed to guarantee that physical engine gets the correct translations.

The static partial reconstruction policy selects a certain amount of pages to apply with relaxed shadowing. It reconstructs the selected pages each time to make them consistent with their corresponding guest pages while the unselected pages still remain in the strict shadowing. Ac-

cording to the profiling of cases with the Massive Update Issue in section 3.3, there are some pages being accessed much more frequently than other pages, which are referred to as hot pages. These hot pages are specifically selected to utilize the relaxed shadowing mechanism based on the observed access pattern.

The dynamic partial reconstruction policy is utilized to apply the relaxed shadowing mechanism to pages dynamically, based on the access pattern of workload. At the time VM is created, all the pages are applied with strict shadowing and gHyvi maintains a list to record pages that are run with the relaxed shadowing. When a page is modified for the first time, a page fault occurs. gHyvi will add this page to the list and switch it into the relaxed shadowing mechanism. The new pages will then be continuously added to the list while the workload is running. Eventually the pages in the list will cover all the modified pages.

The dynamic segmented partial reconstruction policy is an optimization for the dynamic partial reconstruction policy. Like the dynamic partial reconstruction policy, gHyvi puts modified pages in the dirty list, and every time when the commands submitted to the physical engine, the shadow page table will be consistent with guest page table again, by reconstruction. However, in this optimized policy, gHyvi will reset the dirty list, and switch the pages in the list back to the strict shadowing mechanism after the reconstruction.

Currently, gHyvi uses the dynamic segmented partial reconstruction policy as default, according to the performance evaluation in section 5.2.

5 Evaluation

This section presents a set of evaluations to compare the performance of gHyvi with the original gVirt. We run media transcoding and 2D/3D workloads in Linux, along with 2D/3D workloads in Windows. We first compare the four reconstruction policies in gHyvi, which confirms that dynamic segmented partial reconstruction policy is with the best performance. Then, we use this policy to compare gHyvi with the original gVirt as well as native performance. In summary, our results show that gHyvi achieves 85% of native performance in most media transcoding test cases on Linux. For Linux 3D workloads, gHyvi has no negative effect in LightsMark, OpenArena, and UrbanTerror, respectively. For Linux 2D workloads, gHyvi shows no negative effect in firefox-asteroids, firefox-scrolling, midori-zoomed, and gnome-system-monitor, respectively. For windows 2D/3D workloads, gHyvi has no negative effect on performance in 3Dmark06 [1], Heaven3D [3], and PassMark2D [8] respectively.

5.1 Configuration

Our test platform deploys a 4th generation Intel Core processor i5 4570 with 4 CPU cores (3.2Ghz), Intel Z87 chipset, 8GB system memory and a 250GB Seagate HDD disk. The Intel Processor Graphics integrated in the CPU supports a 2GB global graphics memory space and multiple 2GB local graphics memory spaces. We run 64-bit Ubuntu 14.04 with a 3.14.1 kernel in both Dom0 and Linux guest, and 64-bit Windows 7 in Windows guest, on Xen 4.3. Both Linux and Windows run a native graphics driver. Each VM is allocated with 2 vCPUs, 2GB system memory and 672MB global graphics memory.

We evaluate the performance on native, gVirt, and gHyvi respectively. For evaluations on Linux, our customized media performance benchmark was used for media performance. The Phoronix Test Suite 3D benchmark including LighsMark, OpenArena, UrbanTerror are used for 3D performance. Additionally, Cario-perftrace 2D benchmark including firefox-asteriods (firefox-ast), firefox-scrolling (firefox-scr), midori-zoomed (midori), and gnome-system-monitor (gnome) is used for 2D performance. For evaluations on Windows, we run 3DMark06, Heaven3D and PassMark2D workloads. All the benchmarks are run under 1920*1080 resolution. We will compare the performance of VM under gHyvi, gVirt, and the native system.

5.2 Reconstruction Policy

In this section, we evaluate four reconstruction policies designed for gHyvi, full reconstruction, static partial reconstruction with four different settings (50, 100, 200, 300), dynamic partial reconstruction, and dynamic segmented reconstruction. The dynamic segmented reconstruction achieves the best performance, up to 13x of gVirt and 85% of native.

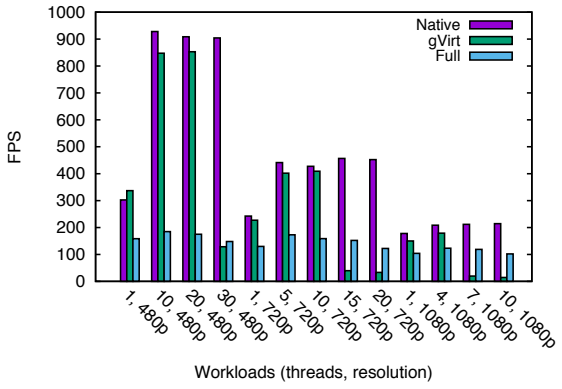


Figure 12: gHyvi with full reconstruction policy

Figure 12 presents the performance of gHyvi with the full reconstruction policy, and all multiple threads are

normalized into a single thread. Throughout all cases, the FPS of full reconstruction policy is between 100 and 200. gHyvi shows a worse performance than gVirt in cases without the Massive Update Issue, while achieving a better performance when the issue occurs. As we discussed in section 4.5, all 512 pages are applied with the relaxed mechanism, so full reconstruction brings more overhead on reconstructing non-accessed pages, which is the reason for cases with little page update showing poor performance.

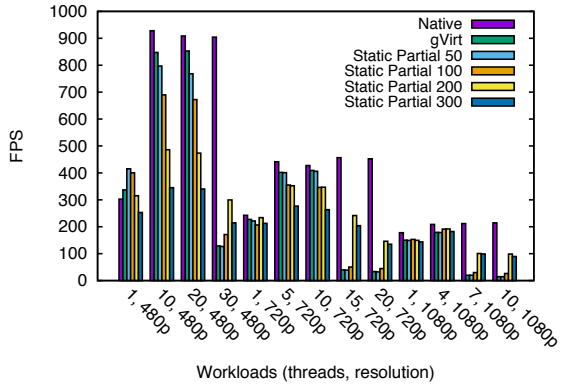


Figure 13: gHyvi with static reconstruction policy

We selectively switch 50, 100, 200, and 300 pages into the relaxed mechanism to evaluate the static partial reconstruction policy. As shown in Figure 13, for cases without the issue static partial reconstruction policy achieves a worse performance than gVirt. The more pages that are switched into the relaxed mechanism, the worse the performance static partial reconstruction becomes. For pages with few page table updates, reconstruction is meaningless. For cases with the Massive Update Issue, the static partial reconstruction policy works and achieves a superior performance than gVirt. Policy with 200 pages setting achieves the best performance for cases with the Massive Update Issue, because policies with less pages cannot cover all the frequently accessed pages, and policies with more pages include some useless pages.

Figure 14 confirms that the dynamic segmented partial reconstruction achieves better performance than dynamic partial reconstruction comprehensively. gHyvi performs better than gVirt in issued cases, and has similar performance in normal cases. The dynamic partial reconstruction switches the PTE pages into the relaxed mechanism progressively. However, some pages switched into the relaxed mechanism may never be accessed again, and reconstructing these pages will produce extra overhead. Dynamic segmented partial reconstruction resets the relaxed pages, after setting them to the guest pages. So for each cycle, dynamic segmented policy only reconstructs

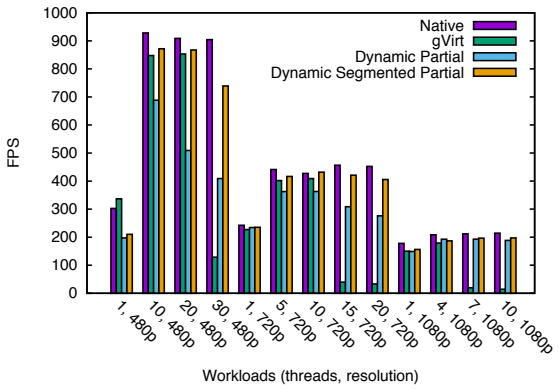


Figure 14: gHyvi with dynamic partial reconstruction and dynamic segmented partial reconstruction

pages that need to be reconstructed. Overall, dynamic segmented partial reconstruction is the most efficient policy, which is finally adopted by gHyvi.

5.3 2D and 3D performance

In this section, we evaluate the 2D and 3D performance of gHyvi under Linux and Windows. The results show that gHyvi has comparable performance with gVirt's 2D and 3D performance. Moreover, gHyvi achieves slightly superior performance than gVirt in some cases.

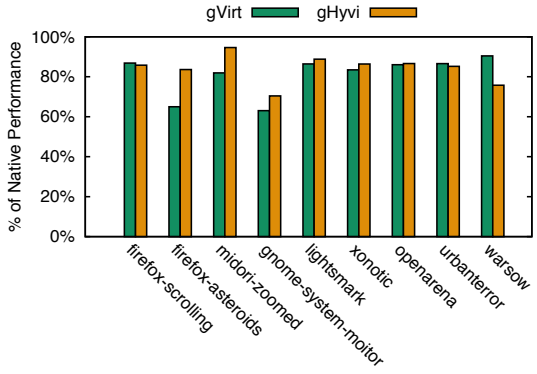


Figure 15: Performance running Linux 2D/3D workloads

Figure 15 demonstrates that gHyvi achieves up to 94.63% of native performance in 2D workloads and 88.81% in 3D workloads on Linux. Figure 16 demonstrates that gHyvi achieves up to 88.81% on Windows.

With the exception of the firefox-scrolling, urbanterror, warsow, SM2.0 and Pass2D, gHyvi outperforms gVirt. However, the performance discrepancy between gHyvi and gVirt are acceptable.

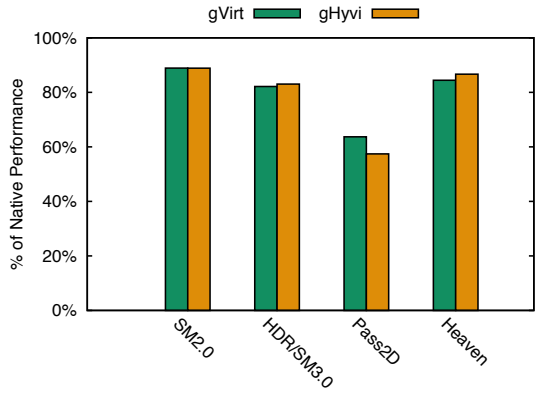


Figure 16: Performance running Windows 2D/3D workloads

6 Related Work

6.1 GPU Benchmarks

Since GPUs are used for acceleration of general purpose computing, some benchmarks have been implemented for evaluating their performance. Rodinia [12] is a benchmark suite for heterogeneous computing. It aids architects in the study of emerging platforms such as GPUs. Rodinia includes applications and kernels that target multi-core CPU and GPU platforms. And Parboil [27] is a set of throughput computing applications useful for studying the performance of throughput computing architecture and compilers. It collects benchmarks from throughput computing application researchers in many different scientific and commercial fields including image processing, bio-molecular simulation, fluid dynamics, and astronomy.

Unfortunately, the benchmarks above are not available for Intel's GPU now. Meanwhile, GPU's media performance has become a big concern for service providers. However, there is no benchmark specifically for this kind of workload. So, this paper proposes GMedia, a media transcoding benchmark based on Intel's MSDK.

6.2 GPU Virtualization

Though virtualization has been studied extensively in recent years, GPU virtualization is still a nascent area of research. Typically, there are four ways to use GPU in a Virtual Machine (VM): I/O pass-through, device emulation, API remoting, and mediated pass-through.

A naive way to use GPU in virtualized environment would be to directly pass through the device to a specific VM [20, 14]. However, the GPU resources are dedicated and cannot be multiplexed.

Device emulation, similar to binary translation in CPU virtualization, is impractical. GPUs, unlike CPUs, whose specifications are not well documented, vary between vendors [15]. Emulating GPUs from different vendors requires vast engineering work. Notably, following up the new GPU hardware would make it a nightmare to maintain the codebase.

API remoting is widely used in commercial softwares such as VMWare and VirtualBox, and has been studied throughout many years. By using API remoting, graphic commands are forwarded from guest OS to host. VMGL [23] and Oracle VirtualBox [7], both based on Chromium [21], replace the standard OpenGL library in Linux Guests with its own implementation to pass the OpenGL commands to VMM. Nonetheless, forwarding OpenGL commands is not considered a general solution, since Microsoft Windows mainly uses their own DirectX API. Whether forwarding OpenGL or DirectX commands, it would be difficult to emulate the other API. gVirtuS [17], VGRIS [25], GViM [19], rCUDA [16] and vCUDA [26] use the same manner to forward CUDA and OpenCL commands, solving the problem of virtualizing GPGPU applications.

VMware's products consist of a virtual PCI device, SVGA II card [15], and the corresponding driver for different operating systems. The emulated device acts like a real video card which has registers, graphics memory and a FIFO command queue. All accesses to the virtual PCI device inside a VM is handled on the host side, by a user-level process, where the actual work is performed. Moreover, they have designed another graphic API called SVGA3D. The SVGA3D protocol is similar to Direct3D and shares a common abstraction. The purpose of SVGA3D is to eliminate the commands for a specific GPU. Meanwhile, a GPU can also emulate the missing features by SVGA3D protocol, which provides a practical portability for their products.

Recently, two full GPU virtualization solutions have been proposed, i.e., gVirt of Intel [29] and GPUvm [28], respectively. gVirt is the first open source product level full GPU virtualization solution in Intel platforms. gVirt presents a vGPU instance to each VM which allows the native graphics driver to be run in VM. The shadow page table is updated with a coarse-grained model, which could lead to a performance pitfall under some video memory intensive workloads, such as media transcoding.

GPUvm presents a GPU virtualization solution on a NVIDIA card. Both para- and full-virtualization were implemented. However, full-virtualization exhibits a considerable overhead for MMIO handling. The performance of optimized para-virtualization is two to three times slower than native. Since NVIDIA has individual graphics memory on the PCI card, while the Intel GPU uses part of main memory as its graphics memory, the way of handling memory virtualization is different. GPUvm cannot handle page faults caused by NVIDIA

GPUs [18]. As a result, they must scan the entire page table when translation lookaside buffer (TLB) flushes. As gHyvi allocates graphics memory within the main memory, VMM can write-protect the page tables to track the page table modifications. This fine-grained page table update mechanism mitigates the overhead incurred by the Massive Update Issue.

NVIDIA GRID [6] is a proprietary virtualization solution from NVIDIA for Kepler architecture. However, there are no technical details about their products available to the public.

6.3 Memory Virtualization

One important aspect in GPU virtualization is memory virtualization, which has been thoroughly researched. The software method employs a shadow page table to reduce the overhead of translating a VM's virtual memory address. This approach could incur severe overhead under some circumstances. Agesen *et al.* [10] listed three situations where the shadow page table cannot handle well: the hidden page fault, address space switching, and the tracing page table entries. They also pointed out some optimization techniques, such as the trace mechanism and eager validating. Unfortunately, it is hard to trade off these mutually exclusive techniques. Therefore, AMD and Intel have added the hardware support for memory virtualization. All three overheads previously listed before can be eliminated, but it is not the silver bullet, a TLB miss punishment is higher in the hardware solution. In the classical VMM implementations, VMM employs a trace technique to prevent its shadow PTEs from becoming inconsistent with guest PTEs, i.e. updating shadow page table strictly after the guest page table is modified. Typically, VM trace uses write-protection mechanism, which can be the source of overhead. This technique is similar to the current gVirt's strict page table shadowing mechanism, which frequently traps and emulates the page faults of the shadow page table, and it causes overhead. gHyvi removes the write-protection from shadow page table to eliminate the overhead caused by excessive trap-and-emulation, taking advantage of the GPU programming model [9].

7 Conclusion and Future Work

gHyvi is an optimized full GPU virtualization solution, based on the Xen hypervisor, with the adaptive hybrid page table shadowing scheme, which improves performance for workloads with the Massive Update Issue when compared to gVirt. To address this issue, this paper provides a hybrid page table shadowing scheme, i.e., strict and relaxed page table shadowing, to provide an optimized full GPU virtualization based on Xen hypervisor for Intel GPUs. gHyvi combines these two page table shadowing mechanisms to reduce VM-exits to the hypervisor. Further, gHyvi automatically switches page table between them by detecting GPU's current workloads, potentially showing significantly improvement to gVirt's performance for workloads with the Massive Update Issue. In order to decide what type of the page need to be reconstructed, four reconstruction policies are introduced. By running the same testcase through the four policies, the dynamic segmented partial reconstruction policy performs the best.

For future work, we will adapt gHyvi to support KVM [22] when gVirt for KVM is ready. Additionally, gHyvi will be released in the open source community soon. We will focus on the areas of portability, scalability, and scheduling issues. With previous GPU command scheduling methods, such as VGRIS and Pegasus [13], we will investigate the low level access pattern of massive page table modification with the detailed analysis of the performance bottleneck of high level applications. We hope this optimized full GPU virtualization solution gives insight into designing the support of efficient distributed systems for GPU acceleration applications.

8 Acknowledgements

We thank our shepherd Dan Tsafrir, Haibo Chen, and the anonymous reviewers for their insightful comments. This work was supported by National Science and Technology Major Project (No. 2013ZX03002004), National R&D Infrastructure and Facility Development Program (No. 2013FY111900), NRF Singapore CREATE Program E2S2, the Shanghai Science and Technology Development Fund for High-Tech Achievement Translation under Grant No. 14511100902, and Shanghai Key Laboratory of Scalable Computing and Systems. Prof. Haibing Guan is the corresponding author.

References

[1] 3dmark06. http://www.futuremark.com.

[2] Amazone high performance computing cloud using gpu. http://aws.amazon.com/hpc/.

[3] Heaven3d. http://unigine.com/products/heaven.

[4] Intel graphics driver. http://www.x.org/wiki/IntelGraphicsDriver/.

[5] Intel processor graphics prm. https://01.org/linuxgraphics/documentation/2013-intel-core-processor-family.

[6] Nvidia grid: Graphics-accelerated virtualization. http://www.nvidia.com/object/grid-technology.html.

[7] Oracle vm virtualbox. https://www.virtualbox.org/.

[8] Passmark2d. http://www.passmark.com.

[9] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices 41*, 11 (2006), 2–13.

[10] AGESEN, O., GARTHWAITE, A., SHELDON, J., AND SUBRAHMANYAM, P. The evolution of an x86 virtual machine monitor. *ACM SIGOPS Operating Systems Review 44*, 4 (2010), 3–18.

[11] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[12] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), IEEE, pp. 44–54.

[13] DEELMAN, E., SINGH, G., SU, M.-H., BLYTHE, J., GIL, Y., KESSELMAN, C., MEHTA, G., VAHI, K., BERRIMAN, G. B., GOOD, J., ET AL. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming 13*, 3 (2005), 219–237.

[14] DONG, Y., DAI, J., HUANG, Z., GUAN, H., TIAN, K., AND JIANG, Y. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 12.

[15] DOWTY, M., AND SUGERMAN, J. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[16] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTÍ, E. S. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on* (2010), IEEE, pp. 224–231.

[17] GIUNTA, G., MONTELLA, R., AGRILLO, G., AND COVIELLO, G. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010-Parallel Processing*. Springer, 2010, pp. 379–391.

[18] GOTTSCHLAG, M., HILLENBRAND, M., KEHNE, J., STOESS, J., AND BELLOSA, F. Logv: Low-overhead gpgpu virtualization. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on* (2013), IEEE, pp. 1721–1726.

[19] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing* (2009), ACM, pp. 17–24.

[20] HIREMANE, R. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine 4*, 10 (2007).

[21] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM Transactions on Graphics (TOG)* (2002), vol. 21, ACM, pp. 693–702.

[22] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[23] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. Vmm-independent graphics acceleration. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), ACM, pp. 33–43.

[24] LOPES, N., AND RIBEIRO, B. Gpumlib: An efficient open-source gpu machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications 3* (2011), 355–362.

[25] QI, Z., YAO, J., ZHANG, C., YU, M., YANG, Z., AND GUAN, H. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO) 11*, 2 (2014), 17.

[26] SHI, L., CHEN, H., SUN, J., AND LI, K. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on 61*, 6 (2012), 804–816.

[27] STRATTON, J. A., RODRIGUES, C., SUNG, I.-J., OBEID, N., CHANG, L.-W., ANSSARI, N., LIU, G. D., AND HWU, W.-M. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* (2012).

[28] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: why not virtualizing gpus at the hypervisor? In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (2014), USENIX Association, pp. 109–120.

[29] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A full gpu virtualization solution with mediated pass-through. In *Proc. USENIX ATC* (2014).

[30] VECCHIOLA, C., PANDEY, S., AND BUYYA, R. High-performance cloud computing: A view of scientific applications. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on* (2009), IEEE, pp. 4–16.

[31] YANG, J., WANG, Y., AND CHEN, Y. Gpu accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics 221*, 2 (2007), 799–804.

Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems

Sharanyan Srikanthan Sandhya Dwarkadas Kai Shen
Department of Computer Science, University of Rochester
{srikanth,sandhya,kshen}@cs.rochester.edu

Abstract

Modern multicore platforms suffer from inefficiencies due to contention and communication caused by sharing resources or accessing shared data. In this paper, we demonstrate that information from low-cost hardware performance counters commonly available on modern processors is sufficient to identify and separate the causes of communication traffic and performance degradation. We have developed SAM, a Sharing-Aware Mapper that uses the aggregated coherence and bandwidth event counts to separate traffic caused by data sharing from that due to memory accesses. When these counts exceed pre-determined thresholds, SAM effects task to core assignments that colocate tasks that share data and distribute tasks with high demand for cache capacity and memory bandwidth. Our new mapping policies automatically improve execution speed by up to 72% for individual parallel applications compared to the default Linux scheduler, while reducing performance disparities across applications in multiprogrammed workloads.

1 Introduction

Multicore processors share substantial hardware resources including last-level cache (LLC) space and memory bandwidth. At the same time, a parallel application with multiple tasks[1] running on different CPU cores may simultaneously access shared data. Both data and resource sharing can result in performance slowdowns and symptoms including high data traffic (bandwidth consumption) and high LLC miss rates. To maximize efficiency, multicore platforms on server, desktop, as well as mobile environments must intelligently map tasks to CPU cores for multiprogrammed and parallel workloads.

Despite similar symptoms, data and resource sharing behaviors require very different task→CPU mapping policies—in particular, applications with strong data sharing benefit from colocating the related tasks on cores that are in proximity to each other (e.g., cores on one socket) while applications with high memory demand or large working sets might best be distributed across sockets with separate cache and memory bandwidth resources. The mapping efficiency is further complicated by the dynamic nature of many workloads.

A large body of prior work has devised scheduling techniques to mitigate resource contention [5–7, 14, 15, 17, 18, 21, 22] in the absence of data sharing. Relatively few have investigated data sharing issues [19, 20] for parallel applications. Tam et al. [19] used direct sampling of data access addresses to infer data sharing behavior. While more recent multicore platforms do sometimes provide such sampled tracing capability, trace processing in software comes at a significant cost. Tang et al. [20] demonstrated the behavior of latency-critical datacenter applications under different task placement strategies. Their results reinforce the need to provide a low-cost, online, automated approach to place simultaneously executing tasks on CPUs for high efficiency.

This paper presents our operating system strategy for task placement that manages data sharing and resource contention in an integrated fashion. In order to separate the impact of data sharing from resource contention, we use aggregate information from existing hardware performance counters along with a one-time characterization of event thresholds that impact performance significantly. Specifically, we use performance counters that identify on- and off-chip traffic due to coherence activity (when data for a cache miss is sourced from another core) and combine this knowledge with LLC miss rates and bandwidth consumption to separate sharing-related slowdown from slowdown due to resource contention.

Our adaptive online *Sharing-Aware Mapper* (**SAM**) uses an iterative, interval-based approach. Based on the sampled counter values in the previous interval, as well as measured thresholds in terms of performance impact using microbenchmarks, we identify tasks that share data and those that have high memory and/or cache demand. We then perform a rebalance in order to colocate the tasks that share data on cores that are in proximity and with potentially shared caches. This decision is weighed against the need to distribute tasks with high bandwidth uses across cores that share fewer resources.

SAM improves the execution speed by up to 72% for stand-alone parallel applications compared to the default Linux scheduler, without the need for user input or ma-

[1] In this paper, a task refers to an OS-schedulable entity such as a process or a thread.

nipulation. For concurrent execution of multiple parallel and sequential applications, our performance improvements are up to 36%, while at the same time reducing performance disparities across applications. The rest of this paper presents the design, implementation, and evaluation of our sharing-aware mapping of tasks to CPUs on multicore platforms.

2 Sharing and Contention Tracking

Hardware counters are commonplace on modern processors, providing detailed information such as the instruction mix, rate of execution, and cache/memory access behavior. These counters can also be read at low latency (on the order of a μSec). We explore the use of commonly available event counters to efficiently track data sharing as well as resource contention on multicores.

Our work addresses several challenges. First, predefined event counters may not precisely suit our information tracking needs. In particular, no single counter reports the data sharing activities on multicores. Secondly, it may be challenging to identify event thresholds that should trigger important control decisions (such as saturating uses of a bottleneck resource). Finally, while many events may be defined in a performance counter architecture, usually only a small number can be observed at one time. For example, the Intel Ivy Bridge architecture used in our evaluation platform can only monitor four programmable counters at a time (in addition to three fixed-event counters).

In this paper, we use these low-cost performance counters to infer valuable information on various bottlenecks in the system. Particularly, we focus on intra- and inter-socket coherence activity, memory bandwidth utilization, and access to remote memory (NUMA). We use the counters and microbenchmarks to analyze the effect of these factors on performance. We obtain thresholds for memory bandwidth utilization and coherence activity that result in significant performance degradation. These thresholds further enable our system to identify and mitigate sharing bottlenecks during execution.

2.1 Coherence Activities

Coherence can be a significant bottleneck in large scale systems. In multithreaded applications, access to shared data and synchronization variables trigger coherence activities when the threads are distributed across multiple cores. When data-sharing threads are colocated on the same multicore socket, coherence is handled within the socket using a high speed bus or a ring. When threads are distributed across sockets, the coherence cost increases significantly due to the higher latency of off-chip access.

Despite the gamut of events monitored by modern day processors, accounting for coherence activity in a manner portable across platforms can still be a challenge. Our goal is to identify performance counters that are available across a range of architectural performance monitoring units, and that can help isolate intra- and inter-socket coherence. We use the following four counters—last-level cache (LLC) hits, LLC misses, misses at the last private level of cache, and remote memory accesses.

In multi-socket machines, there is a clear increase in overhead when coherence activities cross the socket boundary. Any coherence request that can be resolved from within the socket is handled using an intra-socket protocol. If the request cannot be satisfied locally, it is treated as a last-level cache (LLC) miss and handed over to an inter-socket coherence protocol.

We use the cache misses at the last private level, as well as LLC hits and misses, to indirectly infer the intra-socket coherence activities. LLC hit counters count the number of accesses served directly by the LLC and do not include data accesses satisfied by intra-socket coherence activity. Thus, by subtracting LLC hits and misses from the last private level cache misses, we can determine the number of LLC accesses that were serviced by the intra-socket coherence protocol.

To measure inter-socket coherence activity, we exploit the fact that the LLC treats accesses serviced by both off-socket coherence as well as by memory as misses. The difference between LLC misses and memory accesses gives us the inter-socket coherence activity. In our implementation, the Intel Ivy Bridge processor directly supports counting LLC misses that were not serviced by memory, separating and categorizing them based on coherence state. We sum the counters to determine inter-socket coherence activities.

We devise a synthetic microbenchmark to help analyze the performance impact of cross-socket coherence activities. The microbenchmark creates two threads that share data. We ensure that the locks and data do not induce any false sharing. Using a dual-socket machine, we compare the performance of the microbenchmark when consolidating both threads on a single socket against execution when distributing them across sockets (a common result of Linux's default scheduling strategy). The latter induces inter-socket coherence traffic while all coherence traffic uses the on-chip network in the former case.

We vary the rate of coherence activity by changing the ratio of computation to shared data access within each loop in order to study its performance impact. Figure 1 shows the performance of consolidating the threads onto the same socket relative to distributing across sockets. At low coherence traffic rates, the consolidation and distribution strategies do not differ significantly in performance, but when the traffic increases, the performance improvement from colocation can be quite substantial.

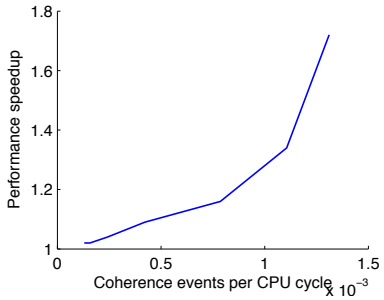


Figure 1: Speedup when consolidating two threads that share data onto the same socket (eliminating inter-socket coherence) in comparison to distributing them across sockets as coherence activity is varied.

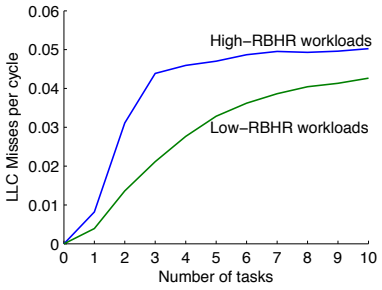


Figure 2: Socket-level memory bandwidth usage (measured by the LLC_MISSES performance counter) for workloads of high and low row buffer hit ratios (RBHRs) at increasing number of tasks.

We use these experiments to identify a per-thread coherence activity threshold at which inter-socket coherence causes substantial performance degradation (which we identify as a degradation of $>5\%$). Specifically on our experimental platform with 2.2 GHz processors, the threshold is 2.5×10^{-4} coherence events per CPU cycle, or 550,000 coherence events per second.

2.2 Memory Bandwidth Utilization

Our approach requires the identification of a memory bandwidth utilization threshold that signifies the resource exhaustion and likely performance degradation. Since all cores on a socket share the access to memory, we account for this resource threshold on a per-socket basis—aggregating the memory bandwidth usage of all tasks running on each particular socket.

One challenge we face is that the maximum bandwidth utilization is not a static hardware property but it further depends on the row buffer hit ratio (RBHR) of the memory access loads. DRAM rows must be pre-charged and activated before they can be read or written to. For spatial locality, DRAMs often activate an entire row of data (on the order of 4 KB) instead of just the cache line being accessed. Once this row is opened, subsequent accesses to the same row incur much lower latency and consume less device time. Hence, the spatial locality in an application's memory access pattern plays a key role in its performance and resource utilization.

Figure 2 shows the aggregate memory bandwidth used with increasing numbers of tasks. All tasks in each test run on one multicore socket in our machine. We show results for both low- and high-RBHR workloads, using a memory copy microbenchmark where the number of contiguous words copied is one cache line within a row or an entire row, respectively. Behaviors of most real-world applications lie between these two curves. We can clearly see that the bandwidth available to a task is indeed affected by its RBHR. For example, on our machine, three tasks with high RBHR may utilize more memory bandwidth than ten low-RBHR tasks do. If we use the maximum bandwidth usage of high-RBHR workloads as the available memory resource, then a low-RBHR workload would never reach it and therefore be always determined as having not used the available resource (even when the opposite is true).

In this paper, we are more concerned with detecting and mitigating memory bandwidth bottlenecks than utilizing the memory bandwidth at its maximum possible level. We can infer from Figure 2 that the difference between the low / high-RBHR bottleneck bandwidths (at 10 tasks) is about 10%. We conservatively use the low RBHR bandwidth as the maximum available without attempting to track RBHR. Our high-resource-use threshold is set at 20% below the above-defined maximum available memory bandwidth. Specifically on our experimental platform with 2.2 GHz processors, the threshold is 0.034 LLC misses per cycle, or 75,000,000 LLC misses per second. A conservative setting might result in detecting bottlenecks prematurely but will avoid missing ones that will actually result in performance degradation.

2.3 Performance Counters on Ivy Bridge

Our experimental platform contains processors from Intel's Ivy Bridge line. On our machine, cores have private L1 and L2 caches, and an on-chip shared L3 (LLC). The Intel Performance Monitoring Unit (PMU) provides the capability of monitoring certain events (each of which may have several variants) at the granularity of individual hardware contexts. Instructions, cycles, and unhalted cycles can be obtained from fixed counters in the PMU. The remaining events must use the programmable counters.

We encounter two constraints in using the programmable counters—there are only four programmable counters and we can monitor only two variants of any particular event using these counters. Solutions to both constraints require multiplexing the programmable counters. We separate the counters into disjoint groups and then

alternate the groups monitored during successive intervals of application execution, effectively sampling each counter group from a partial execution.

In order to monitor intra-socket coherence activity, we use the main event MEM_LOAD_UOPS_RETIRED. One caveat is that since the event is a load-based event, only reads are monitored. In our experiments, we found that since typical long-term data use involves a read followed by a write, these counters were a good indicator of coherence activity. We use three variants of this event, namely L2_MISS (misses at the last private level of cache), L3_HIT (hits at, or accesses serviced by, the LLC), and L3_MISS (misses at the LLC), and use these events to infer the intra-socket coherence activity, as described in Section 2.1.

To obtain inter-socket coherence activity, we use the MEM_LOAD_UOPS_LLC_MISS_RETIRED event. We get the inter-socket coherence activity by summing up two variants of this event—REMOTE_FWD and REMOTE_HITM.

We read the remote DRAM access event count from MEM_LOAD_UOPS_LLC_MISS_RETIRED: REMOTE_DRAM.

We use the LLC_MISSES event (which includes both reads and writes) as an estimate of overall bandwidth consumption. Although the LLC_MISSES event count includes inter-socket coherence and remote DRAM access events, since these latter are prioritized in our algorithm, further division of bandwidth was deemed unnecessary.

In total, we monitor seven programmable events, with three variants for each of two particular events, allowing measurement of one sample for each event across two iterations. Since different interval sizes can have statistical variation, we normalize the counter values with the measured unhalted cycles for the interval.

3 SAM: Sharing-Aware Mapping

We have designed and implemented SAM, a performance monitoring and adaptive mapping system that simultaneously reduces costly communication and maximizes resource utilization efficiency for the currently executing tasks. We identify resource sharing bottlenecks and their associated costs/performance degradation impact. Our mapping strategy attempts to shift load away from such bottleneck resources.

3.1 Design

Using thresholds for each bottleneck as described in Section 2, we categorize each task based on whether its characteristics exceed these thresholds. The four activity categories of interest are: inter-socket coherence, intra-socket coherence, remote DRAM access, and per-socket memory bandwidth demand.

Figure 3 defines the symbols that are used for SAM's task→CPU mapping algorithm illustrated in Figure 4. Reducing inter-socket coherence activity by colocating tasks has the highest priority. Once tasks with high inter-socket coherence traffic are identified, we make an attempt to colocate the tasks by moving them to sockets with idle cores that already contain tasks with high inter-socket coherence activity, if there are any. If not, we use our task categorization to distinguish CPU or memory bound tasks—SAM swaps out memory intensive tasks only after swapping CPU bound tasks, and avoids moving tasks with high intra-socket coherence activity. Moving memory-intensive tasks is our reluctant last choice since they might lead to expensive remote memory accesses.

Our second priority is to reduce remote memory accesses. Remote memory accesses are generally more expensive than local memory accesses. Since a reluctant migration might cause remote memory accesses, we register the task's original CPU placement prior to the migration. We then use this information to relocate the task back to its original socket whenever we notice that it incurs remote memory accesses. To avoid disturbing other tasks, we attempt to swap tasks causing remote memory accesses with each other whenever possible. Otherwise, we search for idle or CPU-bound tasks to swap for the task that generates remote memory accesses.

After reducing remote memory accesses, we look to balancing memory bandwidth utilization. SAM identifies memory-intensive tasks on sockets where the bandwidth is saturated. The identified tasks are relocated to other sockets whose bandwidth is not entirely consumed. We track the increase in bandwidth utilization after every migration to avoid overloading a socket with too many memory-intensive tasks. In some situations, such relocation can cause an increase in remote memory accesses, but this is normally less damaging than saturating the memory bandwidth.

3.2 Implementation Notes

Performance counter statistics are collected on a per task (process or thread) basis with the values accumulated in the task control block. Performance counter values are read on every operating system tick and the values are attributed to the currently executing task.

Our mapping strategy is implemented as a kernel module that is invoked by a privileged daemon process. The collected counter values for currently executing tasks are examined at regular intervals. Counter values are first normalized using unhalted cycles and then used to derive values for memory bandwidth utilization, remote memory accesses, and intra- and inter-socket coherence activity for each running task. These values are attributed to

$\mathcal{C}_T$: *Per task inter-socket coherence threshold*
M_T : *Per task memory utilization threshold*
$\mathcal{R}_T$: *Per task remote memory access threshold*
$\mathcal{S}$: *Set of all sockets*
$\mathcal{C}_i$: *Set of cores in socket i*
$\mathcal{C}_{i,j}^{inter}$: $Inter-socket\ coherence\ activity\ generated\ by\ core\ j\ on\ socket\ i$
$\mathcal{C}_{i,j}^{intra}$: $Intra-socket\ coherence\ activity\ generated\ by\ core\ j\ on\ socket\ i$
$M_{i,j}$: *Memory bandwidth utilization by core j on socket i*
$\mathcal{R}_{i,j}$: *Remote memory accesses by core j on socket i*
$\mathcal{C}ycles_{i,j}^{inter}$: $Cycles\ count\ for\ core\ j\ on\ socket\ i$
$\mathcal{P}_i^{Mem}$: $\{\ j \mid j\ \in \mathcal{C}_i \wedge M_{i,j} > M_T\ \}$
$\mathcal{P}_i^{Rem}$: $\{\ j \mid j\ \in \mathcal{C}_i \wedge R_{i,j} > R_T\ \}$
$\mathcal{P}_i^{Idle}$: $\{\ j \mid j\ \in \mathcal{C}_i \wedge Cycles_{i,j} = 0\}$
$\mathcal{P}_i^{inter}$: $\{\ j \mid j\ \in \mathcal{C}_i \wedge \mathcal{C}_{i,j}^{inter} > \mathcal{C}_T\ \}$
$\mathcal{P}_i^{intra}$: $\{\ j \mid j\ \in \mathcal{C}_i \wedge \left(\mathcal{C}_{i,j}^{inter} \leq \mathcal{C}_T\right) \wedge \left(\mathcal{C}_{i,j}^{intra} \geq \mathcal{C}_T\right)\}$
$\mathcal{P}_i^{CPU}$: $\mathcal{C}_i - \mathcal{P}_i^{Mem} - \mathcal{P}_i^{Idle} - \mathcal{P}_i^{inter} - \mathcal{P}_i^{intra}$
M_i : $\sum_{j \in \mathcal{C}_i} M_{i,j}$
$Soc_{task(i,j)}$: $Original\ socket\ from\ which\ the\ task\ currently\ running\ on\ socket\ i, core\ j\ was\ reluctantly\ migrated.$

Figure 3: SAM algorithm definitions.

the corresponding core and used to update/maintain per-core and per-socket data structures that store the consolidated information. Per-core values are added to determine per-socket values.

Task relocation is accomplished by manipulating task affinity (using `sched_setaffinity` on Linux) to restrict scheduling to specific cores or sockets. This allows seamless inter-operation with Linux's default scheduler and load balancer.

SAM's decisions are taken at 100 mSec intervals. We performed a sensitivity analysis on the impact of the interval length and found that while SAM was robust to changes in interval size (varied from 10 mSecs to over 1 second), a 100-mSec interval hit the sweet spot in terms of balancing reaction times. We find that we can detect and decide on task migrations effectively at this rate. We were able to obtain good control and response for intervals up to one second. Effecting placement changes at intervals higher than a second can reduce the performance benefits due to lack of responsiveness.

4 Evaluation

We assess the effectiveness of our hardware counter-based sharing and contention tracking, and evaluate the performance of our adaptive task→CPU mapper. Our performance monitoring infrastructure was implemented in Linux 3.14.8. Our software environment is Fedora 19 running GCC 4.8.2. We conducted experiments on a dual-socket machine with each socket containing an Intel Xeon E5-2660 v2 "Ivy Bridge" processor (10 physical cores with 2 hyperthreads each, 2.20 GHz, 25 MB of L3 cache). The machine has a NUMA configuration in which each socket has an 8 GB local DRAM partition.

```
// Inter-socket coherence activity found. Need to colocate appropriate tasks.
for every i ∈ S if (P_i^inter != ∅)
    for every j ∈ S ∧ (j != i)
        while ((|P_i^inter + P_i^intra| < |C_i|) ∧ (P_j^inter != ∅))
            while (P_i^Idle != ∅ ∧ P_j^inter != ∅)
                move (P_j^inter[0], P_i^Idle[0])
            while (P_i^CPU != ∅ ∧ P_j^inter != ∅)
                swap (P_j^inter[0], P_i^CPU[0])
            while (P_i^Mem != ∅ ∧ P^inter != ∅)
                swap (P_j^inter[0], P_i^Mem[0]), let a = P_i^Mem[0]
                // This is a reluctant task migration. Store original
                // socket id to restore task when possible.
                if (Soc_task(j,a) == −1)
                    Soc_task(j,a) = i

// Mitigate any remote memory accesses encountered.
for every i ∈ S if (P_i^Rem != ∅)
    for every j ∈ P_i^Rem, let a = Soc_task(i,j)
        // Swap with a task migrated reluctantly from the current socket.
        if (∃_{k∈C_a}(Soc_task(a,k) == i))
            swap (j, k)
        else if (P_a^Idle != ∅)
            move (j, P_a^Idle[0])
        else if (P_a^CPU != ∅)
            swap (j, P_a^CPU[0])
        else if (P_a^Mem != ∅)
            swap (j, P_a^Mem[0]), let b = P_a^Mem[0]
            if (Soc_task(a,b) == −1)
                Soc_task(a,b) = i

// Balance the memory intensive load to other sockets.
for every i ∈ S if (P_i^Mem != ∅) ∧ (M_i > M_T)
    for every j ∈ S if ((j != i) ∧ (M_j < M_T))
        // Balance memory intensive tasks across sockets.
        // If cores are unavailable, look for other sockets to balance load.
        res = balance(P_i^Mem, P_j^Mem)
        if (res == Balance_Sucessful)
            break
```

Figure 4: SAM task→CPU mapping algorithm.

4.1 Benchmarks

We use a variety of benchmarks from multiple domains in our evaluation. First, we use the synthetic *microbenchmarks* described in Sections 2.1 and 2.2 that stress the inter- and intra-socket coherence and memory bandwidth respectively. We create two versions of the coherence activity microbenchmark—one (Hubench) generating a near-maximum rate of coherence activity (1.3×10^{-3} coherence events per CPU cycle) and another (Lubench) generating coherence traffic close to the threshold we identified in Section 2 (2.6×10^{-4} coherence events per cycle). We also use the high-RBHR *memcpy* microbenchmark (MemBench). MemBench saturates the memory bandwidth on one socket and needs to be distributed across sockets to maximize memory bandwidth utilization.

PARSEC 3.0 [2] is a parallel benchmark suite containing a range of applications including image processing, chip design, data compression, and content similarity search. We use a subset of the PARSEC benchmarks that exhibit non-trivial data sharing. In particular, Canneal uses cache-aware simulated annealing to minimize the routing cost of a chip design. Bodytrack is a computer vision application that tracks a human body through an image sequence. Both benchmarks depend on data that

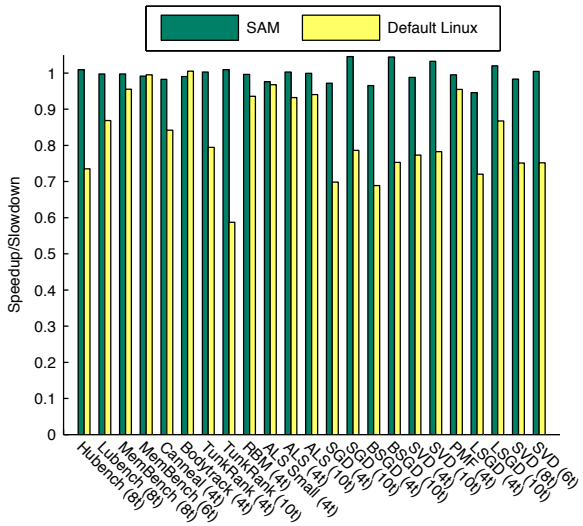


Figure 5: Performance of standalone applications on Linux with SAM and without (default Linux) SAM. The performance metric is the execution speed (the higher the better) normalized to that of the best static task→CPU mapping determined through offline testing.

is shared among its worker threads. We observe that the data sharing however is not as high as other workloads that we discuss below.

The *SPEC CPU2006* [1] benchmark suite has a good blend of memory-intensive and CPU-bound applications. Many of its applications have significant memory utilization with high computation load as well. The memory-intensive applications we use are libq, soplex, mcf, milc, and omnetpp. The CPU-bound applications we use are sjeng, bzip2, h264ref, hmmer, and gobmk.

GraphLab [13] and *GraphChi* [12] are emerging systems that support graph-based parallel applications. Unlike most PARSEC applications, the graph-based applications tend to have considerable sharing across worker threads. In addition, the number of worker threads is not always static and is dependent on the phase of execution and amount of parallelism available in the application. Such phased behaviors are a good stress test to ascertain SAM's stability of control. Our evaluation uses a range of machine learning and filtering applications—TunkRank (Twitter influence ranking), Alternating Least Squares (ALS) [23], Stochastic gradient descent (SGD) [11], Singular Value Decomposition (SVD) [10], Restricted Bolzman Machines (RBM) [8], Probabilistic Matrix Factorization (PMF) [16], Biased SGD [10], and Lossy SDG [10].

4.2 Standalone Application Evaluation

We first evaluate the impact of SAM's mapping decisions on standalone parallel application executions. Figure 5 shows the performance obtained by SAM and the default Linux scheduler. The performance is normalized to that of the best static task→CPU mapping obtained offline for each application.

Application	Coherence	Remote memory	Workload characteristic
Hubench (8t)	4	0	Data sharing
Lubench (8t)	4	0	Data sharing
MemBench (8t)	0	0	Memory bound
MemBench (6t)	0	0	Memory bound
Canneal (4t)	8	3	CPU bound
Bodytrack (4t)	2	0	CPU bound
TunkRank (4t)	6	1	Data sharing
TunkRank (10t)	13	1	Data sharing
RBM (4t)	12	0	CPU bound
ALS (4t) Small Dataset	1	0	CPU bound
ALS (4t)	6	3	CPU bound
ALS (10t)	9	1	CPU bound
SGD (4t)	12	2	Data sharing
SGD (10t)	14	3	Data sharing
BSGD (4t)	8	2	Data sharing
BSGD (10t)	20	3	Data sharing
SVD (4t)	12	1	Data sharing
SVD (10t)	24	7	Data sharing
PMF (4t)	20	0	CPU bound
LSGD (4t)	6	3	Data sharing
LSGD (10t)	18	3	Data sharing
SVD (8t)	22	6	Data sharing
SVD (6t)	18	6	Data sharing

Table 1: Actions taken by our scheduler for each standalone application run. *Coherence* indicates the number of task migrations performed to reduce inter-socket coherence. *Remote memory* indicates the number of task migrations performed to reduce remote memory accesses. *Workload characteristic* classifies each application as either CPU bound, data sharing intensive, or memory bound.

We can see that SAM considerably improves the application performance. Our task placement mirrors that of the best configuration obtained offline, thereby resulting in almost identical performance. Improvement over the default Linux scheduler can reach 72% in the best case and varies in the range of 30–40% for most applications. This performance improvement does not require any application changes, application user/deployer knowledge of system topology, or need for recompilation.

Parallel applications that have nontrivial data sharing among its threads benefit from SAM's remapping strategy. Figure 6(A) shows the average per-thread instructions per unhalted cycle (IPC). SAM results in increased IPC for almost all the applications. Figure 6(B) demonstrates that we have almost eliminated all the per-thread inter-socket coherence traffic, replacing it with intra-

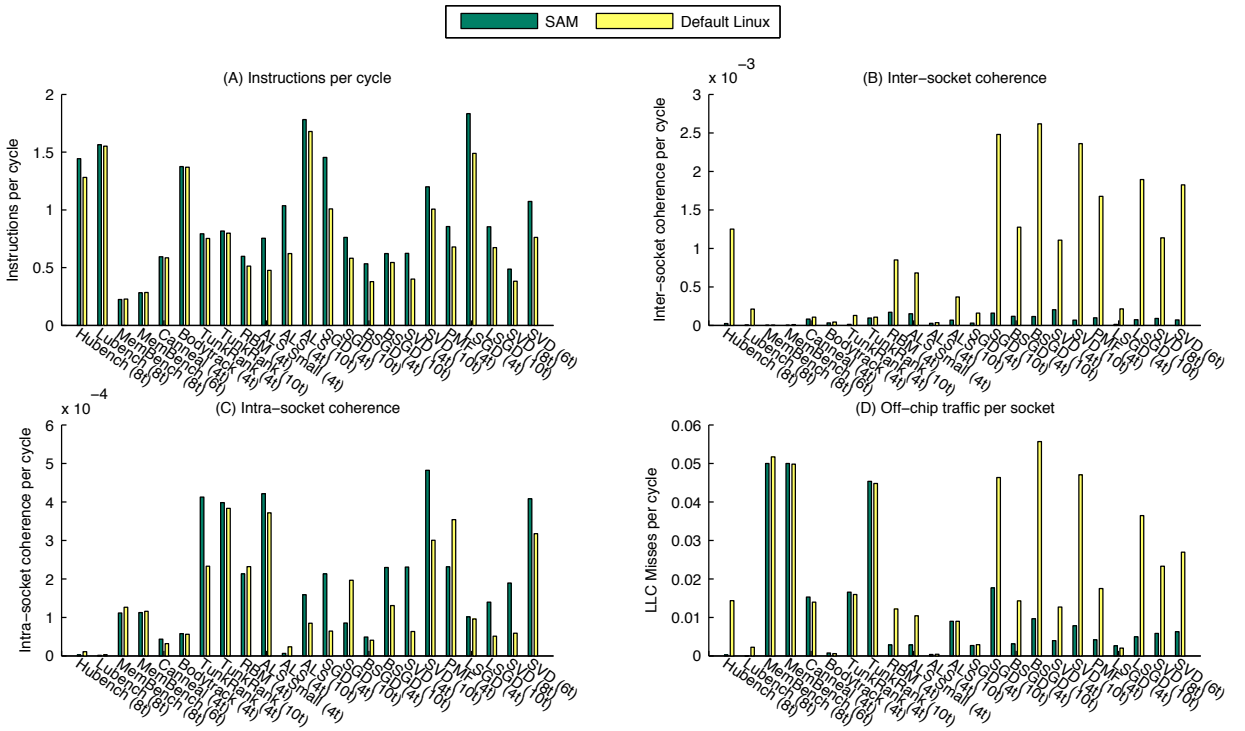


Figure 6: Measured hardware metrics for standalone applications. Fig. (A): per-thread instructions per unhalted cycle (IPC); Fig. (B): per-thread inter-socket coherence activity; Fig. (C): per-thread intra-socket coherence activity; Fig. (D): per-socket LLC misses per cycle. All values are normalized to unhalted CPU cycles.

socket coherence traffic (Figure 6(C)). Figure 6(D) uses LLC misses per socket per cycle to represent aggregate per socket off-chip traffic. We can see that off-chip traffic for applications with high data sharing is reduced. Table 1 summarizes the mapping actions (whether due to inter-socket coherence or remote memory access) taken for every workload shown in Figure 5.

Interestingly, although Figure 6(B) shows that TunkRank (Twitter influence ranking) has fairly low average inter-socket coherence activity, it shows the most performance boost with SAM. In this application, there are phases of high coherence activity between phases of heavy computation, which SAM is able to capture and eliminate. The application also scales well with additional processors (high inherent parallelism). The better a data-sharing application scales, the higher the speedup when employing SAM. With SAM, TunkRank scales almostlinearly on our multicore platform.

While most workloads achieve good speedup, in some cases SAM's performance is only on par with (or slightly better than) default Linux. Such limited benefits are due to two primary reasons. First, applications may be CPU bound without much coherence or memory traffic. ALS, RBM, and PMF are examples of such applications. Although inter-socket coherence activity for these applications is above threshold, resulting in some migrations (see Table 1) due to SAM's affinity control, the relative impact of this colocation is small. Second, some workloads are memory intensive but contain low inter-socket coherence activity. Linux uses a static heuristic of distributing tasks across sockets, which is already the ideal action for these workloads. In such cases, SAM does not have additional room to improve performance.

Table1 does not contain a column for migrations to balance memory bandwidth because for standalone applications, SAM does not effect such migrations. First, for purely memory intensive workloads (MemBench), the default strategy to distribute tasks works ideally and therefore we do not perform any more migrations. Second, for workloads that are both memory intensive and with high data sharing, SAM prioritizes inter-socket coherence activity avoidance over balancing memory bandwidth utilization.

Graph-based machine learning applications exhibit phased execution and dynamic task creation and parallelism (mainly in the graph construction and distribution stages). The burst of task creation triggers load balancing in Linux. While Linux respects core affinity, tasks may be migrated to balance load. Linux task migrations do not sufficiently consider application characteristics and may result in increased remote DRAM accesses, inter-socket communication, resource contention, and bandwidth sat-

uration. SAM is able to mitigate the negative impact of these load balancing decisions, as demonstrated by the non-trivial number of migrations due to coherence activity and remote memory accesses performed by SAM (Table 1). SAM constantly monitors the system and manipulates processor affinity (which the Linux load balancer respects) to colocate or distribute tasks across sockets and cores.

4.3 Multiprogrammed Workload Evaluation

Table 2 summarizes the various application mixes we employed to evaluate SAM's performance on multiprogrammed workloads. They are designed to produce different levels of data sharing and memory utilization. Two factors come into play. First, applications incur performance penalties due to resource competition. Second, bursty and phased activities disturb the system frequently.

In order to compare the performance of different mixes of applications, we first normalize each application's runtime with its offline optimum standalone runtime, as in Figure 5. We then take the geometric mean of the normalized performance of all the applications in the workload mix (counting each application once regardless of the number of tasks employed) to derive a single metric that represents the speedup of the mix of applications. We use this metric to compare the performance of SAM against the default Linux scheduler.

The performance achieved is shown in Figure 7. Naturally, due to competition for resources, most applications will run slower than their offline best standalone execution. Even if applications utilize completely difference resources, they may still suffer performance degradation. For example, when inter-socket coherence mitigation conflicts with memory load balancing, no scheduler can realize the best standalone performance for each co-executing application.

SAM outperforms the default Linux scheduler by 2% to 36% as a result of two main strategies. First, we try to balance resource utilization whenever possible without affecting coherence traffic. This benefits application mixes that have both applications that share data and use memory. Second, we have information on inter-socket coherence activity and can therefore use it to colocate tasks that share data. However, we do not have exact information to indicate which tasks share data with which other tasks. We receive the validation of a successful migration if it produces less inter-socket and more intra-socket activity after the migration. Higher intra-socket activity helps us identify partial or full task groupings inside a socket.

For the mixes of microbenchmarks (#1–#3), SAM executes the job about 25% faster than Linux. SAM's relative performance approaches 1—indicating comparable performance to the case that all microbenchmarks reach respective optimum offline standalone performance simultaneously. SAM is able to preferentially colocate tasks that share data over tasks that are memory intensive. Since the memory benchmark saturates memory bandwidth at fairly low task counts, colocation does not impact performance.

Multiprog. workload #	Application mixes
1	12 MemBench, 8 HuBench
2	14 MemBench, 6 HuBench
3	10 MemBench, 6 HuBench, 4 CPU
4	2 libq, 2 bzip2, 2 sjeng, 2 omnetpp
5	2 libq, 2 soplex, 2 gobmk, 2 hmmer
6	2 mcf, 2 milc, 2 sjeng, 2 h264ref
7	2 milc, 2 libq, 2 h264ref, 2 sjeng
8	2 mcf, 2 libq, 2 h264ref, 2 sjeng, 4 TunkRank
9	10 SGD, 10 BSGD
10	10 SGD, 10 LSGD
11	10 LSGD, 10 BSGD
12	10 LSGD, 10 ALS
13	10 SVD, 10 SGD
14	10 SVD, 10 BSGD
15	10 SVD, 10 LSGD
16	10 SVD, 10 LSGD
17	20 SGD, 20 BSGD
18	20 SGD, 20 LSGD
19	20 LSGD, 20 BSGD
20	20 LSGD, 20 ALS
21	20 SVD, 20 SGD
22	20 SVD, 20 BSGD
23	20 SVD, 20 LSGD
24	16 BSGD, 10 MemBench, 14 CPU
25	16 LSGD, 10 MemBench, 14 CPU
26	16 SGD, 10 MemBench, 14 CPU

Table 2: Multiprogrammed application mixes. For each mix, the number preceding the application's name indicates the number of tasks it spawns. We generate various combinations of applications to evaluate scenarios with varying data sharing and memory utilization.

The speedups obtained for the SPEC CPU benchmark mixes relative to Linux are not high. The SPEC CPU mix of applications tend to be either memory or CPU bound. Since Linux prefers to split the load among sockets, it can perform well for memory intensive workloads. We perform significantly better than the Linux scheduler when the workload mix comprises applications that share data and use memory bandwidth.

A caveat is that Linux's static task→CPU mapping policy is very dependent on the order of task creation. For example, different runs of the same SPECCPU mix of applications discussed above can result in a very different sequence of task creation, forcing CPU-bound tasks to be colocated on a socket and memory-intensive tasks

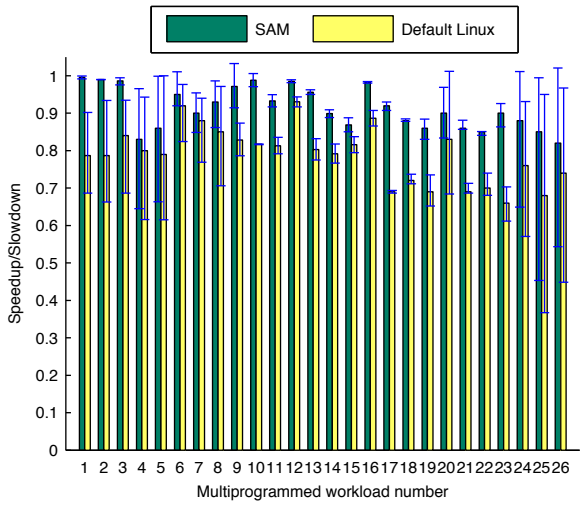


Figure 7: Speedup (execution time normalized to the offline optimum standalone execution: higher is better) of multiprogrammed workloads with (SAM) and without (default Linux) SAM. We show the geometric mean of all application speedups within each workload as well as the max-min range (whiskers) for the individual applications.

to be colocated on the other. We use the best-case Linux performance as our comparison base. Since SAM observes and reacts to the behavior of individual tasks, its scheduling performance does not depend heavily on the task creation order.

Figure 7 also plots the minimum and maximum speedups of applications in each workload mix using whiskers. The minimum and maximum speedups are important to understand the fairness of our mapping strategy. The application that has the minimum speedup is slowed down the most in the workload mix. The application that has the maximum speedup is the least affected by the contention in the system. We can see that SAM improves upon the fairness of the Linux scheduler in a significant fashion. The geometric mean of the minimum speedup for all the workload mixes for SAM is 0.83. The same for the default Linux scheduler is 0.69. Similarly, the geometric mean of the maximum speedup for all workload mixes for SAM and Linux are 0.96 and 0.87 respectively. From this analysis, we can conclude that in addition to improving overall performance, SAM reduces the performance disparity (a measure of fairness) among multiple applications when run together.

Figure 8(A) plots the per-thread instructions per cycle for the mixed workloads. We can see that largely, the effect of our actions is to increase the IPC. As with the standalone applications, Figure 8(B) shows that SAM significantly reduces inter-socket coherence activity, replacing it with intra-socket coherence activity (Figure 8(C)). Note that for workloads 14 and 15, SAM shows reductions in both intra- and inter-socket coherence. This is likely due to working sets that exceed the capacity of the private caches, resulting in hits in the LLC when migrations are effected to reduce inter-socket coherence.

Figure 8(D) uses LLC misses per cycle to represent aggregate per socket off-chip traffic. Our decision to prioritize coherence activity can lead to reduced off-chip traffic, but this may be counteracted by an increase in the number of remote memory accesses. At the same time, our policy may also reduce main memory accesses by sharing the last level cache with tasks that actually share data (thereby reducing pressure on LLC cache capacity). We also improve memory bandwidth utilization when possible without disturbing the colocated tasks that share data. Workload mixes #24–#26 have a combination of data-sharing, memory-intensive, and CPU-bound tasks. In these cases, SAM improves memory bandwidth utilization by moving the CPU-bound tasks to accommodate distribution of the memory-intensive tasks.

In our experiments, both hardware prefetcher and hyperthreading are turned on by default. Hyperthreads add an additional layer of complexity to the mapping process due to resource contention for logical computational units as well as the private caches. Since SAM utilizes one hardware context on each physical core before utilizing the second, when the number of tasks is less than or equal to the number of physical cores, SAM's policy decisions are not affected by hyperthreading.

In order to determine the interaction of the prefetcher with the SAM mapper, we compare the relative performance of SAM when turning off prefetching. We observe that on average, prefetching is detrimental to the performance of multiprogrammed workloads with or without the use of SAM. The negative impact of prefetching when using SAM is slightly lower than with default Linux.

4.4 Overhead Assessment

SAM's overhead has three contributors: accessing performance counters, making mapping decisions based on the counter values, and migrating tasks to reflect the decisions. In our prototype, reading the performance counters, a cost incurred on every hardware context, takes takes 8.89 μSecs. Counters are read at a 1-mSec interval. Mapping decisions are centralized and are taken at 100 mSecs intervals. Each call to the mapper, including the subsequent task migrations, takes about 9.97 μSecs. The overall overhead of our implementation is below 1%. We also check the overall system overhead by running all our applications with our scheduler but without performing any actions on the decisions taken. There was no discernible difference between the two runtimes, meaning that the overhead is within measurement error.

Currently, SAM's policy decisions are centralized in

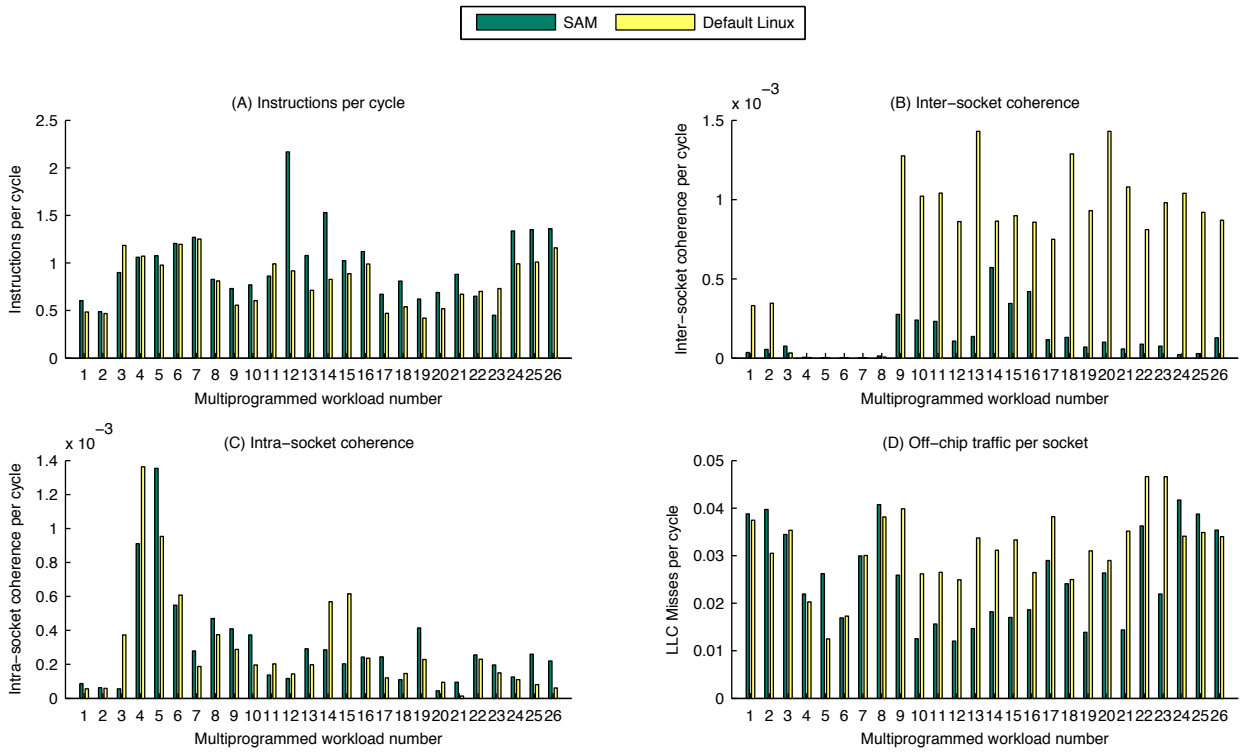


Figure 8: Measured hardware metrics for multiprogrammed application workloads. Fig. (A): per-thread instructions per unhalted cycle (IPC); Fig. (B): per-thread inter-socket coherence activity; Fig. (C): per-thread intra-socket coherence activity; Fig. (D): per-socket LLC misses per cycle. All values are normalized to unhalted CPU cycles.

a single daemon process, which works well for our 40-CPU machine, since most of the overhead is attributed to periodically reading the performance counters. Data consolidation in the daemon process has a linear complexity on the number of processors and dominates in cost over the mapping decisions. For all practical purposes, SAM scales linearly with the number of processors. In the future, if the total number of cores is very large, mapping policies might need to be distributed for scalability.

4.5 Sensitivity Analysis

We study the sensitivity of SAM's performance gains to the thresholds identified in Section 2 for determining high coherence activity and memory bandwidth consumption.

Too low a coherence activity threshold can classify applications with little benefit from colocation as ones with high data sharing; it can also result in misclassifying tasks that have just been migrated as ones with high data sharing due to the coherence activity generated as a result of migration. For some workloads, this may reduce the ability to beneficially map tasks with truly high data sharing. Too high a coherence activity threshold can result in not identifying a task as high data sharing when it could benefit from colocation.

We found that SAM's performance for our workloads was relatively resilient to a wide range of values for the threshold. The coherence activity threshold could be varied from 10% of the value determined in Section 2 to 2 times this value. Using threshold values below the low end result in a loss of performance of up to 9% for one mixed workload. Using threshold values >2 times the determined threshold for the mixed workloads and >3 times for the standalone applications results in a loss of up to 18% and 30% performance respectively for a couple of workloads.

SAM's resilience with respect to the memory bandwidth threshold is also quite good, although the range of acceptable values is tighter than for coherence activity. For the memory bandwidth threshold, too low a value can lead to prematurely assuming that a socket's memory bandwidth is saturated, resulting in lost opportunities for migration. Too high a value can result in performance degradation due to bandwidth saturation. In the case of the SPECCPU benchmarks, lowering our determined threshold by 30% resulted in both sockets being considered as bandwidth saturated, and a performance loss of up to 27% due to lost migrations opportunities. Similarly, for MemBench, raising the memory threshold by 25% leads to non-recognition of bandwidth saturation, which produces up to 85% loss in performance.

5 Related Work

The performance impact of contention and interference on shared multicore resources (particularly the LLC cache, off-chip bandwidth, and memory) has been well recognized in previous work. Suh et al. [18] used hardware counter-assisted marginal gain analysis to minimize the overall cache misses. Software page coloring [5, 22] can effectively partition the cache space without special hardware features. Mutlu et al. [15] proposed parallelism-aware batch scheduling in DRAM to reduce inter-task interference at the memory level. Mars et al. [14] utilized active resource pressures to predict the performance interference between colocated applications. These techniques, however, manage multicore resource contention without addressing the data sharing issues in parallel applications.

Blagodurov et al. [3] examined data placement in non-uniform memory partitions but did not address data sharing-induced coherence traffic both within and across sockets. Tam et al. [19] utilized address sampling (available on Power processors) to identify task groups with strong data sharing. Address sampling is a relatively expensive mechanism to identify data sharing (compared to our performance counter-based approach) and is not available in many processors in production today.

Calandrino and Anderson [4] proposed cache-aware scheduling for real-time schedulers. The premise of their work is that working sets that do not fit in the shared cache will cause thrashing. The working set size of an application was approximated to the number of misses incurred at the shared cache. Their scheduling is aided with job start times and execution time estimates, which non-real time systems often do not have. Knauerhase et al. [9] analyze the run queue of all processors of a system and schedule them so as to minimize cache interference. Their scheduler is both fair and less cache contentious. They do not, however, consider parallel workloads and the effect of data sharing on the last level cache.

Tang et al. [20] demonstrate the impact of task placement on latency-critical datacenter applications. They show that whether applications share data and/or have high memory demand can dramatically affect performance. They suggest the use of information on the number of accesses to "shared" cache lines to identify intra-application data sharing. While this metric is a useful indicator of interference between tasks, it measures only one aspect of data sharing—the cache footprint, but misses another important aspect of data sharing—off-chip traffic in the case of active read-write sharing. Additionally, their approach requires input statistics from an offline stand-alone execution of each application.

Previous work has also pursued fair uses of shared multicore resources between simultaneously executing tasks. Ebrahimi et al. [6] proposed a new hardware design to track contention at different cache/memory levels and throttle tasks with unfair resource usage or disproportionate progress. At the software level, fair resource use can be accomplished through scheduling quantum adjustment [7] or duty cycle modulation-enabled speed balancing [21]. These techniques are complementary and orthogonal to our placement strategies, and can be used in conjunction with our proposed approach for improved fairness and quality of service.

6 Conclusions

In this paper, we have designed and implemented a performance monitoring and sharing-aware adaptive mapping system. Our system works in conjunction with Linux's default scheduler to simultaneously reduce costly communications and improve resource utilization efficiency. The performance monitor uses commonly available hardware counter information to identify and separate data sharing from DRAM memory access. The adaptive mapper uses a cost-sensitive approach based on the performance monitor's behavior identification to relocate tasks in an effort to improve both parallel and mixed workload application efficiency in a general-purpose machine. We show that performance counter information allows us to develop effective and low-cost mapping techniques without the need for heavy-weight access traces. For stand-alone parallel applications, we observe performance improvements as high as 72% without requiring user awareness of machine configuration or load, or performing compiler profiling. For multiprogrammed workloads consisting of a mix of parallel and sequential applications, we achieve up to 36% performance improvement while reducing performance disparity across applications in each mix. Our approach is effective even in the presence of workload variability.

Acknowledgments This work was supported in part by the U.S. National Science Foundation grants CNS-1217372, CCF-1217920, CNS-1239423, CCF-1255729, CNS-1319353, CNS-1319417, and CCF-137224, and by the Semiconductor Research Corporation Contract No. 2013-HJ-2405.

References

[1] SPECCPU2006 benchmark. www.spec.org.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention man-

agement on multicore systems. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[4] J. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *21st EUROMICRO Conf. on Real-Time Systems (ECRTS)*, Dublin, Ireland, July 2009.

[5] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Int'l Symp. on Microarchitecture (MICRO)*, pages 455–468, Orlando, FL, Dec. 2006.

[6] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 335–346, Pittsburgh, PA, Mar. 2010.

[7] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 25–38, Brasov, Romania, Sept. 2007.

[8] G. E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade - Second Edition*, pages 599–619. 2012.

[9] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[10] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 426–434, Las Vegas, NV, 2008.

[11] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, Aug. 2009.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 31–46, Hollywood, CA, 2012.

[13] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, CA, July 2010.

[14] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *44th Int'l Symp. on Microarchitecture (MICRO)*, Porto Alegre, Brazil, Dec. 2011.

[15] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 63–74, Beijing, China, June 2008.

[16] R. Salakhutdinov and A. Mnih. Bayesian probabilistic matrix factorization using Markov Chain Monte Carlo. In *25th Int'l Conf. on Machine Learning (ICML)*, pages 880–887, Helsinki, Finland, 2008.

[17] K. Shen. Request behavior variations. In *15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 103–116, Pittsburg, PA, Mar. 2010.

[18] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, Apr. 2004.

[19] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Second EuroSys Conf.*, pages 47–58, Lisbon, Portugal, Mar. 2007.

[20] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 283–294, San Jose, CA, June 2011.

[21] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conf.*, San Deigo, CA, June 2009.

[22] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *4th EuroSys Conf.*, pages 89–102, Nuremberg, Germany, Apr. 2009.

[23] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proc. 4th Intl Conf. Algorithmic Aspects in Information and Management, LNCS 5034*, pages 337–348. Springer, 2008.

Establishing a base of trust with performance counters for enterprise workloads

Andrzej Nowak, *CERN openlab and EPFL* Ahmad Yasin, *Intel* Avi Mendelson, *Technion* Willy Zwaenepoel, *EPFL*

Abstract

Understanding the performance of large, complex enterprise-class applications is an important, yet non-trivial task. Methods using hardware performance counters, such as profiling through event-based sampling, are often favored over instrumentation for analyzing such large codes, but rarely provide good accuracy at the instruction level.

This work evaluates the accuracy of multiple event-based sampling techniques and quantifies the impact of a range of improvements suggested in recent years. The evaluation is performed on instances of three modern CPU architectures, using designated kernels and full applications. We conclude that precisely distributed events considerably improve accuracy, with further improvements possible when using Last Branch Records. We also present practical recommendations for hardware architects, tool developers and performance engineers, aimed at improving the quality of results.

1. Introduction

Optimizing large codes is difficult. It requires dealing with millions of lines of code developed by many engineers, processing diverse data sets that run on complicated warehouse-scale systems [1]. Furthermore, it requires a deep understanding of the specific hardware architecture [3] or mandates use of recently published cycle-accounting methods suitable for out-of-order cores [27][36]. Modern processors feature counters that aim to assist users in understanding how well their application is performing. The hardware component in charge of gathering this information is usually called the Performance Monitoring Unit (PMU). The methodologies based on using these counters are well-established, especially in the HPC domain [2].

Many profilers [4][5][6] provide the means to narrow down the information gathered to select locations in the code that may cause an inefficiency. Methodologies that perform their analysis at basic-block granularity provide high source-level resolution, and are therefore the focus of this paper. Accurately obtaining basic block execution counts is a key problem facing the abovementioned profilers when analyzing enterprise-class, large-scale object-oriented workloads. These have challenging long-tail profiles with few hotspots where instrumentation is usually unsuitable [39].

This paper surveys Event Based Sampling (EBS) techniques and the parameters that influence measurement accuracy. We conduct our study on instances of modern enterprise processors. We develop a set of microbenchmarks, use a subset of the CPU2006 workloads, and use a large production workload from the CERN datacenter [7], running in deployments exceeding 300'000 cores. The contributions of this paper are:

- A first, to our knowledge, experimental evaluation of the accuracy of EBS techniques – serving as a necessary base for further work in this field.
- The conclusion that precise events considerably improve accuracy with little or no added cost, with further improvements resulting from the collection and analysis of Last Branch Records.
- Recommendations for hardware architects, tool developers and performance engineers working with EBS, aimed at improving the quality of results.

2. Motivation and background

2.1. The need for accurate basic-block profiles

Accurate and efficient basic block execution counts are important for a wide spectrum of use cases. Basic block graphs can rely on accurate basic block profiles. These can greatly improve the compiler's capability to make better decisions on inlining, while increasing code locality. Code level energy-efficiency monitors demand accuracy by using metrics such as Watts-per-instruction (WPI) [8][9]. Loop tripcounts are widely used for a variety of purposes [10][11][12], but are hard to obtain with pure EBS methods. In automatic or semi-automatic optimization of whole complex applications consisting of millions of lines of code, performance tuning must be driven by rather precise methods – e.g., basic block execution counts or precise function/method-granularity profiles.

2.2. Increasing processor complexity

Common tasks described in section 2.1 are made even more difficult by the ever-increasing complexity of modern processors. Out-of-order execution, superscalar pipelines, speculation and hardware prefetching increase performance, but complicate analysis [27]. At the same time, the core infrastructure of the PMUs has not progressed much, with only incremental updates between generations. Consequently, attributing samples to the exact program location that triggers a

performance event is becoming more and more difficult.

2.3. Growing code size and sophistication

The sheer number of lines of code (LOC), many of which are active, is steadily increasing across the application spectrum. An example study [14] focused on the SPEC CPU suite indicates the CPU2000 suite's codebase was below 1000 KLOC (thousand LOC) with no C++ code included, while CPU2006 reached over 3000 KLOC with only 25% of the benchmarks written in C++. Heavily object-oriented scientific toolkits, such as Geant4 [15, p. 4] and ROOT [16], are self-contained software packages that reach millions of LOC. Yasin et al. [13] demonstrate how abstractions incur significant performance overhead in data analytics and lead to code fragmentation with very small code-block sizes (typical ratios of instructions to branches taken around 6-12 with three lead JVMs).

2.4. Applicability of standard PMU methods

Traditional PMU-based methods as deployed in common profilers - such as perf [4], oprofile, VTune [5] or Code Analyst [6] - are rather designed for HPC and steady-state traditional workloads. They do not cope well with large object-oriented codes with highly fragmented profiles having thousands of entries and very few hotspots. A good tool has to consciously adjust the setup of the underlying generic hardware mechanisms for the best possible handling of the characteristics of a workload.

Our research indicates that more advanced methods, that can be particularly competitive and accurate, are infrequently used (e.g., PBA profiler [17]).

2.5. Related work

In recent years, the topic of PMU trust has been touched on by several publications. Works by Mytkowicz [18] and Weaver [3][19][20] analyze and identify CPU and OS effects that influence accuracy in *counting-mode* – some of which we also observe in our study of *sampling (i.e. EBS)*. Chen et al. have discussed some key sampling effects such as skid and shadow [21]. We go beyond these works and beyond Zaparanuks et al. [37] by focusing on the root causes of sampling inaccuracy, showing where to look for control over multiple events and how to identify optimal configurations for enterprise-like workloads.

3. Comments on sampling

3.1. Event Based Sampling

EBS exploits the capability of the PMU to count pre-defined hardware events and to generate an interrupt when the counted event is observed N specified times. N is called the sampling period, and the interrupts are called Performance Monitoring Interrupts (PMI). Instruction pointer locations are sampled on PMIs, and their distribution is used to generate profiles.

According to Chen et al. [21] and Levinthal [23], errors in the distribution of samples are the result of three major factors: (1) *synchronization* of monitored code with the sampling period, (2) the *sampling skid effect*, when the address reported by the hardware sample does not necessarily match the address of the instruction causing counter overflow, and (3) the *sampling shadow effect,* when instructions in the shadow of a long latency instruction get low sample counts.

When seeking more accurate profile information about basic block execution counts, tools average samples across all instructions in the same block. While helpful, such mitigation is still insufficient for short blocks, such as e.g., jump tables, as samples have higher chances of getting attributed to adjacent blocks.

3.2. Last Branch Records

Some processors feature a branch recording facility, such as Last Branch Record (LBR) in the x86 architecture, which records the addresses of the most recently executed branches. These facilities can be used for basic block execution counts, as suggested by Levinthal [23] and implemented in the emerging Gooda [26][36] and PBA [17] tools. These tools and their relevant heuristics are scarcely documented and not yet widely adopted in the community. Consequently, we implement our own version of LBR analysis, as documented below.

An LBR facility has a number of stacked entries, which represent source-target pairs $<S_i, T_i>$ of branches executed by the processor. When sampling on the *Taken Branches* event, branches between a target T_i and the next source S_{i+1} in the stack are not taken. Thus, all basic blocks between T_i and S_{i+1} are executed exactly once.

3.3. Profiling accuracy

In order to estimate the accuracy of PMU-based sampling, we cross-reference results with instrumentation-based basic block counts obtained through Pin [25] ("REF").

$$\text{Accuracy Error (x)} = \frac{\sum_{i \in BB} |(\text{BB}_\text{x}[\text{i}] - \text{BB}_\text{REF}[\text{i}])|}{\text{net_instruction_count}}$$

For a given sampling method *x*, our accuracy error is defined as the sum of all deviations between the *x* method and the REF method, of the number of instructions executed in each basic block. This error is normalized to the total number of instructions executed. Ideally, we would like the accuracy error as close as possible to zero.

3.4. Evaluation of existing methods

Out of the combinations of profiling approaches assessed and discussed in Section 4 (also see Appendix), we choose three particular groups of methods that exemplify how sampling can be used:

- *Classic Sampling* is a representative, widely employed method of the first group. It uses an imprecise counter, where the sampling period is fixed, even and not randomized. No filtering is applied. This is the default in mainline tools such as perf [4], where an architectural event is typically set to capture a sample every ~1 millisecond.
- *Precise Sampling* represents the advanced group of methods. It uses a general-purpose counter with a precise-sampling mechanism featured by the PMU, where the period is a prime number, preferably randomized.
- *LBR Sampling* is performed when using a retired Taken Branches event[1]. The full contents of each collected LBR stack serve as the basis for basic block execution counts, while the address that comes with the PMI is ignored.

4. Experimental environment

4.1. Hardware and software setup

We evaluate out-of-order processors from the x86 family, as implemented by Intel and AMD. From the AMD side, we choose a representative of the "Magny-Cours" family, the 12-core 6164 HE. Software support for this family was the most robust at the time of writing. On the Intel side, we choose the Xeon X5650, as the representative of the 1st Core i7 family, a.k.a. Westmere, and a Xeon E3-1265L, representing the 3rd generation Core family, a.k.a. Ivy Bridge. Again, stable software support and existing experience played a role in our choice. Frequency scaling and "turbo mode" are disabled.

To obtain PMU samples we use a modified version of the perf utility in Linux 3.6.6, on RHEL6 compatible systems. Perf has a very fluid codebase, which impacts measurement overheads much more than hardware does [38], and this particular version is used throughout the whole study. Non-essential services/daemons are disabled.

Each of our kernels, emphasizing specific difficulties leading to reduced accuracy, is measured five times.

4.2. PMU configurations and events

The methods described in this section are presented in more detail in Table 3 in the Appendix.

Magny-Cours does not feature LBRs, nor a fixed architectural counter. The latter could be an issue with the version of perf available at time of writing. The standard event of choice was RETIRED_INSTRUCTIONS. Instruction Based Sampling (IBS) is the precise mechanism offered by AMD. We program the PMU to sample with prime and non-prime periods. Due to perf limitations, software-based period randomization was unavailable, but the hardware randomizes the 4 least significant bits.

On Westmere, we choose to work with the fixed instructions retired counter and with the programmable instructions retired event supplemented with Precise Event Based Sampling (PEBS). LBRs are sampled with the BR_INST_EXEC:TAKEN event, with PEBS disabled.

On Ivy Bridge, we use the instructions retired event on the fixed counter (INST_RETIRED:ANY) as well as the recently added Precisely Distributed event (INST_RETIRED:PREC_DIST, a.k.a. PDIR, [24]). LBRs are sampled using BR_INST_RETIRED:NEAR_TAKEN, with PEBS disabled.

4.3. Workloads

4.3.1. *Latency-biased kernel*

The latency-biased kernel is the simplest form of emulation of workloads with basic blocks with non-uniform execution times. Here, a loop executes a relatively costly calculation when a certain condition is true:

```
while (n--) ((n%2) ? x /= y : x += y);
```

Such code occurs in practice, for example, when a pre-computed value is returned in the standard case, and re-computed otherwise. Typically, the PMU would bias samples towards the long latency instruction, thus distorting overall results [21].

4.3.2. *Call chain kernel*

This kernel is a simple 10-deep call chain enveloped by a loop. Since the functions do equal work, they are expected to produce equal numbers of samples.

This example serves as a vivid illustration of potential sampling bias on call chains - such as those commonly seen in object-oriented programming with frequently called short methods.

4.3.3. *G4Box test*

The G4Box micro-benchmark, written in C++, executes only two functions, with an even work split. It could be thought of as a heavier version of our Latency Biased kernel. The length of the main function depends on the input data.

This kernel is particularly difficult for hardware sampling, since it contains a chain of tests and branches that generates short basic blocks – a good case for LBR analysis.

[1] BR_INST_RETIRED.NEAR_TAKEN on Ivy Bridge

4.3.4. *Geant4 test40*

Test40 is a kernelized doppelganger of large Geant4 applications. In this test, an electron travels through a detector with a very simple geometry, triggering physics processes on its way. The signature workload is therefore a collection of small, fragmented methods, conditionally executed depending on where the particle is and what matter it interacts with.

4.3.5. *Applications*

First, we select a non-HPC subset of both INT and FP benchmarks from the SPEC2006 CPU suite: 429.mcf, 453.povray, 471.omnetpp and 483.xalancbmk. This subset, written in C/C++, has (to some extent) the characteristics of enterprise workloads [39], as described by Wong in [28] and [29]. Some enterprise vendors commonly use these benchmarks as proxies for real applications.

Second, the FullCMS application is based on parts of Geant4 and is designed to simulate complex physics events taking place in one of CERN's Large Hadron Collider particle detectors. It is similar to the enterprise class of workloads in the sense that it executes similar fragmented operations, albeit using floating point rather than integers. This production-grade workload has successfully served as an enterprise "proxy" in the past and runs on ~300'000 cores.

5. Results

5.1. Kernel results

The results in Table 1 present accuracy errors as defined in Section 3.3 of the various sampling methods defined in Section 4.2. Overall results show that:

- LBR-based methods are highly beneficial, significantly reducing errors by up to 18x (3-6x on average).
- Progressive improvements from randomization and period adjustment are observed as better techniques are applied.
- The precisely distributed event (PDIR) significantly improves results across all kernels and especially for Latency Biased.
- AMD systems are consistently burdened with high error rates, worsening when built-in hardware randomization was used.

On Latency Biased, we observe improvements introduced by the Ivy Bridge precisely distributed PDIR event. These accuracy boosts, on the other hand, are not observed on the Westmere microarchitecture, where that event is not featured.

Results for the Callchain kernel show how applying prime as well as randomized periods gradually improves accuracy as we move to the right with improvements. In the Ivy Bridge case, combining the LBR-based IP+1 fix with PDIR (see Appendix) gives the best results. While there is no definitive indication of the reason, it would appear that out-of-order clustering of uops, which causes uops to be retired in bursts, is responsible for this characteristic.

On testG4Box, medium error rates are reduced when LBR is employed. This is because this test case is dominated by very short basic blocks, which challenge sampling in general. The LBR-based technique addresses this issue by averaging samples across the last 15 uninterrupted basic block segments, which extends the effective number of instructions that the sample corresponds to.

On test40, we see that Westmere suffers from distribution problems linked to the sampling event, which disappear on Ivy Bridge. Employing LBRs alleviates these issues on both platforms.

5.2. Application results

Table 2 shows error averages for applications. The general observations are as follows:

Table 1: Sampling methods used on kernels and their errors according to our accuracy metric (lower is better).

		INST RETIRED default	Precise event or uops					Taken Branches Retired
		Even period	Even period		Prime period			
		Fixed	Fixed	Rand.	Fixed	Rand.	Rand. Fix IP+1 (LBR)	LBR
AMD	latencybias	0.84	N/A	0.88	N/A	0.84	N/A	N/A
	callchain	0.61		0.80		0.73		
	testG4Box	0.28		0.29		0.35		
	test40	0.60		0.91		0.85		
WSM	latencybias	1.57	1.57	1.57	1.59	1.59	1.57	0.09
	callchain	0.48	0.53	0.41	0.68	0.38	0.37	0.21
	testG4Box	0.32	0.36	0.29	0.75	0.30	0.42	0.05
	test40	0.58	0.57	0.57	0.55	0.55	0.54	0.16
IVB	latencybias	1.21	0.28	0.19	0.22	0.19	0.23	0.07
	callchain	0.59	0.48	0.19	0.24	0.15	0.10	0.12
	testG4Box	0.18	0.17	0.16	0.15	0.17	0.18	0.05
	test40	0.60	0.20	0.19	0.14	0.14	0.13	0.11

- Randomization has little to no impact on full applications. Full applications often do not have specific loop trip-counts to synchronize with sampling periods, as tight kernels can have.
- Using a counter with precise distribution and applying an LBR address fix provides good results.
- Pure LBR basic block counts further improve accuracy (except for FullCMS). Overall improvement is 4-5x over the classic case and 1-10x over the precise case.

The classic method registers high overall error rates, much improved with the precise event on IVB. Also, the LBR method is noticeably better than precise sampling, especially so in the case of mcf.

As a side note, our FullCMS experiments showed that already choosing an EBS method on a counter with precise distribution and applying an LBR-based IP offset correction (but not full LBR sampling) improves average per basic block accuracy by 5x over the leftmost classic case. Pure LBR-based results, however, do not bring improvement over the precise method in this case, since the workload has similar characteristics to the callchain kernel.

LBR, while of great benefit to performance monitoring activities, is not entirely free of issues. None of the methods produces the top 10 functions from the FullCMS profile in the right order.

Table 2: Errors per machine/app (lower is better).

		INST RETIRED Default	Precise; Fixed period	Precise; Prime period	LBR
AMD	mcf	0.66	1.225	1.212	N/A
	povray	0.48	0.545	0.557	
	omnetpp	0.80	1.018	1.013	
	xalancbmk	0.53	0.606	0.606	
	FullCMS	0.53	0.611	0.606	
WSM	mcf	0.44	0.432	0.446	0.194
	povray	0.50	0.511	0.514	0.177
	omnetpp	0.67	0.682	0.679	0.302
	xalancbmk	0.46	0.485	0.484	0.087
	FullCMS	0.55	0.547	0.547	0.177
IVB	mcf	0.50	0.306	0.305	0.034
	povray	0.49	0.164	0.161	0.123
	omnetpp	0.65	0.246	0.249	0.116
	xalancbmk	0.56	0.341	0.340	0.112
	FullCMS	0.55	0.179	0.177	0.120

6. Recommendations

6.1. Recommendations for tool developers

First, we recommend that tool developers make distinctions between similar performance events. Users of perf, PAPI or OProfile are presented with opaque tools that obscure events and make adjustments (such as those of the sampling period) hard. For example, in the case of standard perf, a recompilation and installation of an extra library (libpfm4) is required to obtain reasonable access to hardware performance events.

Second, sampling periods need to be chosen with a dose of care. Prime number periods reduce the risk of synchronizing with the workload, and randomization further improves results on artificial kernels, but neither produced noticeable improvements on our large benchmarks (unlike what is reported in [21]). As of today, neither perf nor major commercial tools support fixed period randomization.

Third, the LBR-based methods we evaluated allow for enhanced degrees of accuracy, which – with some post-processing – could serve as input to PGO, code coverage or other sensitive optimization techniques. Only a couple tools (PBA [17] and GOODA [26][36]) use LBRs to obtain basic block execution counts, and their documentation only sparsely describes the methods employed.

6.2. Recommendations for PMU hardware designers

The IP+1 inaccuracy fix in sample addresses based on an LBR sourced address (not the full LBR) can lead to good improvements, especially for branchy code with a high rate of calls or taken branches. Implementing such functionality in hardware would not only remove the workaround burden in drivers, but also avoid collisions on LBRs – a valuable single resource – with other filtered collections such as call-stack mode.

A precise instruction event in AMD's IBS is missing, which led us to use precise uops instead.

6.3. Recommendations for application optimizers

We have shown that the methods used to obtain performance data matter and influence potential conclusions. Overall, we recommend to sample on a modern platform with support for precise distributed events, while using a prime period. Kernel-like code additionally benefits from more frequent sampling periods and period randomization. For ultimate sampling performance, we recommend liaising with tool developers to employ LBR-based methods that maximize accuracy.

7. Conclusions and Summary

In this short survey of a somewhat underexplored area, we quantify the level to which choices related to performance monitoring methods influence results. The precise events introduced in the Ivy Bridge microarchitecture considerably improve accuracy with little or no added cost. Period randomization shows improvements on kernels, but not on complete applications. LBR-based methods improve results even more over precise counters, and work especially well on Westmere machines.

REFERENCES

[1] L. A. Barroso and U. Holzle, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[2] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Dept. of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[3] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008, pp. 141–150.

[4] A. Carvalho de Melo, "The New Linux 'perf' tools." Linux Kongress, 2010.

[5] Intel Corporation, "Intel VTune Amplifier XE 2013," 2012. [Online]. Available: http://software.intel.com/en-us/intel-vtune-amplifier-xe. [Accessed: 22-Nov-2012].

[6] P. J. Drongowski, A. M. D. C. A. Team, and B. D. Center, "An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer," *Advanced Micro Devices, Inc*, 2008.

[7] S. Banerjee, "Readiness of CMS simulation towards LHC startup," *Journal of Physics: Conference Series*, vol. 119, no. 3, p. 032006, Jul. 2008.

[8] S. Schubert, D. Kostic, W. Zwaenepoel, and K. Shin, "Profiling Software for Energy Consumption," in *Proceedings of the IEEE International Conference on Green Computing and Communications (GreenCom)*, 2012.

[9] M. D. DeVuyst, "Efficient Use of Execution Resources in Multicore Processor Architectures," University of California, San Diego, 2011.

[10] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, New York, NY, USA, 2009, pp. 225–236.

[11] T. Sherwood and B. Calder, "Loop Termination Prediction," in *High Performance Computing*, M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, Eds. Springer Berlin Heidelberg, 2000, pp. 73–87.

[12] K. Muthukumar and G. Doshi, "Software Pipelining of Nested Loops," in *Compiler Construction*, R. Wilhelm, Ed. Springer Berlin Heidelberg, 2001, pp. 165–181.

[13] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive Analysis of the Data Analytics Workload in CloudSuite," presented at the IISWC'14, 2014.

[14] J. L. Henning, "SPEC CPU suite growth: an historical perspective," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 65–68, Mar. 2007.

[15] J. Apostolakis, "Geant4—a simulation toolkit," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 506, no. 3, pp. 250–303, Jul. 2003.

[16] R. Brun and F. Rademakers, "ROOT — An object oriented data analysis framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1–2, pp. 81–86, Apr. 1997.

[17] "Intel Performance Bottleneck Analyzer." Intel Corporation, 2011.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!," in *ACM Sigplan Notices*, 2009, vol. 44, pp. 265–276.

[19] V. Weaver, "Can Hardware Performance Counters Produce Expected, Deterministic Results?," presented at the 3rd Workshop on Functionality of Hardware Performance Monitoring, 2010.

[20] V. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[21] D. Chen, N. Vachharajani, R. Hundt, S. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for FDO compilation," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, New York, NY, USA, 2010, pp. 42–52.

[22] A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. J. Reddi, and D. A. Connor, "Analysis of path profiling information generated with performance monitoring hardware," in *9th Annual Workshop on Interaction between Compilers and Computer Architectures, 2005. INTERACT-9*, 2005, pp. 34 – 43.

[23] D. Levinthal, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors." Intel Corporation, 2009.

[24] "Intel® 64 and IA-32 Architectures Software Developer Manuals," *Intel*. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. [Accessed: 26-Sep-2014].

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized

program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2005, pp. 190–200.

[26] Google, *Gooda - a pmu event analysis package (http://code.google.com/p/gooda/)*. 2012.

[27] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," presented at the 2014 IEEE International Symposium Performance Analysis of Systems and Software (ISPASS), 2014.

[28] M. Wong, "C++ benchmarks in SPEC CPU2006," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, p. 77, Mar. 2007.

[29] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[30] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations," *IEEE Transactions on Computers*, vol. PP, no. 99, p. 1, 2011.

[31] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, pp. 1319–1360, Jul. 1994.

[32] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "We have it easy, but do we have it right?," in *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, 2008, pp. 1 –7.

[33] K. Walcott-Justice, J. Mars, and M. L. Soffa, "THeME: a system for testing by hardware monitoring events," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, New York, NY, USA, 2012, pp. 12–22.

[34] M. L. Soffa, K. R. Walcott, and J. Mars, "Exploiting hardware advances for software testing and debugging: NIER track," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 888 –891.

[35] S. Narayanasamy, T. Sherwood, S. Sair, B. Calder, and G. Varghese, "Catching accurate profiles in hardware," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*, 2003, pp. 269 – 280.

[36] A. Nowak, D. Levinthal and W. Zwaenepoel, "Hierarchical Cycle Accounting – a new method for application performance tuning" in *2015 IEEE International Symposium on Performance Analysis of Systems and Software,* 2015.

[37] D. Zaparanuks, M. Jovic and M. Hauswirth, "Accuracy of performance counter measurements" in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[38] G. Bitzes, A. Nowak, "The overhead of profiling using PMU hardware counters", *CERN openlab report,* 2014.

[39] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei and D. Brooks, "Profiling a Warehouse-Scale Computer" in *International Symposium on Computer Architecture (ISCA)*, Jun 2015.

APPENDIX:

Table 3: An overview of reviewed sampling methods

	Method	Parameters			Comments	Drawbacks
		Period size (Example value)	**Period Rando-mization**	**Event** (Intel nomenclature)		
Classic method	Default	Round (2'000'000)	No	INST_RETIRED.ANY (non-precise)	Used by default in many tools. Uses a fixed-function counter to free up general counters.	The period is fixed and round which increases the risk of synchronization, the hardware event is imprecise
Precise methods	Precise event	Round (2'000'000)	No	INST_RETIRED.ALL (precise)	Uses a precise mechanism to capture the event location (IP+1)	The distribution of samples is not guaranteed
	Precise event with randomization	Round (2'000'000)	Yes		A randomized sampling period to avoid synchronization risk	As above
	Precise event with prime period	Prime number (2'000'003)	No		The introduction of prime numbers reduces resonance which leads to improved accuracy	Lack of randomization and overall low accuracy in some cases alike the Latency-Biased kernel
	Precise event with randomized prime period	Prime number (2'000'003)	Yes		Randomization applied on the prime period further improves accuracy	Still overall low accuracy in some cases
	precise event with distribution fix plus IP+1 offset fix	Prime number (2'000'003)	Yes/No	INST_RETIRED. PREC_DIST (precisely distributed)	To remedy skid, the top address from the LBR backtrace is used to determine which basic block the trigger occurred in; thus fixing IP+1 and enhancing accuracy.	Good for large basic blocks. Some inaccuracies for small ones
LBR method	Last Branch Record	N/A	N/A	BR_INST_RETIRED. NEAR_TAKEN	Full LBR-based basic block execution count accounting with manageable errors per basic block	Errors can still reach 30-50% of basic block execution count (for some basic blocks). Overhead (in collection and post-processing)

Utilizing the IOMMU Scalably

Omer Peleg Adam Morrison
Technion
{omer,mad}@cs.technion.ac.il

Benjamin Serebrin
Google
serebrin@google.com

Dan Tsafrir
Technion
dan@cs.technion.ac.il

Abstract

IOMMUs provided by modern hardware allow the OS to enforce memory protection controls on the DMA operations of its I/O devices. An IOMMU translation management design must *scalably* handle frequent concurrent updates of IOMMU translations made by multiple cores, which occur in high throughput I/O workloads such as multi-Gb/s networking. Today, however, OSes experience performance meltdowns when using the IOMMU in such workloads.

This paper explores *scalable IOMMU management designs* and addresses the two main bottlenecks we find in current OSes: (1) assignment of I/O virtual addresses (IOVAs), and (2) management of the IOMMU's TLB.

We propose three approaches for scalable IOVA assignment: (1) dynamic identity mappings, which eschew IOVA allocation altogether, (2) allocating IOVAs using the kernel's `kmalloc`, and (3) per-core caching of IOVAs allocated by a globally-locked IOVA allocator. We further describe a scalable IOMMU TLB management scheme that is compatible with all these approaches.

Evaluation of our designs under Linux shows that (1) they achieve 88.5%–100% of the performance obtained *without* an IOMMU, (2) they achieve similar latency to that obtained *without* an IOMMU, (3) scalable IOVA allocation and dynamic identity mappings perform comparably, and (4) `kmalloc` provides a simple solution with high performance, but can suffer from unbounded page table blowup.

1 Introduction

Modern hardware provides an I/O memory management unit (IOMMU) [2, 6, 24, 27] that mediates direct memory accesses (DMAs) by I/O devices in the same way that a processor's MMU mediates memory accesses by instructions. The IOMMU interprets the target address of a DMA as an I/O virtual address (IOVA) [32] and attempts to translate it to a physical address, blocking the DMA if no translation (installed by the OS) exists.

IOMMUs thus enable the OS to restrict a device's DMAs to specific physical memory locations, and thereby protect the system from errant devices [18, 29], malicious devices [8, 13, 46], and buggy drivers [7, 15, 22, 31, 41, 44], which may misconfigure a device to overwrite system memory. This *intra-OS* protection [45] is recommended by hardware vendors [23, 29] and implemented in existing OSes [5, 12, 25, 36]. OSes can employ intra-OS protection both in non-virtual setups, having direct access to the physical IOMMU, and in virtual setups, by exposing the IOMMU to the VM through hardware-supported nested IOMMU translation [2, 27], by paravirtualization [9, 31, 40, 45], or by full emulation of the IOMMU interface [3].

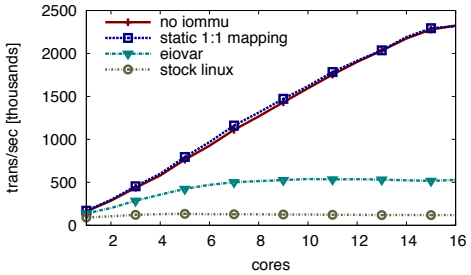


Figure 1. Parallel `netperf` throughput (Linux): Performance meltdown due to dynamic IOMMU mapping updates.

Intra-OS protection requires each DMA operation to be translated with a transient IOMMU mapping [12] dedicated to the DMA, which is destroyed once it completes so that the device cannot access the memory further [29, 37]. For example, a network interface card (NIC) driver maps the buffers it inserts into the NIC's receive (RX) rings to receive packets. Once a packet arrives (via DMA), the driver unmaps the packet's buffer.

These transient *dynamic* IOMMU mappings pose a performance challenge for driving high-throughput I/O workloads. Such workloads require dynamic mappings to be created and destroyed millions of times a second by *multiple cores concurrently*, since a single core often cannot sustain high enough throughput [30]. This paper specifically targets *multi-Gb/s networking*—NICs providing 10–40 Gb/s (and soon 100 Gb/s)—as a representative demanding case.

Current OSes melt down under load when the IOMMU is enabled in such a workload. Figure 1 demonstrates the problem on Linux. It shows the combined

throughput of 270 `netperf` instances running a request-response workload (described in § 6) on a 16-core x86 server with a 10 Gb/s NIC, which we use as a running example throughout the paper. Mediating DMAs by the IOMMU imposes only negligible overhead by itself, as evidenced by the throughput obtained when the IOMMU is configured with *static* identity mappings that pass DMA requests through the IOMMU unchanged. In contrast, when IOMMU management is enabled, throughput drops significantly and ceases to increase with the number of cores. This occurs with both Linux's stock IOMMU subsystem or the recent EiovaR [35] optimization, which targets the *sequential* performance of Linux's IOMMU subsystem (see § 3.1). Other OSes suffer from similar scalability problems (§ 3).

This paper thus considers the **IOMMU management** problem of *designing a subsystem supporting high-throughput concurrent updates of IOMMU mappings*. We analyze the bottlenecks in current IOMMU management designs (§ 3) and explore the trade-offs in the design space of their solutions (§§ 4–5). Our designs address the two main bottlenecks we find in current OSes: IOVA assignment and IOTLB invalidation.

IOVA assignment Creating an IOMMU mapping for a physical memory location requires the OS to designate a range of IOVAs that will map to the physical location. The driver will later configure the device to DMA to/from the IOVAs. OSes presently use a dedicated, centralized (lock-protected) IOVA allocator that becomes a bottleneck when accessed concurrently.

We propose three designs for *scalable* IOVA assignment (§ 4), listed in decreasingly radical order: First, *dynamic identity mapping* eliminates IOVA allocation altogether by using a buffer's physical address for its IOVA. Consequently, however, maintaining the IOMMU page tables requires more synchronization than in the other designs. Second, *IOVA-kmalloc* eliminates only the specialized IOVA allocator by allocating IOVAs using the kernel's optimized `kmalloc` subsystem. This simple and efficient design is based on the observation that we can treat the addresses that `kmalloc` returns as IOVA page numbers. Finally, *per-core IOVA caching* keeps the IOVA allocator, but prevents it from becoming a bottleneck by using *magazines* [11] to implement per-core caches of free IOVAs, thereby satisfying allocations without accessing the IOVA allocator.

IOTLB invalidation Destroying an IOMMU mapping requires invalidating relevant entries in the IOTLB, a TLB that caches IOMMU mappings. The Linux IOMMU subsystem amortizes the invalidation cost by batching multiple invalidation requests and then performing a single global invalidation of the IOTLB instead. The batching data structure is lock-protected and quickly becomes a bottleneck. We design a compatible scalable batching data structure as a replacement (§ 5).

Design space exploration We evaluate the performance, page table memory consumption and implementation complexity of our designs (§ 6). We find that (1) our designs achieve 88.5%–100% of the throughput obtained *without* an IOMMU, (2) our designs achieve similar latency to that obtained *without* an IOMMU, (3) the savings dynamic identity mapping obtains from not allocating IOVAs are negated by its more expensive IOMMU page table management, making it perform comparably to scalable IOVA allocation, and (4) IOVA-kmalloc provides a simple solution with high performance, but it can suffer from unbounded page table blowup if empty page tables are not reclaimed (as in Linux).

Contributions This paper makes four contributions:

- Identifying IOVA allocation and IOTLB invalidation as the bottlenecks in the IOMMU management subsystems of current OSes.
- Three designs for scalable IOVA allocation: (1) dynamic identity mappings, (2) IOVA-kmalloc, and (3) per-core caching of IOVAs, as well as a scalable IOTLB invalidation scheme.
- Evaluation of the new and existing designs on several high throughput I/O workloads.
- Design space exploration: we compare the performance, page table memory consumption and implementation complexity of the proposed designs.

2 Background: IOMMUs

The IOMMU mediates accesses to main memory by I/O devices, much like the MMU mediates the memory accesses performed by instructions. IOMMUs impose a translation process on each device DMA. The IOMMU interprets the target address of the DMA as an *I/O virtual address* (IOVA) [32], and attempts to translate it to a physical address using per-device *address translations* (or *mappings*) previously installed by the OS. If a translation exists, the DMA is routed to the correct physical address; otherwise, it is blocked.

In the following, we provide a high-level description of Intel's x86 IOMMU operation [27]. Other architectures are conceptually similar [2, 6, 24].

IOMMU translations The OS maintains a *page table* hierarchy for each device, implemented as a 4-level radix tree (as with MMUs). Each radix tree node is a 4 KB page. An inner node (*page directory*) contains 512 pointers (*page directory entries*, or PDEs) to child radix-tree nodes. A leaf node (*page table*) contains 512 pointers (*page table entries*, or PTEs) to physical addresses. PTEs also encode the type of access rights provided through this translation, i.e., read, write or both.

The virtual I/O address space is 48-bit addressable.

The 36 most significant bits of an IOVA are its *page frame number* (PFN), which the IOMMU uses (when it receives a DMA request) to walk the radix tree (9 bits per level) and look up the physical address and access rights associated with the IOVA. If no translation exists, the IOMMU blocks the DMA and interrupts the processor. Unlike the analogous case in virtual memory, this is not a *page fault* that lets the OS install a new mapping and transparently resume operation of the faulting access. Instead, the DMA is simply dropped. The I/O device observes this and may not be able to recover. [1]

IOTLB translation caching The IOMMU maintains an IOTLB that caches IOVA translations. If the OS modifies a translation, it must *invalidate* (or *flush*) any TLB entries associated with the translation. The IOMMU supports individual invalidations as well as *global* ones, which flush all cached translations. The OS requests IOTLB invalidations using the IOMMU's *invalidation queue*, a cyclic buffer in memory into which the OS adds invalidation requests and the IOMMU processes them asynchronously. The OS can request to be notified when an invalidation has been processed.

2.1 IOMMU protection

IOMMUs can be used to provide *inter-* and *intra-OS* protection [3, 43, 45, 47]. IOMMUs are used for inter-OS protection in virtualized setups, when the host assigns a device for the exclusive use of some guest. The host creates a *static* IOMMU translation [45] that maps guest physical pages to the host physical pages backing them, allowing the guest VM to directly program device DMAs. This mode of operation does not stress the IOMMU management code and is not the focus of this work.

We focus on intra-OS protection, in which the OS uses the IOMMU to restrict a device's DMAs to specific physical memory locations. This protects the system from errant devices [18, 29], malicious devices [8, 13, 46], and buggy drivers [7, 15, 22, 31, 41, 44].

Intra-OS protection is implemented via the DMA API [12, 32, 37] that a device driver uses when programming the DMAs. To program a device DMA to a physical buffer, the driver must pass the buffer to the DMA API's *map* operation. The map operation responds with a *DMA address*, and it is the DMA address that the driver must program the device to access.

Internally, the map operation (1) allocates an IOVA range the same size as the buffer, (2) maps the IOVA range to the buffer in the IOMMU, and (3) returns the IOVA to the driver. Once the DMA completes, the driver must *unmap* the DMA address, at which point the mapping is destroyed and the IOVA range deallocated.

High throughput I/O workloads can create and destroy such *dynamic* IOMMU mappings [12] millions of times a second, on multiple cores concurrently, and thereby put severe pressure on the IOMMU management subsystem implementing the DMA API.

3 Performance Analysis of Present Designs

Here we analyze the performance of current IOMMU management designs under I/O workloads with high throughput and concurrency. We use Linux/Intel-x86 as a representative study vehicle; other OSes have similar designs (§ 3.4). Our test workload is a highly parallel RR benchmark, in which a `netperf` [28] server is handling 270 concurrent TCP RR requests arriving on a 10 Gb/s NIC. § 6 fully details the workload and test setup.

To analyze the overhead created by IOMMU management (shown in Figure 1), we break down the execution time of the parallel RR workload on 16 cores (maximum concurrency on our system) into the times spent on the subtasks required to create and destroy IOMMU mappings. Figure 2 shows this breakdown. For comparison, the last 3 bars show the breakdown of our scalable designs (§§ 4–5).

We note that this parallel workload provokes pathological behavior of the stock Linux IOVA allocator. This behavior, which does not exist in other OSes, causes IOVA allocation to hog $\approx 60\%$ of the execution time. The recent EiovaR optimization [35] addresses this issue, and we therefore use Linux with EiovaR as our baseline. We discuss this further in § 3.1.

In the following, we analyze each IOMMU management subtask and its associated overhead in the Linux design: IOVA allocation (§ 3.1), IOTLB invalidations (§ 3.2), and IOMMU page table management (§ 3.3).

3.1 Linux IOVA Allocation

In Linux, each device is associated with an IOVA allocator that is protected by a coarse-grained lock. Each IOVA allocation and deallocation for the device acquires its allocator's lock, which thus becomes a sequential bottleneck for frequent concurrent IOVA allocate/deallocate operations. Figure 2 shows that IOVA allocation lock acquisition time in the baseline accounts for 31.9% of the cycles. In fact, this is only because IOVA allocations are throttled by a *different* bottleneck, IOTLB invalidations (§ 3.2). Once we address the invalidations bottleneck, the IOVA allocation bottleneck becomes much more severe, accounting for nearly 70% of the cycles.

This kind of design—acquiring a global lock for each operation—would turn IOVA allocation into a sequential bottleneck no matter which allocation algorithm is used

[1] I/O page fault standardization exists, but since it requires support from the device, it is not widely implemented or compatible with legacy devices [2, 38, 39].

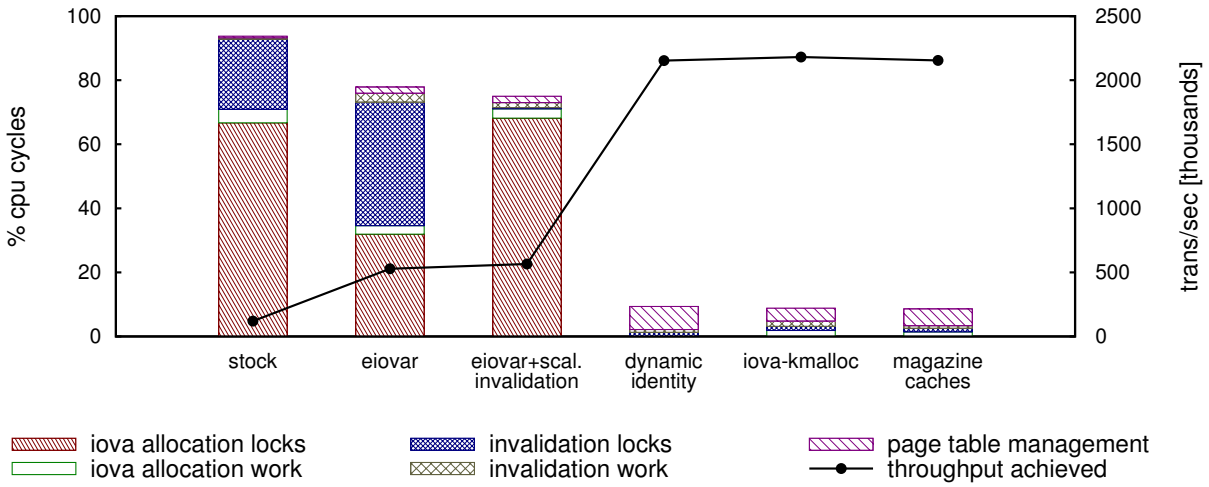


Figure 2. Throughput and cycle breakdown of time spent in IOMMU management on a 16-core parallel `netperf` RR workload. Stock Linux is shown for reference. Our baseline is Linux with **EiovaR** [35], which addresses a performance pathology in the stock Linux IOVA allocation algorithm (see § 3.1). In this baseline, the IOTLB invalidation bottleneck masks the IOVA allocation bottleneck, as evidenced by the third bar, which shows the breakdown after applying scalable IOTLB invalidations (§ 5) to the EiovaR baseline. Our designs are represented by the last three bars, nearly eliminating lock overhead.

once the lock is acquired. We nevertheless discuss the Linux IOVA allocation algorithm itself, since it has implications for IOMMU page table management.

The IOVA allocator packs allocated IOVAs as tightly as possible towards the end of the virtual I/O address space. This minimizes the number of page tables required to map allocated IOVAs—an important feature, because Linux rarely reclaims a physical page that gets used as an IOMMU page table (§ 3.3).

To achieve this, the IOVA allocator uses a red-black tree that holds pairwise-disjoint *ranges* of allocated virtual I/O page numbers. This allows a new IOVA range to be allocated by scanning the virtual I/O address space from highest range to lowest range (with a right-to-left traversal of the tree) until finding an unallocated gap that can hold the desired range. Linux attempts to minimize such costly linear traversals through a heuristic in which the scan starts from some previously cached tree node. This often finds a desired gap in constant time [35]. [2]

Unfortunately, the IOVA allocation patterns occurring with modern NICs can cause the heuristic to fail, resulting in frequent long linear traversals during IOVA allocations [35]. The EiovaR optimization avoids this problem by adding a cache of recently freed IOVA ranges that can satisfy most allocations without accessing the tree [35]. IOVA allocation time in Linux/EiovaR is thus comparable, if not superior, to other OSes. However, IOVA allocation remains a sequential bottleneck with EiovaR as well (Figure 2), since the EiovaR cache is accessed under the IOVA allocator lock.

[2] We refer the reader to [35] for the exact details of the heuristic, which are irrelevant for our purpose.

3.2 Linux IOTLB Invalidation

Destroying an IOMMU mapping requires invalidating the IOTLB entry caching the mapping, both for correctness and for security. For correctness, if the unmapped IOVA gets subsequently remapped to a different physical address, the IOMMU will keep using the old translation and misdirect any DMAs to this IOVA. Security-wise, destroying a mapping indicates the device should no longer have access to the associated physical memory. If the translation remains present in the IOTLB, the device can still access the memory.

Unfortunately, waiting for the invalidation to complete prior to returning control from IOVA unmapping code is prohibitively expensive (§ 6). In addition to the added latency of waiting for the invalidation to complete, issuing the invalidation command requires writing to the IOMMU invalidation queue—an operation that must be serialized and thus quickly becomes a bottleneck.

As a result, Linux does not implement this *strict* invalidation policy by default. Instead, it implements a *deferred* invalidation policy that amortizes the cost of IOTLB invalidation across multiple unmappings. Here, an unmap operation buffers an invalidation request for its IOVA in a global *flush queue* data structure and returns without waiting for the invalidation to complete. Periodically (every 10 ms) or after batching 250 invalidation requests, Linux performs a single global IOTLB invalidation that empties the entire IOTLB (possibly flushing valid entries as well). Once the global invalidation com-

pletes, the IOVA allocator is invoked to deallocate each IOVA range buffered in the flush queue.

Thus, deferred invalidation maintains correctness, but trades off some security—creating a window of time in which a device can access unmapped IOVAs—in exchange for performance. In practice, unmapped physical pages rarely get reused immediately upon returning from the unmap function. For example, a driver may unmap multiple pages—possibly triggering a global invalidation—before returning control to the system. Thus, deferred invalidation appears to be a pragmatic trade-off, and other OSes use similar mechanisms (§ 3.4).

While deferred invalidation amortizes the latency of waiting for invalidations to complete, the flush queue is protected by a single (per IOMMU) *invalidation lock*. As with the IOVA allocation lock, this is a non-scalable design that creates a bottleneck—21.7% of the cycles are spent waiting for the invalidation lock in our experiment.

Masking the IOVA allocator bottleneck Interestingly, the IOTLB invalidation bottleneck throttles the rate of IOVA allocation/deallocation operations, and thereby *masks* the severity of the IOVA allocator bottleneck. Deallocations are throttled because they occur while processing the flush queue—i.e., under the invalidation lock—and are therefore serialized. Allocations (mappings) are throttled because they are interleaved with unmappings. Since unmapping is slow because of the IOTLB bottleneck, the interval between mappings increases and their frequency decreases.

Once we eliminate the IOTLB invalidation bottleneck, however, pressure on the IOVA allocation lock increases and with it the severity of its performance impact. Indeed, as Figure 2 shows, adding scalable deferred IOTLB invalidation (§ 5) to Linux/EiovaR increases IOVA lock waiting time by 2.1×.

3.3 Linux Page Table Management

IOMMU page table management involves two tasks: updating the page tables when creating/destroying mappings, and reclaiming physical pages that are used as page tables when they become empty.

3.3.1 Updating Page Tables

We distinguish between updates to *last-level* page tables (leafs in the page table tree) and *page directories* (inner nodes).

For last-level page tables, the Linux IOVA allocator enables synchronization-free updates. Because each mapping is associated with a unique IOVA page range, updates of distinct mappings involve distinct page table entries. Further, the OS does not allow concurrent mapping/unmapping of the same IOVA range. Consequently, it is safe to update entries in last-level page tables without locking or atomic operations.

Updating page directories is more complex, since each page directory entry (PDE) may map multiple IOVA ranges (any range potentially mapped by a child page table), and multiple cores may be concurrently mapping/unmapping these ranges. A new child page table is allocated by updating an empty PDE to point to a new page table. To synchronize this action in a parallel scenario, we allocate a physical page *P* and then attempt to point the PDE to *P* using an atomic operation. This attempt may fail if another core has pointed the PDE to its own page table in the mean time, but in this case we simply free *P* and use the page table installed by the other core. Deallocation of page tables is more complex, and is discussed in the following section.

The bottom line, however, is that updating page tables is relatively cheap. Page directories are updated infrequently and these updates rarely experience conflicts. Similarly, last-level page table updates are cheap and conflict free. As a result, page table updates account for about 2.8% of the cycles in the system (Figure 2).

3.3.2 Page Table Reclamation

To reclaim a page table, we must first be able to remove any reference (PDE) to it. This requires some kind of synchronization to atomically (1) determine that the page table is empty, (2) remove the pointer from the parent PDE to it, (3) prevent other cores from creating new entries in the page table in the mean time. In addition, because the IOTLB could cache PDEs [27], we can only reclaim the physical page that served as the now-empty page table after invalidating the IOTLB. Before that, it is not safe to reclaim this memory or to map any other IOVA in the range controlled by it.

Due to this complexity, the Linux design sidesteps this issue and does not reclaim page table memory, unless the entire region covered by a PDE is freed in one unmap action. Thus, once a PDE is set to point to some page *P*, it is unlikely to ever change, which in turn reduces the number of updates that need to be performed for PDEs. This simple implementation choice is largely enabled by the IOVA allocation policy of packing IOVA ranges close to the top of the address space. This policy results in requiring a minimal number of page tables to map the allocated IOVA ranges, which makes memory consumption by IOMMU page tables tolerable.

3.4 IOMMU Management in Other OSes

This section compares the Linux/Intel-x86 IOMMU management design to the designs used in the FreeBSD, Solaris[3], and Mac OS X systems. Table 1 summarizes our findings, which are detailed below. In a nutshell, we find that (1) all OSes have scalability bottlenecks sim-

[3]Our source code references are to illumos, a fork of OpenSolaris. However, the code in question dates back to OpenSolaris.

	IOVA allocation		IOTLB invalidation		PT management	
	Allocator	Scales?	Strict?	Scales?	Scales?	Free mem?
Linux/Intel-x86 (§§ 3.1–3.3)	Red-black tree: linear time (made constant by EiovaR).	✗	✗	✗	✓	Rarely
FreeBSD [20]	Red-black tree: logarithmic time	✗	✓	✗	✗	Yes
Solaris [21]	Vmem [11]: constant time	✗	✗	✗	✗	No
Mac OS X [4]	Buddy allocator/red-black tree (size dependent): logarithmic time	✗	✗	✗	✗	No

Table 1. Comparison of IOMMU management designs in current OSes

ilar to—or worse than—Linux, (2) none of the OSes other than FreeBSD reclaim IOMMU page tables, (3) FreeBSD is the only OS to implement strict IOTLB invalidation. The other OSes loosen their intra-OS protection guarantees for increased performance. The last observation supports our choice of optimizing the Linux deferred invalidation design in this work. Our scalable IOMMU management designs (or simple variants thereof) are thus applicable to these OSes as well.

IOVA allocation All systems use a central globally-locked allocator, which is invoked by each IOVA allocation/deallocation operation, and is thus a bottleneck. The underlying allocator in FreeBSD is a red-black tree of allocated ranges, similarly to Linux. However, FreeBSD uses a different traversal policy, which usually finds a range in logarithmic time [35]. Solaris uses the Vmem resource allocator [11], which allocates in constant time. Mac OS X uses two allocators, both logarithmic—a buddy allocator for small ($\leq$ 512 MB) ranges and a red/black tree allocator for larger ranges.

IOTLB invalidation FreeBSD is the only OS that implements strict IOTLB invalidations, i.e., waits until the IOTLB is invalidated before completing an IOVA unmap operation. The other OSes defer invalidations, although differently than Linux: Solaris does not invalidate the IOTLB when unmapping. Instead, it invalidates the IOTLB when mapping an IOVA range, to flush any previous stale mapping. This creates an unbounded window of time in which a device can still access unmapped memory. An unmap on Mac OS X buffers an IOTLB invalidation request in the cyclic IOMMU invalidation queue and returns without waiting for the invalidation to complete. All these designs acquire the lock protecting the IOMMU invalidation queue for each operation, and thus do not scale.

Page table management Linux has the most scalable IOMMU page table management scheme—exploiting IOVA range disjointness to update last-level PTEs without locks and inner PDEs with atomic operations. In contrast, FreeBSD performs page table manipulations under a global lock. Solaris uses a more fine-grained technique, protecting each page table with a read/write lock. However, the root page table lock is acquired by every operation and thus becomes a bottleneck, since even acquisitions of a read/write lock in read mode create contention on the lock's shared cache line [33]. Finally, Mac OS X updates page tables under a global lock when using the buddy allocator, but outside of the lock—similarly to Linux—when allocating from the red-black tree.

Page table reclamation Similarly to Linux, Solaris does not detect when a page table becomes empty and thus does not attempt to reclaim physical pages that serve as page tables. Mac OS X bounds the number of IOMMU page tables (and therefore the size of I/O virtual memory supported) and allocates them on first use while holding the IOVA allocator lock. Mac OS X does not reclaim page tables as well. Only FreeBSD actively manages IOMMU page table memory; it maintains a count of used PTEs in each page table, and frees the page table when it becomes empty.

4 Scalable IOVA Allocation

Here we describe three designs for scalable IOVA assignment, exploring several points in this design space: (1) dynamic identity mapping (§ 4.1), which does away with IOVA allocation altogether, (2) IOVA-kmalloc (§ 4.2) which implements IOVA allocation but does away with its dedicated allocator, and (3) scalable IOVA allocation (§ 4.3), which uses *magazines* [11] to alleviate contention on the IOVA allocator using per-core caching of freed IOVA ranges.

4.1 Dynamic Identity Mapping

The fastest code is code that never runs. Our *dynamic identity mapping* design applies this principle to IOVA allocation. We observe that ordinarily, the buffers that a driver wishes to map are already physically contiguous. We thus propose to use such a physically contiguous buffer's physical address as its IOVA when mapping it, resulting in an identity (1-to-1) mapping in the IOMMU. Due to intra-OS protection reasons, when the

driver unmaps a buffer we must destroy its identity mapping. We therefore refer to this scheme as *dynamic* identity mappings—while the same buffer will always use the same mapping, this mapping is dynamically created and destroyed to enforce protection of the buffer's memory.

Dynamic identity mapping eliminates the work and locking involved in managing a distinct space of IOVAs, replacing it with the work of translating a buffer's kernel virtual address to a physical address. In most OSes, this is an efficient and scalable operation—e.g., adding a constant to the kernel virtual address. However, while dynamic identity mapping completely eliminates the IOVA allocation bottleneck, it turns out to have broader implications for other parts of the IOMMU management subsystem, which we now discuss.

Page table entry reuse Standard IOVA allocation associates each mapping with a distinct IOVA. As a result, multiple mappings of the same page (e.g., for different buffers on the same page) involve different page table entries (§ 3.3). Dynamic identity mapping breaks this property: a driver—or concurrent cores—mapping a previously mapped page will need to update exactly the same page table entries (PTEs).

While repeated mapping operations will always write the same value (i.e., the physical address of the mapped page), unmapping operations pose a challenge. We need to detect when the last unmapping occurs, so we can clear the PTE. This requires maintaining a reference count for each PTE. We use 10 of the OS-reserved bits in the last-level PTE structure [27] to maintain this reference count. When the reference count hits zero, we clear the PTE and request an IOTLB invalidation.

Because multiple cores may concurrently update the same PTE, all PTE updates—including reference count maintenance—require atomic operations. In other words, we need to (1) read the PTE, (2) compute the new PTE value, (3) update it with an atomic operation that verifies the PTE has not changed in the mean time, and (4) repeat this process if the atomic operation fails.

Conflicting access permissions A second problem posed by not having unique PTEs for each mapping is how to handle mappings of the same physical page with conflicting access permission (read, write, or both). For example, two buffers may get allocated by the network stack on the same page—one for RX use, which the NIC should write to, and one for TX, which the NIC should only read. To maintain intra-OS protection, we must separately track mappings with different permissions, e.g., so that once all writable mappings of a page are destroyed, no PTE with write permissions remains. Furthermore, even when a mapping with write permissions exists, we want to avoid promoting PTEs that should only be used to read (and vice-versa), as this kind of access should not happen during the normal course operation for a properly functioning device and should be detected and blocked.

To solve this problem, we exploit the fact that current x86 processors support 48-bit I/O virtual memory addresses, but only 46-bits of physical memory addresses. The two most significant bits in each IOVA are thus "spare" bits, which we can use to differentiate mappings with conflicting access permissions. Effectively, we partition the identity mapping space into regions: three regions, for read, write and read/write mappings, and a fourth fallback region that contains addresses that cannot be mapped with identity mappings—as discussed next.

Non-contiguous buffers Some mapping operations are for physically non-contiguous buffers. For example, Linux's scatter/gather mapping functions map non-consecutive physical memory into contiguous virtual I/O memory. Since identity mappings do not address this use case, we fall back to using the IOVA allocator in such cases. To avoid conflicts with the identity mappings, IOVAs returned by the IOVA allocator are used in the fourth fallback region.

PTE reference count overflow We fall back to standard IOVA allocation for mappings in which the 10-bit reference count overflows. That is, if we encounter a PTE whose reference count cannot be incremented while creating a dynamic identity mapping, we abort (decrementing the references of any PTEs previously incremented in the mapping process) and create the mapping using an IOVA obtained from the IOVA allocator.

Page table memory Because physical addresses mapped by drivers are basically arbitrary, we lose the property that IOVAs are densely packed. Consequently, more memory may be required to hold page tables. For example, if the first 512 map operations are each for a single page, IOVA allocation will map them all through the same last-level page table. With physical addresses, we may need at least a page table per map operation. Unfortunately, the physical pages used to hold these page tables will not get reclaimed (§ 3.3). In the worst case, page table memory consumption may keep increasing as the system remains active.

4.2 IOVA-kmalloc

Our IOVA-kmalloc design explores the implications of treating IOVA allocation as a general allocation problem. Essentially, to allocate an IOVA range of size *R*, we can allocate a block of *R* bytes in physical memory using the system's `kmalloc` allocator, and use the block's address as an IOVA (our actual design is much less wasteful, as discussed below). The addresses `kmalloc` returns are unique per allocation, and thus IOVA-kmalloc maintains the efficient conflict-free updates of IOMMU page

tables enabled by the original IOVA allocator.

One crucial difference between `kmalloc` and IOVA allocation is that IOVAs are abstract—essentially, opaque integers—whereas `kmalloc` allocates physically contiguous memory. It might therefore seem that the IOVA-kmalloc approach unacceptably wastes memory, since allocating R bytes to obtain an R-byte IOVA range doubles the system's memory consumption. Fortunately, we observe that the IOVA allocator need only allocate virtual I/O *page frame numbers* (PFNs), and not arbitrary ranges. That is, given a physical buffer to map, we need to find a range of *pages* that contains this buffer. This makes the problem tractable: we can treat the physical addresses that `kmalloc` returns as PFNs. That is, to map a range of R bytes, we `kmalloc` $\lceil R/4096 \rceil$ bytes and interpret the address of the returned block B as a range of PFNs (i.e., covering the IOVA range of $[4096B, 4096(B + \lceil R/4096 \rceil)]$.

While this still wastes physical memory, the overhead is only 1 byte per virtual I/O page, rounded up to 8 bytes (the smallest unit `kmalloc` allocates internally). By comparison, the last-level PTE required to map a page is itself 8 bytes, so the memory blowup caused by IOVA-kmalloc allocating actual physical memory is never greater than the memory overhead incurred by stock Linux for page table management.

In fact, being a full-blown memory allocator that manages actual memory (not abstract integers) might actually turn out to be advantageous for `kmalloc`, as this property enables it to use the many techniques and optimizations in the memory allocation literature. In particular, `kmalloc` implements a version of *slab allocation* [10], a fast and space-efficient allocation scheme. One aspect of this technique is that `kmalloc` stores the slab that contains metadata associated with an allocated address in the `page` structure of the address. This allows `kmalloc` to look up metadata in constant time when an address gets freed. In contrast, the Linux IOVA allocator has to maintain metadata *externally*, in the red-black tree, because there is no physical memory backing an IOVA. It must thus access the globally-locked tree on each deallocation.

Page table memory blowup The main downside of IOVA-kmalloc is that we have no control over the allocated addresses. Since `kmalloc` makes no effort to pack allocations densely, the number of page tables required to hold all the mappings will be larger than with the Linux IOVA allocator. Moreover, if the same physical address is mapped, unmapped, and then mapped again, IOVA-kmalloc may use a different IOVA when remapping. Because Linux does not reclaim page table memory, the amount of memory dedicated to page tables can grow without bound. In contrast, dynamic identity mapping always allocates the same IOVA to a given physical buffer. However, in an OS that manages page table memory, unbounded blowup cannot occur.

To summarize, IOVA-kmalloc represents a classic *time/space* trade-off—we relinquish memory in exchange for the efficiency and scalability of `kmalloc`, which is highly optimized due to its pervasive use throughout the kernel. Notably, these advantages come *for free*, in terms of implementation complexity, debugging and maintenance, since `kmalloc` is already there, performs well, and is trivial to use.

Handling IOVA collisions IOVAs are presently 48 bits wide [27]. x86 hardware presently supports 46-bits of physical address space [26]. Thus, because we treat `kmalloc` addresses as virtual I/O PFNs, IOVA-kmalloc may allocate two addresses that collide when interpreted as PFNs in the 48-bit I/O virtual address space. That is, we have 2^{36} possible IOVA PFNs since pages are 4 KB, but `kmalloc` allocates from a pool of up to 2^{46} bytes of physical memory.

Such collisions are mathematically impossible on systems with at most 64 GB of physical memory (whose physical addresses are 36-bit). To avoid these collisions on larger systems, IOVA allocations can invoke `kmalloc` with the `GFP_DMA` flag. This flag instructs `kmalloc` to satisfy the allocation from a low memory zone whose size we can configure to be at most 64 GB.[4]

Why allocate addresses? We could simply use a mapped buffer's kernel virtual address as its IOVA PFN. However, we then lose the guarantee that different mappings obtain different IOVA PFNs (e.g., as the same buffer can be mapped twice). This is exactly the problem dynamic identity mapping tackles, only here we do not have "spare" bits to distinguish mappings with different access rights as the virtual address space is 48 bits.

4.3 Scalable IOVA Allocation with Magazines

Our final design addresses the IOVA allocator bottleneck head-on, turning it into a scalable allocator. The basic idea is to add a *per-core cache* of previously deallocated IOVA ranges. If most allocations can be satisfied from the per-core cache, the actual allocator—with its lock—will be invoked rarely.

Per-core caching poses several requirements. First, the per-core caches should be *bounded*. Second, they should efficiently handle producer/consumer scenarios observed in practice, in which one core continuously allocates IOVAs, which are later freed by another core. In a trivial design, only the core freeing IOVAs will build up a cache, while the allocating core will always invoke the under-

[4] In theory, the `GFP_DMA` memory zone must be accessible to legacy devices for DMA and thus addressable with 24 bits. But as we have an IOMMU, we only need to ensure that the IOVA ranges mapped to legacy devices fall into the 24-bit zone.

lying non-scalable allocator. We require that *both* cores avoid interacting with the IOVA allocator.

Magazines [11] provide a suitable solution. A *magazine* is an M-element per-core cache of objects—IOVA ranges, in our case—maintained as a stack of objects. Conceptually, a core trying to allocate from an empty magazine (or deallocate into a full magazine) can return its magazine to a globally-locked *depot* in exchange for a full (or empty) magazine. Magazines actually employ a more sophisticated *replenishment policy* which guarantees that a core can satisfy at least M allocations and at least M deallocations from its per-core cache before it must access the depot. Thus, a core's miss rate is bounded by $1/M$.

We implement magazines on top of the original Linux IOVA allocator, maintaining separate magazines and depots for different allocation sizes. Thus, this design still controls page table blowup, as the underlying allocator densely packs allocated IOVAs.

5 Scalable IOTLB Invalidation

This section describes a scalable implementation of deferred IOTLB invalidations. Recall that Linux maintains a *flush queue* containing a globally ordered list of all IOVA ranges pending invalidation. We observe, however, that a global flush queue—with its associated lock contention—is overkill. There is no real dependency between the invalidation process of distinct IOVA ranges. Our only requirements are that until an entire IOVA range mapping is invalidated in the IOTLB:

1. The IOVA range will not be released back to the IOVA allocator, to avoid having it allocated again before the old mapping is invalidated.

2. The page tables that were mapping the IOVA range will not be reclaimed. Since page directory entries are also cached in the IOTLB, such a reclaimed page table might get reused and data that appears as a valid page table entry be written to it, and this data might then be used by the IOMMU.

Our approach We satisfy these requirements in a much more scalable manner by batching invalidation requests *locally* on each core instead of globally. Implementing this approach simply requires replicating the current flush queue algorithm on a per-core basis. With this design, the lock on a core's flush queue is almost always acquired by the owning core. The only exception is when the queue's global invalidation timeout expires—the resulting callback, which acquires the lock, may be scheduled on a different core. However, not only does such an event occur rarely (at most once every 10 ms), but it suggests that the IOMMU management subsystem is not heavily used in the first place.

The remaining source of contention in this design is the serialization of writes to the cyclic IOMMU invalidation queue—which is protected by a lock—in order to buffer global IOTLB invalidation requests. Fortunately, accesses to the invalidation queue are infrequent, with at most one invalidation every 250 invalidations or 10 ms, and short, as an invalidation request is added to the buffer and the lock is released; the core waits for the IOMMU to process its queued invalidation without holding the lock.

Security-wise, while we now might batch 250 invalidation requests per core, a destroyed IOVA range mapping will usually be invalidated in the IOTLB just as quickly as before. The reason is that *some* core performs a global IOTLB invalidation, on average, every 250 global invalidation requests. Thus, we do not substantially increase the window of time in which a device can access an unmapped physical page. Unmapped IOVA ranges may, however, remain in a core's flush queue and will not be returned to the IOVA allocator for a longer time than the baseline design—they will be freed only when the core itself processes its local flush queue. We did not observe this to be a problem in practice, especially when the system experiences high IOMMU management load.

6 Evaluation

Here we evaluate our designs (§§ 4–5) and explore the trade-offs they involve. We study two metrics: the performance obtained (§ 6.1), and the complexity of implementing the designs (§ 6.2).

6.1 Performance

Experimental setup We implement the designs in Linux 3.17.2. Our test setup consists of a client and a server, both Dell PowerEdge R430 rack machines. Each machine has dual 2.4 GHz Intel Xeon E5-2630 v3 8-core processors, for a total of 16 cores per machine (hyperthreading is disabled). Both machines are equipped with 32 GB 2133 MHz memory. The server is equipped with a Broadcom NetXtreme II BCM57810 10 Gb/s NIC. The client is equipped with an Intel 82599 10 Gb/s NIC and runs an unmodified Ubuntu 3.13.0-45 Linux kernel. The client's IOMMU is disabled for all evaluations. The client and the server NICs are cross-connected to avoid network interference.

Methodology To obtain the best scalability, we (1) configure the NIC on the server to use 15 rings, which is the maximum number supported by the Broadcom NIC, allowing cores to interact with the NIC with minimal lock contention (no ring contention with < 16 cores), and (2) distribute delivery of interrupts associated with different rings across different cores.[5] Benchmarks are

[5] Default delivery is to core #0, which overloads this core.

executed in a round-robin fashion, with each benchmark running once for 10 seconds for each possible number of cores, followed by all benchmarks repeating. The cycle is run five times and the reported results are the medians of the five runs.

Benchmarks In our benchmarks, we aim to use concurrent workloads that stress the IOMMU management subsystem.

Our first few benchmarks are based on `netperf` [28], a prominent network analysis tool. In our highly parallel RR (request-response) benchmark, we run 270 instances of the `netperf` TCP RR test on the client, and limit the server side of `netperf` to the number of cores we wish to test on. Each TCP RR instance sends a TCP packet with 1 byte of payload to the server and waits for a 1 byte response before sending the next one. In all, our benchmark has 270 ongoing requests to the `netperf` server at any given time, which bring the server close to 100% CPU utilization even with IOMMU disabled. We report the total number of such request-response transactions the server responds to during the testing period.

The second benchmark we use `netperf` for is a parallel latency test. To achieve that, we run the `netperf` TCP RR test with as many instances as we allow server cores. This allows each `netperf` request-response connection to run on its own dedicated core on both the client and the server.

Our third benchmark is `memcached` [19], a key-value store service used by web applications to speed up read operations on slow resources such as databases and remote API calls. To avoid lock contention within `memcached`, we run multiple instances, each pinned to run on a specific core. We measure `memcached` throughput using `memslap` [1], which distributes requests among the `memcached` instances. We configure `memslap` to run on the client with 16 threads and 256 concurrent requests (16 per thread). We use `memslap`'s default configuration of a 64-byte key, 1024-byte value and a ratio of 10%/90% `SET`/`GET` operations.

We note that network bandwidth benchmarks would be less relevant here, as a single core can saturate the network on our machines. As an example, `netperf` TCP STREAM, which makes use of the NIC's offloading features, is able to saturate the network using 22% CPU time on a single core even running under EiovaR-Linux.

Results Figure 3 depicts the throughput achieved by our highly parallel RR benchmark. Without an IOMMU, Linux scales nicely and obtains 14.4× TPS with 16 cores than with a single core. Because of the IOMMU management bottlenecks, EiovaR-Linux does not scale as well, obtaining only a 3.8× speedup. In particular, while EiovaR obtains 86% of the No-IOMMU throughput with 1 core (due to the single-threaded overheads of IOMMU

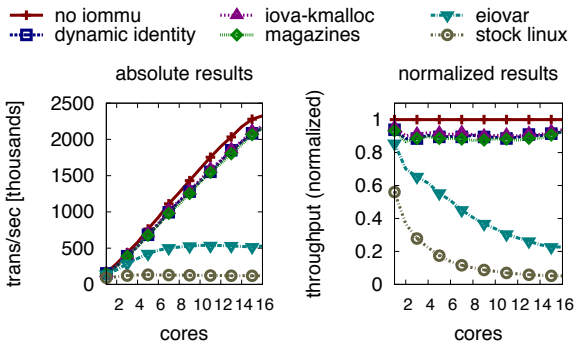


Figure 3. Highly parallel RR throughput

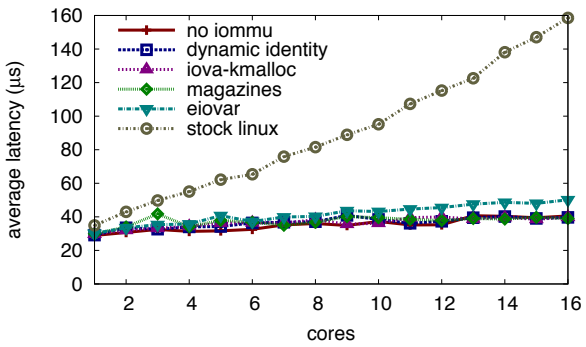


Figure 4. 1 `netperf` TCP RR instance per core latency test

management), it only achieves 23% of the No-IOMMU throughput at 16 cores.

Our designs all perform at 93%–95% of No-IOMMU on a single core and scale far better than EiovaR, with 16-core throughput being 93% (magazines and dynamic identity mapping) and 94% (IOVA-kmalloc) of the No-IOMMU results. Because of the overhead dynamic identity mapping adds to page table updates, it does not outperform the IOVA allocating designs—despite not performing IOVA allocation at all.

To summarize, in our designs IOMMU overhead is essentially *constant* and does not get worse with more concurrency. Most of this overhead consists of page tables updates, which are essential when managing the IOMMU in a transient manner.

In our parallel latency test, shown in Figure 4, we see that Linux's latency increases with multiple cores, even without IOMMU, from 29μs with one instance to 41μs with 16 instances, each with its own core. While EiovaR starts within 1.2μs of Linux's latency on one core, the contention caused by its locks causes a 10μs gap at 16 cores. For the most part, our designs are on par with Linux's performance, actually achieving slightly shorter latency than No-IOMMU Linux on 16 cores.

The evaluation of `memcached` in Figure 5 paints a similar picture to Figure 3 with up to 12 cores. The IOMMU subsystem is indifferent to the different packet size in this workload (1052 bytes here, 1 byte in TCP RR) as the

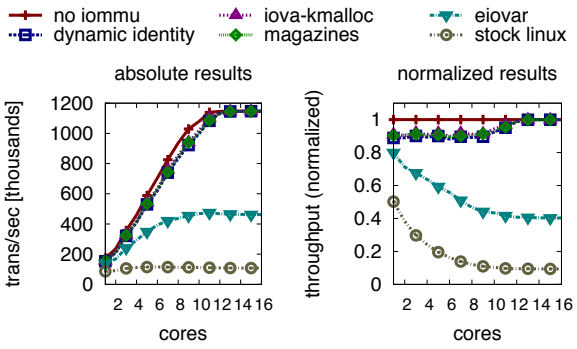


Figure 5. `memcached` throughput

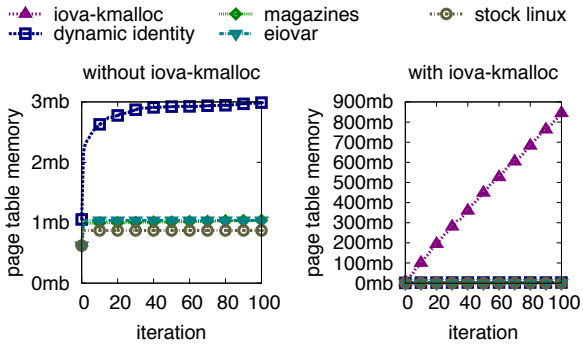


Figure 6. Page table memory over time

mappings are done per page and no data copying takes place. Starting at 12 cores, all of our designs achieve line rate and therefore close the gap with No-IOMMU performance. Here, too, IOVA-kmalloc demonstrates the best performance out of our designs by a small margin (in all but 2 cores and >12 cores, where they all achieve line rate), as it is the most highly optimized of the three. With a single core, all of our designs are between 89% (dynamic identity) and 91% (IOVA-kmalloc) of No-IOMMU performance. At 9 cores, after which No-IOMMU Linux throughput begins to near line rate, our designs' relative throughput is between 89.5% (dynamic identity) and 91.5% (IOVA-kmalloc). EiovaR also stops scaling well before 16 cores, but as opposed to our designs, it does not do that due to achieving line rate, plateauing at 40% of it, starting with 9 cores.

Page table memory consumption Linux rarely reclaims a page used as an IOMMU page table (§ 3.3). Figure 6 illustrates the memory consumption dynamics this policy creates. We run 100 iterations of our parallel RR benchmark and plot the number of IOMMU page tables (in all levels) that exist in the system before the tests start (but after system and NIC initialization) and after each iteration.

Both EiovaR and our magazine-based design, which are based on Linux's current IOVA allocator, minimize page table memory consumption by packing allocated IOVAs at the top of the address space. As Figure 6 shows,

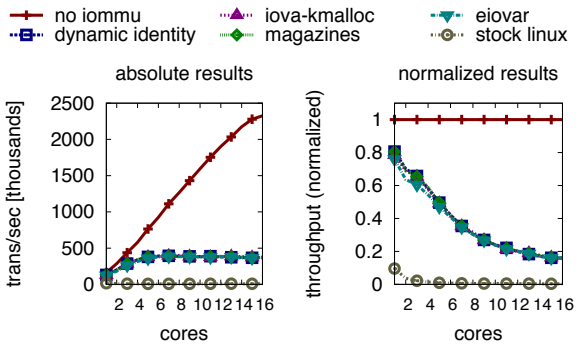


Figure 7. Highly parallel `netperf` txns/sec with strict IOTLB invalidation

both of them, as well as stock Linux, allocate most page tables at NIC initialization, when all RX rings are allocated and mapped, as well as all TX/RX descriptors and other metadata.

Our magazine-based design consumes $1.2\times$ more page table memory than stock Linux, because the cores cache freed IOVAs in their per-core caches instead of returning them to the global allocator, requiring other cores to allocate new IOVAs from the allocator and thus increasing page table use.

While dynamic identity mapping does not guarantee an upper bound on the number of IOVAs it utilizes over time, it turns out that the APIs used by the driver to allocate buffers use caching themselves. Consequently, mapped addresses repeat across the iterations and the number of page tables does not explode. Still, $3.4\times$ more page tables than stock Linux are allocated, since the addresses are not packed.

In contrast to the other schemes, IOVA-kmalloc exhibits a blowup of page table memory use. Since the addresses IOVA-kmalloc receives from `kmalloc` are influenced by system-wide allocation activity, IOVA-kmalloc uses a much wider and constantly changing set of IOVAs. This causes its page table memory consumption to grow in an almost linear fashion. We conclude that while IOVA-kmalloc is the best performing design, by a slight margin, its use must be accompanied by memory reclamation of unused page tables. This underscores the need for managing the IOMMU page table memory in a scalable manner—§ 3.3 describes the challenges involved.

Strict invalidation Figure 7 shows the parallel RR benchmark throughput with *strict* IOTLB invalidation, i.e., full intra-OS protection. As invalidations are not deferred, our scalable IOTLB invalidation optimization is irrelevant for this case. While we observe very minor throughput difference between the IOVA allocation designs when using a small number of cores, even this difference is no longer evident when more than 6 cores are used, with all designs obtaining about $1/6$ the TPS

Design	Lines add	Lines del	Files	Impl. time
Dynamic identity mapping (§ 4.1)	+397	-92	1	Weeks
IOVA-kmalloc (§ 4.2)	+56	-44	1	Hours
IOVA allocation with magazines (§ 4.3)	+362	-79	3	Days
Scalable IOTLB invalidation (§ 5)	+97	-37	1	Hours

Table 2. Implementation complexity of our designs

of No-IOMMU Linux by 16 cores. In all designs, the contention over the invalidation queue lock becomes the dominating factor (§ 3.2). Thus, strict IOTLB invalidation appears incompatible with a scalable implementation.

6.2 Implementation Complexity

Table 2 reports the source code changes required to implement our designs in Linux 3.17.2. The table also estimates the time it would take us to re-implement each approach from scratch. IOVA-kmalloc is the simplest design to implement, essentially replacing calls to the IOVA allocator with a `kmalloc()` call. Most of the line changes reported for IOVA-kmalloc are due to technical changes, replacing the `struct` representing an IOVA with an integer type. Implementation of the magazines design is a bit more complex, requiring an implementation of a magazine-based caching layer on top of the existing IOVA allocator, as well as optimizing its locking strategy. Dynamic identity mapping is the most complicated and intrusive of our IOVA assignment designs, as it calls for a surgical modification of page table management itself, maintaining mapping book-keeping within the table itself and synchronizing updates to the same mapping from multiple cores in parallel.

7 Related Work

Most work addressing the poor performance associated with using IOMMUs tackles only sequential performance [3, 9, 14, 34, 42, 45, 47, 35], for example by reducing the number of mapping operations [45], deferring or offloading IOTLB invalidation work [3], and improving the IOVA allocator algorithm [14, 35, 42]. Cascardo [14] does consider multicore workloads, but his proposal improves only the sequential part of the IOVA allocator. In contrast, our work identifies and attacks the scalability bottlenecks in current IOMMU management designs. In addition, our scalable IOVA allocation is orthogonal to sequential improvements in the IOVA allocator, since it treats it as a black box.

Clements et al. propose designs for scalable management of process virtual address spaces [16, 17]. I/O virtual memory has simpler semantics than standard virtual memory, which allows us to explore simpler designs. In particular, (1) IOVA ranges cannot be partially unmapped, unlike standard `mmap()`ed ranges, and (2) IOVA mappings can exploit the preexisting address of the mapped buffers (as in dynamic identity mapping), while creation of a virtual memory region can occur before the allocation of the physical memory backing it.

Bonwick introduced magazines to improve scalability in the Vmem resource allocator [11]. Since Vmem is a general allocator, however, it does not minimize page table memory consumption of allocated IOVAs, in contrast to the specialized Linux allocator. Our IOVA-kmalloc design additionally shows that a dedicated resource allocator may not be required in the first place. Finally, part of our contribution is the re-evaluation of magazines on modern machines and workloads.

8 Conclusion

Today, IOMMU-based intra-OS protection faces a performance challenge in high throughput and highly concurrent I/O workloads. In current OSes, the IOMMU management subsystem is not scalable and creates a prohibitive bottleneck.

Towards addressing this problem, we have explored the design space of scalable IOMMU management approaches. We proposed three designs for scalable IOVA assignment—dynamic identity mapping, IOVA-kmalloc, and per-core IOVA caching—as well as a scalable IOTLB invalidation scheme. Our designs achieve 88.5%–100% of the performance obtained *without* an IOMMU.

Our evaluation demonstrates the trade-offs of the different designs. Namely, (1) the savings dynamic identity mapping obtains from not allocating IOVAs are negated by its more expensive IOMMU page table management, making it perform comparably to scalable IOVA allocation, and (2) IOVA-kmalloc provides a simple solution with high performance, but suffers from unbounded page table blowup. This emphasizes the importance of managing the IOMMU *page table memory* in a scalable manner as well, which is interesting future work.

References

[1] AKER, B. Memslap - load testing and benchmarking a server. `http://docs.libmemcached.org/bin/memslap.html`. libmemcached 1.1.0 documentation.

[2] AMD INC. AMD IOMMU architectural specification, rev 2.00. `http://developer.amd.com/wordpress/media/2012/10/488821.pdf`, Mar 2011.

[3] AMIT, N., BEN-YEHUDA, M., TSAFRIR, D., AND SCHUSTER, A. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)* (2011), pp. 73–86.

[4] APPLE COMPUTER. IOPCIFamily/IOPCIFamily-196.3/vtd.c, Source Code File of Mac OS X. `http:`

//www.opensource.apple.com/source/IOPCIFamily/IOPCIFamily-196.3/vtd.c. (Accessed: Jan 2015).

[5] APPLE INC. Thunderbolt device driver programming guide: Debugging VT-d I/O MMU virtualization. https://developer.apple.com/library/mac/documentation/HardwareDrivers/Conceptual/ThunderboltDevGuide/DebuggingThunderboltDrivers/DebuggingThunderboltDrivers.html, 2013. (Accessed: May 2014).

[6] ARM HOLDINGS. ARM system memory management unit architecture specification — SMMU architecture version 2.0. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0062c/IHI0062C_system_mmu_architecture_specification.pdf, 2013.

[7] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *ACM EuroSys* (2006), pp. 73–85.

[8] BECHER, M., DORNSEIF, M., AND KLEIN, C. N. FireWire: all your memory are belong to us. In *CanSecWest Applied Security Conference* (2005).

[9] BEN-YEHUDA, M., XENIDIS, J., OSTROWSKI, M., RISTER, K., BRUEMMER, A., AND VAN DOORN, L. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)* (2007), pp. 9–20.

[10] BONWICK, J. The Slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Annual Technical Conference* (1994), pp. 87–98.

[11] BONWICK, J., AND ADAMS, J. Magazines and Vmem: Extending the Slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference (ATC)* (2001), pp. 15–44.

[12] BOTTOMLEY, J. E. Dynamic DMA mapping using the generic device. https://www.kernel.org/doc/Documentation/DMA-API.txt. Linux kernel documentation.

[13] CARRIER, B. D., AND GRAND, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1*, 1 (Feb 2014), 50–60.

[14] CASCARDO, T. DMA API performance and contention on IOMMU enabled environments. http://events.linuxfoundation.org/images/stories/slides/lfcs2013_cascardo.pdf, 2013.

[15] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An empirical study of operating systems errors. In *ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 73–88.

[16] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scalable Address Spaces Using RCU Balanced Trees. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 199–210.

[17] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *ACM EuroSys* (2013), pp. 211–224.

[18] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium* (2009), pp. 83–100.

[19] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug 2004).

[20] FREEBSD FOUNDATION. x86/iommu/intel_gas.c, source code file of FreeBSD 10.1.0. https://github.com/freebsd/freebsd/blob/release/10.1.0/sys/x86/iommu/intel_gas.c. (Accessed: Jan 2015).

[21] GARRETT D'AMORE. i86pc/io/immu_dvma.c, Source Code File of illumos. https://github.com/illumos/illumos-gate/blob/master/usr/src/uts/i86pc/io/immu_dvma.c. (Accessed: Jan 2015).

[22] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Failure resilience for device drivers. In *IEEE/IFIP Annual International Conference on Dependable Systems and Networks (DSN)* (2007), pp. 41–50.

[23] HILL, B. Integrating an EDK custom peripheral with a LocalLink interface into Linux. Tech. Rep. XAPP1129 (v1.0), XILINX, May 2009.

[24] IBM CORPORATION. PowerLinux servers — 64-bit DMA concepts. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/topic/liabm/liabmconcepts.htm. (Accessed: May 2014).

[25] IBM CORPORATION. AIX kernel extensions and device support programming concepts. http://public.dhe.ibm.com/systems/power/docs/aix/71/kernextc_pdf.pdf, 2013. (Accssed: May 2015).

[26] INTEL CORPORATION. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3: System Programming Guide. http://download.intel.com/products/processor/manual/325384.pdf, 2013.

[27] INTEL CORPORATION. Intel Virtualization Technology for Directed I/O, Architecture Specification - Architecture Specification - Rev. 2.3. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, Oct 2014.

[28] JONES, R. A. A network performance benchmark (revision 2.0). Tech. rep., Hewlett Packard, 1995. http://www.netperf.org/netperf/training/Netperf.html.

[29] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Tolerating hardware device failures in software. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 59–72.

[30] LEITAO, B. H. Tuning 10Gb network cards on Linux. In *Ottawa Linux Symposium (OLS)* (2009), pp. 169–189.

[31] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (2004), pp. 17–30.

[32] Documentation/intel-iommu.txt, Linux 3.18 documentation file. https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt. (Accessed: Jan 2015).

[33] LIU, R., ZHANG, H., AND CHEN, H. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *USENIX Annual Technical Conference (ATC)* (2014), pp. 219–230.

[34] MALKA, M., AMIT, N., BEN-YEHUDA, M., AND TSAFRIR, D. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015), pp. 355–368.

[35] MALKA, M., AMIT, N., AND TSAFRIR, D. Efficient Intra-Operating System Protection Against Harmful DMAs. In *USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 29–44.

[36] MAMTANI, V. DMA directions and Windows. http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t304_wh07.pptx, 2007. (Accessed: May 2014).

[37] MILLER, D. S., HENDERSON, R., AND JELINEK, J. Dynamic DMA mapping guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt. Linux kernel documentation.

[38] PCI-SIG. Address translation services revision 1.1. https://www.pcisig.com/specifications/iov/ats, Jan 2009.

[39] ROEDEL, J. IOMMU Page Faulting and MM Integration. http://www.linuxplumbersconf.org/2014/ocw/system/presentations/2253/original/iommuv2.pdf. Linux Plumbers Conference 2014.

[40] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 335–350.

[41] SWIFT, M., BERSHAD, B., AND LEVY, H. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS) 23*, 1 (Feb 2005), 77–110.

[42] TOMONORI, F. DMA representations sg_table vs. sg_ring IOMMUs and LLD's restrictions. In *USENIX Linux Storage and Filesystem Workshop (LSF)* (2008). https://www.usenix.org/legacy/events/lsf08/tech/IO_tomonori.pdf.

[43] WALDSPURGER, C., AND ROSENBLUM, M. I/O virtualization. *Communications of the ACM (CACM) 55*, 1 (Jan 2012), 66–73.

[44] WILLIAMS, D., REYNOLDS, P., WALSH, K., SIRER, E. G., AND SCHNEIDER, F. B. Device driver safety through a reference validation mechanism. In *USENIX Symposium on Operating System Design and Implementation (OSDI)* (2008), pp. 241–254.

[45] WILLMANN, P., RIXNER, S., AND COX, A. L. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conference (ATC)* (2008), pp. 15–28.

[46] WOJTCZUK, R. Subverting the Xen hypervisor. In *Black Hat* (2008). http://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf. (Accessed: May 2014).

[47] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. On the DMA mapping problem in direct device assignment. In *ACM International Systems and Storage Conference (SYSTOR)* (2010), pp. 18:1–18:12.

Selectively Taming Background Android Apps to Improve Battery Lifetime

Marcelo Martins
Brown University

Justin Cappos
New York University

Rodrigo Fonseca
Brown University

Abstract

Background activities on mobile devices can cause significant battery drain with little visibility or recourse to the user. They can range from useful but sometimes overly aggressive tasks, such as polling for messages or updates from sensors and online services, to outright bugs that cause resources to be held unnecessarily. In this paper we instrument the Android OS to characterize background activities that prevent the device from sleeping. We present Tamer, an OS mechanism that interposes on events and signals that cause task wakeups, and allows for their detailed monitoring, filtering, and rate-limiting. We demonstrate how Tamer can help reduce battery drain in scenarios involving popular Android apps with background tasks. We also show how Tamer can mitigate the effects of well-known energy bugs while maintaining most of the apps' functionality. Finally, we elaborate on how developers and users can devise their own application-control policies for Tamer to maximize battery lifetime.

1 Introduction

The accelerated growth of sensing, computational, storage, and communication capabilities of mobile devices has enabled a rich application environment that rivals the performance of desktop computers. Even so, battery technology has not followed the same advancement pace and there is little evidence that this situation will dramatically improve. As a result, battery lifetime has become a major usability concern, with users willing to enjoy the latest apps on their smartphones and tablets, but, at the same time, worrying that their battery will not last long enough.

Since the inception of mobile computing, both industry and academia have developed a slew of techniques to reduce power at the architecture [8, 24], OS [49, 43] and application levels [16, 21], and today's systems draw little power while idling. Due to its user-centric and interactive nature, the flow of a mobile application is driven by events such as user actions, sensor I/O, and message exchanges. Such event-driven paradigm lets the system idle until a new event arrives. Mobile OSes, such as Android, iOS, and Windows Phone, take advantage of such idling opportunities to engage in *opportunistic suspend*. Upon brief periods of idling, the handheld switches to the default suspend state. Hardware blocks, including the CPU, GPU, GPS, and network modem, shift to low-power mode and software state is kept in self-refreshing RAM. The same blocks return from suspension upon interrupts emitted by hardware or software indicating that they have pending requests. With the rise of multitasking and the multiplication of background services and complex mobile applications, we expect this amount of interrupts to increase, forcing the system to spend more time *active* to attend requests. Such active periods take a toll on battery lifetime. A recent study by Google clearly shows this impact: each second of active use of a typical phone reduces the standby time by two minutes [18].

This paper studies the problem of battery drain mostly due to app-originated background operations that wake up the mobile system. We present Tamer, an OS mechanism we built for Android that interposes on events and signals responsible for task wakeups – alarms, wakelocks, broadcast receivers, and service invocations. Like a number of profiling tools, Tamer allows us to characterize the background behavior of different apps installed on a device. In §2, using Tamer's instrumentation, we show how a set of installed applications can dramatically affect the battery lifetime of four different devices. Unlike existing profiling tools, however, Tamer can also selectively block or rate-limit the handling of such events and signals following flexible policies. In this way, Tamer can, for example, limit the frequency at which an application schedules alarms or receives notifications of specified events, providing fine-grained control over the energy usage of apps that may be useful, but are irresponsible or inefficient with respect to their background activities. In §5.2 we show via a few case studies how Tamer can reduce the energy consumed by energy bugs [35] in legitimate apps. We summarize our contributions as follows:

- We characterize how applications and core components of the Android OS use specific features to enable background computing, and how this computing significantly affects energy use. In special, we note that Google Mobile Services play a major role on battery drain while the device is dormant (§2).
- We introduce Tamer, an OS mechanism to control the frequency at which background tasks are handled, thereby limiting their impact on energy consumption (§4). Tamer leverages code-injection technology and is applicable to any Java-based Android application.
- We demonstrate how Tamer can successfully throttle the background behavior of popular applications, thus reducing their energy footprint (§5). We show

how different policies reduce power draw in exchange for little to no impact on functionality.

Despite being a powerful mechanism, Tamer is only a step towards effective control of background energy usage. In particular, there are still challenges in helping users define policies that are effective, yet not disruptive to the user experience. We discuss such challenges in §7.

2 Motivation

Smartphone and tablet users are used to being always connected, expecting immediate notifications of a new e-mail or application update. Other common background operations include polling navigational sensors for location clues and turning on the network radio for incoming messages. Especially in Android, where there is little restriction on what apps can do in the background and developer's discipline is the main factor preventing inefficient applications, apps can hog resources and waste energy.

Traditionally there has been little visibility, both to app developers and to users, on the contribution of individual apps to energy use, especially while in the background. It is even harder to know whether an app is actually running or idling. Recent monitoring and profiling tools have helped bridge this visibility gap [9, 39, 19, 36, 30, 33, 45], as one cannot optimize what cannot be measured.

Today's average handheld contains a large amount of third-party software. A 2013 survey shows a global average of 26 apps installed on a mobile device [20]. Even with the best currently available tools, the end user can do little to cope with inefficient apps. Most of the tools above target developers and provide little help for the user. Even the friendlier ones, such as eStar [30] and Carat [33], when highlighting energy-inefficient programs, can only offer to kill or uninstall the culprit app, perhaps suggesting replacements. Unfortunately, this is too coarse-grained a solution and some apps with irreplaceable functionality become an inconvenience one has to live with.

Tamer offers the possibility of much finer-grained control once an energy hog or bug is found. It provides information on which tasks are expending the most energy and can rate-limit their execution. Tamer detects most causes of device wakeup that are visible at the framework level of Android, and can filter their continuation in real time.

To demonstrate the significant difference that a set of running tasks can make in a device's battery life, we measured the battery drop of four Android devices (two smartphones and two tablets, cf. Table 1) running two different application sets, *while idling and with the screen off*. Conservatively, we consider three scenarios: the first testing environment (Pure AOSP) consists of a stripped version of the Android Open Source Project (AOSP) OS containing a minimum number of services and apps; the second one adds Google Mobile Services (GMS) on top of Pure AOSP. GMS consists of proprietary applications and services developed by Google, such as Calendar, Google+ (social media), Google Now (personal assistant), Hangouts (instant messaging), Maps, Photos, Play Service (integrating API), Play Store, and Search. Due to their popularity and added value, GMS apps are included in most Android devices sold today. For the third scenario, which we only ran on the Galaxy Nexus phone, we also installed the ten most popular free apps of Google's Play Store as of January 2015[1]. We based all environments on the KitKat (4.4) version of Android. For the experiments, we left each device unattended running with its configuration at default settings. Other relevant settings include connection to a WiFi access point, enabled location-reporting, and background network synchronization. We expect most of the battery drainage to stem from static-voltage leakage and eventual background processing.

Figure 1 shows the time taken by each environment-device combination to deplete the battery. For all devices, Pure AOSP took the longest to completely drain the battery. In the case of tablets, this difference spanned dozens of hours. To investigate why this happened, we instrumented the Android software stack to timestamp the occurrence of background events. Additionally, we connected one of our devices (Galaxy Nexus) to a Monsoon power monitor [31] and collected power traces from the battery. Finally, we aligned and synchronized both the event and power timelines to understand their correlation. Figure 2 depicts a six-minute slice of this combination. We observe that GMS triggers more events in the background and that they are correlated with the surge of power peaks. We used this tracing knowledge to build a mechanism that counters the energy effect due to excessive wakeups. Because this mechanism relies on OS internals, we first need to understand how an Android app functions while in the background.

3 Background

This section provides a concise description of Android's power-management system followed by an overview of the components constituting a mobile app and how applications behave while running in the background. Finally, we highlight the influence of background execution on battery drain using four types of events: wakelocks, services, broadcast receivers, and alarms. §4 describes Tamer, our control system that adjusts the frequency at which such events occur.

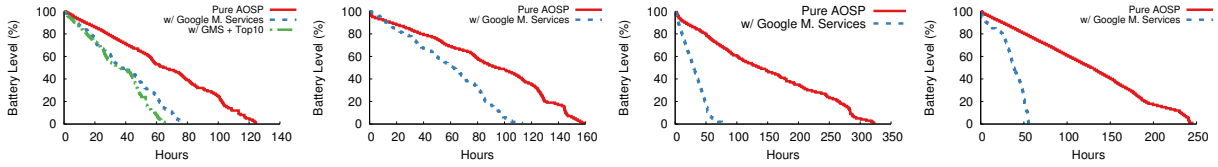
3.1 Mobile Power Management

Android employs an aggressive form of power management to extend battery life. By default, the entire system suspends itself, sometimes even when there are processes running. Opportunistic suspend is effective in preventing programs from keeping the system awake and quickly

[1] Crossy Road, Candy Crush Soda Saga, Pandora Radio, Trivia Crack, Snapchat, Facebook Messenger, Facebook, 360 Security Antivirus, Instagram and Super-Bright LED Flashlight.

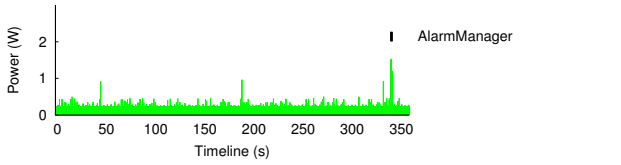
Device Name	Device Type	Processor	Features
Google Galaxy Nexus	Smartphone	Dual-core 1.2GHz ARM Cortex-A9	WiFi, GPS+A-GPS, 3G
Samsung Galaxy S4	Smartphone	Quad-core 1.9GHz Qualcomm Krait 300	WiFi, GPS+A-GPS, 3G/LTE
Amazon Kindle Fire 2	Tablet	Dual-core 1.2GHz ARM Cortex-A9	WiFi
ASUS MeMO Pad 7 (ME176C)	Tablet	Quad-core 1.83GHz Intel Atom Z3560	WiFi, GPS+A-GPS

Table 1: List of devices used for battery-drop monitoring.

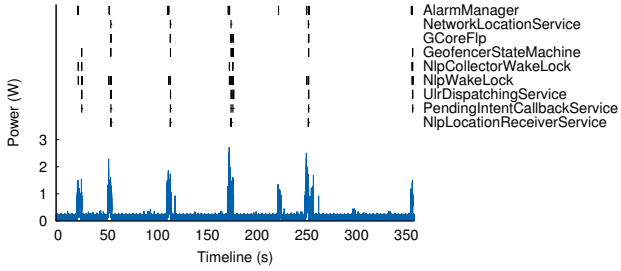


(a) Galaxy Nexus (b) Galaxy S4 (c) Fire 2 (d) MeMO Pad 7

Figure 1: Battery drop of four Android devices idling with the screen off. With Google Mobile Services installed, battery life decreased to 29.5% (Fire 2) and 77.5% (MeMO Pad 7) of its initial decay (without GMS).



(a) Pure AOSP



(b) Pure AOSP + GMS

Figure 2: Six minutes sampled from measurements on the Galaxy Nexus for two scenarios. For each graph, the top stack shows different event occurrences over time. The bottom curve depicts the corresponding system's power draw during the same period.

draining the battery. To curb system suspension, Android uses `Wakelocks` to keep the system awake. Wakelocks are reference-counted objects, similar to concurrency locks, that can be acquired/released by kernel and privileged userspace code. A wakelock acquire expresses a process's need for the system to remain awake until run completion. A wakelock acquire either holds a resource awake until a release call occurs or sets up a timer to relinquish the lock at a later time.

Kernel drivers use wakelocks to block the suspension of different system components (e.g., CPU, network, screen), whereas the Android application framework leverages wakelocks for different levels of suspension, represented by groups of components (e.g., keep the network radio awake vs. keep the radio, screen, and CPU awake). As an example of suspend-blocking by the OS, Android automatically acquires a wakelock as soon as it is notified of an input event and only releases the wakelock once some application handles the event or there is a timeout. Application developers can also directly instantiate and manipulate wakelocks using the Wakelock API. A proper e-book reader app must acquire a wakelock to keep the screen awake so that the user can read her favorite novel without interruption. Wakelocks play an important role in guaranteeing proper background task execution in face of default suspension, as we will see next.

3.2 Android Applications: Dealing with Lifecycle Changes

Barring a few interface-less system processes, an Android application consists of a set of `Activities` that places the UI widgets on the screen. An application starts with a single thread of execution attached to the *foreground* UI, which is mostly responsible for dispatching interface events. To avoid app unresponsiveness and user frustration, a wise programmer would move other computations to concurrent worker threads while the UI responds to input events. Support for concurrency comes in the form of a number of standard Java primitives, such as `Threads` and `Futures`, as well as Android's own flavors: `AsyncTasks` and message `Handlers`. However, such primitives only work when the application is in the *active state*.

As the user navigates through, out of, and back to an application, its lifecycle transitions between different states according to activity visibility. An application is active if one of its activities receives user focus in the foreground. If the user switches to another app or decides to turn off the screen, the application is paused and moved to the background. Because mobile apps are multitasked, developers must have a way to run code even when their app is not occupying the screen.

3.2.1 Dispatching Background Tasks

The small screen size of a smartphone or tablet prevents multiple applications from running simultaneously. To conserve energy, apps are frozen and stop working once sent to the background, either due to the opening of another application or a screen timeout. Context switching opens room for opportunistic suspend – by making apps invisible, the Android OS frees its own set of wakelocks, opening space for hardware throttling.

Android offers application developers a small and well-defined interface for background-task offloading that takes care of scheduling latent tasks [2]. This interface comprises a handful of components including services, broadcast receivers, and alarms. The internal implementation of such components also leverages wakelocks to keep the device awake while executing tasks.

`Services` are application components that run asynchronously on background threads or processes and do not directly interact with the user. Instead, `Activities` dispatch services to perform long-duration operations or to access resources on behalf of users, such as downloading remote files or synchronizing data with a cloud-based storage. An advantage of running services separately is that their running persists even after closing the owner's interface. Apps and widgets rely on services being operational without need of manual restart.

`BroadcastReceiver` is a reactive mechanism that permits programs to asynchronously respond to specified events. An application registers a `BroadcastReceiver` along with an event-subscription list – the `IntentFilter` – that is used to determine if the application is eligible to respond to a given event. Events can be predefined by the system (e.g., "battery fully charged") or developer-defined (e.g., "backup finished"). Receiver threads remain dormant until a matched event arrives and respond by running a callback function. A file-hosting app could, for instance, register a receiver to display a notification box once it discovers that a scheduled data synchronization has finished.

Another common programming pattern is the ability to perform time-based operations outside the lifetime of an application. For instance, checking for incoming e-mails every so often is a recurrent user operation that could be automatized. The Android SDK offers developers the `AlarmManager` mechanism to fulfill the scheduling of periodic tasks at set points. At each alarm trigger, the system wakes up and executes the scheduled callback function, whose contents can take various forms: a UI update, a service call, an I/O operation, scheduling a new alarm, etc. Alarms are a good fit for opportunistic suspend: apps are only activated when there is work to do.

In summary, Android uses at least three types of asynchronous mechanisms to perform background tasking: services, broadcast receivers, and alarms. Aligned with wakelocks, we have a powerful collection of events that can keep the system awake. In the next section, we introduce Tamer, a system that acts on this small and well-defined interface to throttle the rate at which background events are handled in exchange for energy savings.

4 Tamer

4.1 Design

Having seen that background events can noteworthily affect the sleeping pattern of mobile devices, we consider the possibility of regulating their frequency to improve battery life. We introduce a configuration mechanism for declaring thresholds to the frequency of these events. We model this regulation process using three sequential steps: (1) observation; (2) comparison; and (3) action.

Event-frequency regulation works based on the specification of occurrence limits. We establish a policy mechanism that lets users define how often the running system should permit a given background event to proceed. A policy is a contract that declares the conditions for an event execution. This contract specifies the event type as well as its identifier; an optional list of affected apps, in case we want to restrict such enforcement to a subset of event dispatchers or receivers; whether the policy enforcement also takes place when the event owner (app) is in the foreground; and the rate at which they are allowed to execute. From this definition, an energy-savvy user could program her smartphone to permit calls to `WeatherUpdateService` from a weather-forecast app at most once every six hours, whereas calls to `LocationUpdateService` from the same app would remain unlimited.

To enforce user-defined policies, we outline a controller comprising three agents: observer, arbiter and actuator. The *observer* intercepts every event occurrence and bookkeeps its frequency. The *arbiter* verifies whether the measured event rate is above the policy-defined threshold, if it exists, and notifies the *actuator*, which hijacks the event continuation to artificially reduce its occurrence rate. Figure 3 illustrates our control sequence.

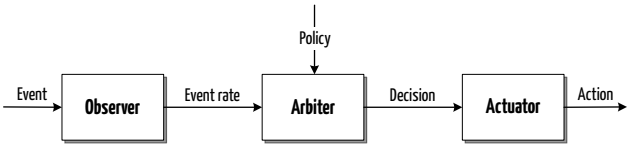


Figure 3: Sketch of our event-control system as a three-stage pipeline.

There are two ways to fulfill event throttling: canceling or delaying. An event cancel denies the continuation of its call, with an early return that prevents the payload or the event callback from running. An event delay, on the other hand, postpones the continuation of the event call for a limited time. In this work, we opt for canceling the event handling. We discuss the pros and cons of each choice in §7. It is important to understand that an event cancel

does not result in a crash. Alarms, services and broadcast receivers all run asynchronously. Wakelock requests, on the other hand, are synchronous and their denial will result in a sleeping system when a waiting task is expected to run. Still, the task is never aborted, but runs in chunks when the system wakes up. Our method prevents the triggering of unwanted events. When it is not possible, we abort the *event continuation* at the earliest opportunity to reduce the energy cost of the event payload.

To drive the implementation of our system, we establish a few requisites and considerations:

Comprehensive support for events. The control system should be inclusive. While app-specific solutions are effective, they do not scale to other programs. The stage pipeline should monitor and, if necessary, actuate on every event instance. Particulars about a specified event should be confined to its policy and not affect the controller. It is the responsibility of the policy designer to define a sane event frequency, considering, perhaps, the context and the impact of an event hijack. To implement an all-encompassing monitor system, we target an OS-level solution.

Support for power-oblivious applications. Users should not abstain from using their favorite apps even if they are power hogs. Uninstalling or suggesting alternative apps for the same purpose should not be acceptable. The system should cope with the existence of ill-behaved apps and act upon their misbehavior if directed by the policy designer.

Compatibility. Many solutions that rely on deep system introspection require extensive rewrites of system components [11, 14, 6] or even writing systems from scratch [43]. Although tempting, straying from the mainline can severely limit the userbase, especially in the case of consumer-oriented OSes. With that in mind, our solution should exhibit high compatibility and keep a minimum amount of changes to the underlying OS.

Efficiency. Mobile apps must cope with limited computational and energy resources. The control system should avoid high computational overhead to prevent high battery drain and system slowdown.

4.2 Implementation

To avoid reimplementing OS components to regulate event handling, Tamer uses the Xposed framework [42] to enable system modifications at runtime.

While requiring the device to be rooted, Xposed enables deep system modifications with no need to decompile applications nor flash the device[2]. Xposed intercepts Java methods and temporarily diverts the execution flow to function-hook callbacks for inspection and modification. Developers define these callbacks and compile them as separate modules. Function-call hooking happens by matching the method's name and signature of the declared callback with the running code. Callbacks run on the context of the intercepted application. Xposed allows for changing the parameters of a method call, modifying its return value or terminating it early. We leverage the hooking mechanism to intercept function calls originating from or directed at our events of interest. Finally, function hooks can be distributed as separate programs in self-contained APK files. They are not bound to a specific Android version and work without changes on the majority of customized Android releases, including those from Samsung, HTC, Sony, LG, and the CyanogenMod open-source community [10].

Figure 4 shows how Tamer relates to the Android OS. Tamer sits, along with Xposed, between user applications and the Java-based application framework, which serves as the foundation for the Android SDK. Events have directions, which helps us define how to write the interception payload. While service and wakelock calls originate from apps and are forwarded to the framework, alarms and broadcast receivers work in the opposite direction.

Tamer consists of a series of function hooks that interpose on the background-processing interface and act as a controller mechanism to enforce user-defined policies. To implement Tamer's event-canceling mechanism, we leverage Xposed's introspection API to explore, monitor, intercept and modify public and private classes, methods and members of the framework (Table 2). We used our knowledge on the Android SDK aligned with the source code of Android's framework stack to decide where to place the instrumentation points that would constitute our controller. We analyzed the source code stemming from each event call on the SDK's public interface. Modeling the relationships between subroutines as a call graph, we considered each interface function as a leaf node. In some occasions, we had to backtrack the call graph to find a proper instrumentation point. This was necessary for three reasons: (1) the public interface did not offer enough context to feed our monitoring system (e.g., missing receiver name, unclear caller-callee relationship, etc.); (2) in the case of receiving events, it was better to interpose on a call as early as possible to avoid unnecessary operations before a cancellation; (3) an event call may have more than one function signature, therefore we looked for a converging function node. We found one exception to the last rule when handling broadcast receivers. Applications can declare receivers in two ways: statically via a `Manifest` file or dynamically using the Android SDK API. Since the Android framework keeps separate data structures for each case, we had to instrument them separately.

Tamer's interception can suppress wakeups due to service invocations, wakelock acquires, and intent broadcasts. Because of the way Android handles alarms, our implementation can only curtail the alarm's callback payload. The system will still periodically wake up according to the alarm's schedule, but will immediately return to sleep. Our

[2] For brevity reasons, we refer readers to [41] for an explanation on how such thing is possible.

Event	Class	Method	Instrumentation Payload
Wakelock	com.android.server.PowerManagerService	acquireWakeLockInternal	Search for policy. If found, early return in case call happens before grace period. Else, let acquire proceed; bookkeep wakelock and start grace-period timer.
		releaseWakeLockInternal	Only called if there was no block; report how long wakelock was held.
Service	com.android.server.am.ActivityManagerService	startServiceLocked	Search for active policy. Proceed as in wakelock case.
BroadcastReceiver	android.app.ContextImpl	registerReceiverInternal	Add reference to API-registered receiver.
		unregisterReceiverInternal	Remove reference to API-registered receiver.
	com.android.server.pm.PackageManagerService	addActivity	Add reference to receiver registered statically.
		removeActivity	Remove reference to receiver unregistered statically.
	com.android.server.am.ActivityManagerService	broadcastIntentLocked	Search for active policy. Temporarily remove receiver from framework index to prevent event broadcasts. Update stats.
Alarm	com.android.server.AlarmManagerService	triggerAlarmsLocked	Search for active policy. Proceed as in wakelock case.
GPS (See §5.3)	android.location.LocationRequest	requestLocationUpdates	Enable GPS throttling for calling app.
		removeUpdate	Disable GPS throttling for calling app.
	com.android.server.LocationManagerService	reportLocation	Search for active policy. Let callback report incoming location, but temporarily switch off GPS sensor for blocking period.

Table 2: A brief description of TAMER's instrumentation points.

controller implementation covers all versions of Android ranging from Ice Cream Sandwich to KitKat[3]. In a handful of occasions, we resorted to different instrumentation points for a given event, mostly due to small differences in the function signatures between OS versions. Because the framework interface is fairly stable, covering future versions of Android should not require major changes.

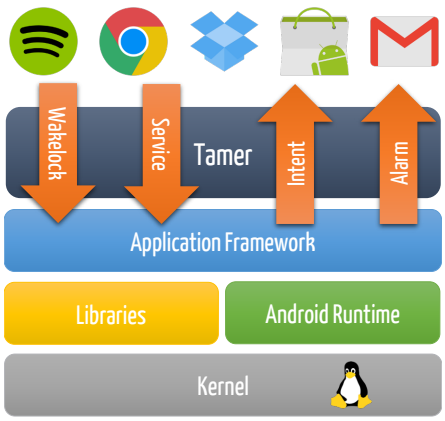


Figure 4: TAMER sits between apps and the framework stack and interposes on events between these two. Lower system layers are oblivious to our system.

5 Evaluation

We evaluate TAMER in four ways. First, we revisit our motivating scenario (§2) and use TAMER to extend the battery life of the GMS-based installation. We then investigate how TAMER can effectively mitigate energy bugs, a system behavior that causes unexpected heavy use of energy not intrinsic to the desired functionality of an application. Next, we show how to use code injection to create specialized versions of controllers for situations that our four-event toolset cannot handle. Last, we measure the overhead caused by TAMER on performance and energy.

5.1 Dealing With Google Mobile Services

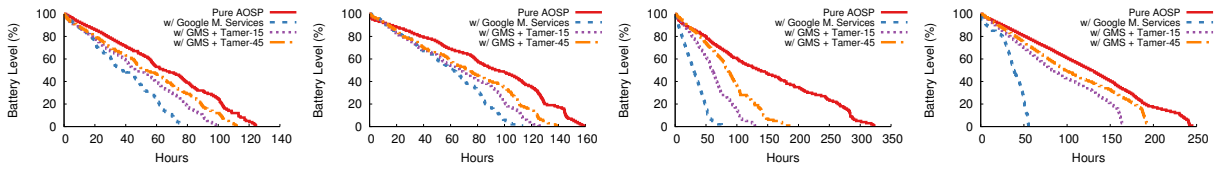
In §2, we saw how the inclusion of GMS into the baseline AOSP significantly reduced the battery life of all tested devices. Nonetheless, GMS adds a series of services and applications that truly enhance the user's mobile experience. In fact, most users do not even have the option of uninstalling them, as GMS comes pre-installed as a system package in the majority of handhelds. We show how TAMER can reach a tradeoff between GMS's functionality and battery savings. We aim to keep the added value of GMS without the cost of a silent battery depletion.

Event Name	Type	Count	Duration (s)
NlpWakelock	W	5963	1662.71
NlpCollectorWakelock	W	2121	3926.63
LocationManagerService	W	2030	67.12
NlpLocationReceiverService	S	1159	-
NetworkLocationService	S	579	-

Table 3: Top event occurrences for the Galaxy Nexus' battery drain due to GMS. A handful of events are responsible for the major impact on the battery. W signifies a wakelock event, whereas S stands for service invocation.

With the control mechanism established, our next

[3]This limitation is due to Xposed's limited OS support. Support for Android's latest release (Lollipop) is not stable enough to cover our needs.



(a) Galaxy Nexus (b) Galaxy S4 (c) Fire 2 (d) MeMO Pad 7

Figure 5: Battery drop for the four devices while idling with the screen off. After applying our policies to GMS, battery life improved in dozens of hours for all devices.

step is to design a policy that reduces the battery impact of events originating from or destined to GMS. Table 3 ranks the top triggered events reported by Tamer's monitoring module. For wakelocks, we also report the time they were held. We use event frequency as an heuristic to guide policy configuration. We note that `NlpCollectorWakelock` was primarily responsible for keeping the system awake in the background. Online reports [40, 5] indicate that `NlpCollectorWakelock` is related to `Location Reporting`, an Android feature to estimate and report the current location based on WiFi APs and cell-tower signals. Apps that use this feature include Google Now and Google Maps, among others. The other frequently reported events are also related to the same feature. Disabling location reporting on the device's Settings menu would be a logic solution to increase sleep time, if dependent apps did not stop working.

The problem with `NlpCollectorWakelock` and associated events is the frequency they wake up the system and keep it awake, which sums to a substantial period of non-sleepiness. During a discharge period of 80 hours for the Galaxy Nexus smartphone, `NlpWakelock` was called once a minute on average, whereas `NlpCollectorWakelock` contributed to keeping the system awake for more than one hour. Such a high battery impact coming from a single package does not justify the benefit of having GMS running as it is in the background. For this reason, we devised two policies for GMS, targeting `NlpCollectorWakelock` and its associated events, to alleviate this wakeup burden. For each wakelock and service in Table **??**, the first policy (Tamer-15) allows a single call every 15 minutes. The second policy (Tamer-45) allows one call every 45 minutes. Deciding on an appropriate rate is a subjective matter. Our setup tries to reach a balance between informing subordinate apps of location updates and increasing battery lifetime. Figure 5 shows that our policies substantially reduced the battery-drain rate of all tested devices.

5.2 Chasing Energy Bugs

An energy bug, or *ebug*, is a system error either in an application, OS, firmware or hardware that causes an unexpected amount of high energy consumption [35]. Such errors occur due to a variety of reasons such as programming mistakes, faulty hardware, malicious intent, etc. Because such errors may not result in a crash, users will only notice their effect when it is too late: an early dead battery.

Differently from previous research which identified and characterized energy bugs [46, 37], in our evaluation we focus on mitigating them at runtime. Tamer allows users to run the offending applications without the adverse effects of the bugs. Finding ebugs is not trivial and the lack of a centralized repository of updated samples prevents us from testing our controller more extensively. We successfully reproduced and circumvented three application ebugs described in [23]. For the other described cases, we could not confirm the existence of bugs. We assume these defects have been fixed by developer updates.

Next, we present two detailed case studies of new ebugs that we found. To identify them, we used eStar [30], a tool that ranks the contribution of apps to battery depletion. We first selected apps that display poor energy efficiency and filtered them based on high popularity at the Google Play Store (our two selections featured on the top-20 ranking of their respective categories: Games and Health & Fitness). Although eStar ranks energy-inefficient apps, we still had to manually verify whether such inefficiency was due to foreground or background activity. For each application, we simulated a user interaction consisting of a short-length active session followed by a long period in the background. **Bejeweled Blitz** [13] is an award-winning puzzle game with over 10 million installs from Google Play Store. After a 15-minute play session on the Galaxy S4 smartphone, Tamer reported that our game generated a single background event – the acquire of the `AudioIn` wakelock. Because games are resource-hungry apps, this event call initially did not instigate any suspicion. We discovered a red flag, though, after switching Bejeweled to the background: `AudioIn` was not released after the game suspension. To mitigate this bug, we wrote a simple policy targeting Bejeweled Blitz that blocks the renewal of the culprit wakelock during background time. Figure 6 depicts the battery drop of a 12-hour session with Bejeweled loaded in the background before and after applying our policy. We see a 4× improvement on battery drain. Figure 7 confirms the effect of releasing the ill-behaved wakelock: before being *tamed*, the smartphone spent approximately 95% of the time with an awake CPU. With Tamer's interposition, most of the residency ratio was converted to deep sleep.

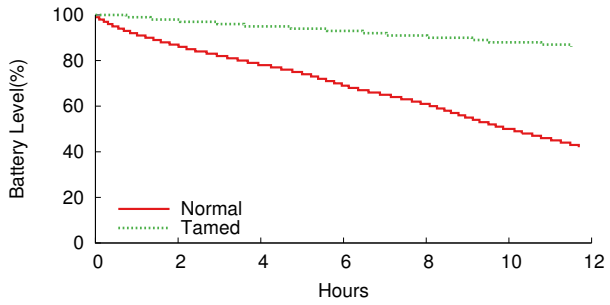


Figure 6: Battery drain over 12 hours for Bejeweled Blitz without TAMER (Normal) and with a TAMER policy blocking the `AudioIn` wakelock. In both cases, the game was started and the phone switched to idle mode with the screen off.

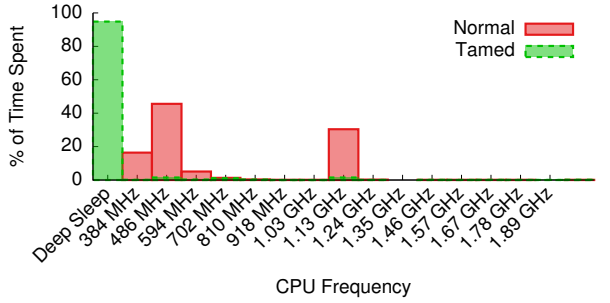


Figure 7: CPU residency time for the original and tamed versions of Bejeweled Blitz on the Galaxy S4. The tamed version spent 94% of the time in Deep Sleep.

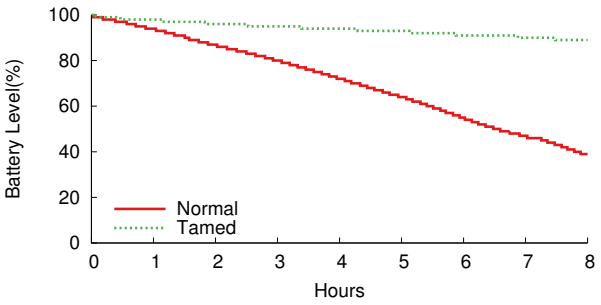


Figure 8: Battery drain over 8 hours for Nike+ Running without TAMER (Normal) and with a TAMER policy that blocked all five held wakelocks. In both curves, the app was started and the phone switched to idle mode with the screen off.

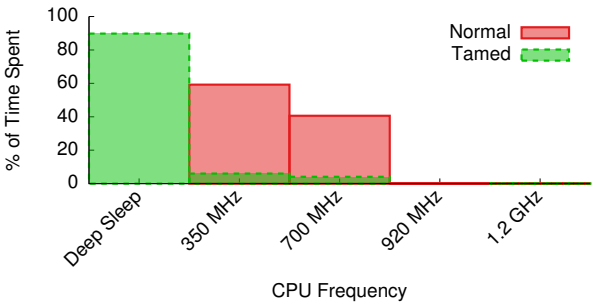


Figure 9: CPU residency time for the original and tamed versions of Nike+ Running on the Galaxy Nexus. The tamed version spent 89.4% of the time in Deep Sleep.

Nike+ Running [32] is a fitness app for tracking runs. Nike+ relies on the GPS sensor, the accelerometer, and barometer to estimate distance and speed. According to TAMER, Nike+ acquired five wakelocks while running: `AudioMix`, `FullPower Acc Sensor`, `FullPower Pressure Sensor`, `FullPower Recording` and `NlpWakeLock`. Judging from the wakelock names, we can assume a few hardware components remained awake to prevent device-sleeping. We found an ebug when pausing our running session and switching Nike+ to the background. In this case, we expected the application to release all wakelocks. Like Bejeweled Blitz, Nike+ forgot to relinquish the locks upon leaving the foreground. Our policy was also equivalent: block the culprit wakelocks during background time. Figure 8 shows the battery drop after an eight-hour session before and after applying our policy. We see a 5× improvement on battery drainage. Figure 9 displays the CPU residency on both scenarios: CPU deep-sleep residency jumped from a 0% to 89.8%. We also acknowledge a major contribution of the GPS and sensors to battery decay. Their duty cycle is equivalent to the time the homonymous wakelocks were held.

5.3 Looking at the Other Side of Events

TAMER can block events at their imminent arrival or dispatch, thus saving energy that would be consumed by their continuation. One can say that TAMER focuses on the *effect* of an event. Still, there are situations in which the energy cost of its *cause* is significant. A user could write a policy targeting an instant messenger's broadcast receiver to throttle the effect of a message arrival – a notification in the form of sound or vibration. However, she cannot block the message arrival itself, the major contributor to energy consumption in this case, as the remote sender is not covered by TAMER.

Acting on the cause of an event is complicated because its originator, when known, can take various shapes, like a disk or network I/O operation. Consequently, finding a converging instrument point in the source code that encompasses all such shapes is complex. In parallel, we should avoid point solutions that only fit one app. Between these two extremes, we can reach a balance by instrumenting code that abstracts common functionality and generates events used by a subset of applications. We consider the case of navigation apps to illustrate such scenario.

Throttling Localization. Android applications that rely on the GPS sensor follow a basic model: (1) they register a position listener and (2) they periodically receive location updates from a provider proxy, the only interface to the localization system [3]. A provider proxy serves as an interface to various location sources, including the GPS sensor, WiFi APs, and cell towers.

Given that the GPS is an energy-hungry resource, we consider a throttling mechanism for its duty cycle. Paek et al. [34] successfully demonstrated the potential energy savings of duty cycling by creating a rate-adaptive positioning system that switches the GPS sensor on and off and uses alternative location sources based on position accuracy. As a demonstration, we consider a simpler GPS-throttling implementation sans secondary sources. An advantage of our approach is the dispensing of OS recompilation, keeping it compatible with the majority of Android devices. The GPS sensor provides periodic position fixes (every second) to the OS. Some of these fixes are not relayed by the Android framework to the navigation app as they are not significantly different. We can reach a more energy-efficient navigation by directing the GPS duty cycle. We inject code into internal classes related to the framework's GPS provider and open a direct communication channel between the GPS device and our throttling mechanism. This direct channel permits our controller to switch the GPS on and off. Table 2 summarizes our instrumentation.

Evaluation. We consider two location-based apps, Google Maps and Nike+ Running. We ran these two applications separately on the Galaxy Nexus phone and applied three different throttling policies to the GPS duty cycle: location updates every one, five and fifteen seconds. The rate choice depends on the user's purpose. For pedestrian navigation, a slower update rate does not affect the estimated position as much as in the case of a highway car trip. Paek et al.'s work includes a thorough tradeoff analysis between position accuracy and energy savings. We, on the other hand, only report the potential savings. For each scenario, we programmed a pedestrian route lasting ten minutes. We turned the screen off while running the application in the background. Because of the small timespan, we compare the energy dispensed instead of battery-level drop. We used the Monsoon monitor to measure the energy consumption. Figure 10 portrays the savings per application. We observe an upper bound of 27.7% on savings for the Nike+ Running application when reducing the location-update rate to 1:15s. Although a lower update rate increases the energy savings, position accuracy is penalized.

5.4 Performance Impact

Just like a network firewall, Tamer intercepts every event-happening, inspects it, evaluates the corresponding policy criterion, and finally actuates to fulfill the policy's conditions. Because Tamer diverts the normal flow of applications, it should incur as little performance overhead and energy burden as possible. We instrumented Tamer to measure the time taken to hijack an event and perform its blockage. For the longest diverted execution flow, Tamer took, on average, 320μs to execute on the Galaxy Nexus device. With regards to energy consumption, Tamer is activated only when other applications generate events.

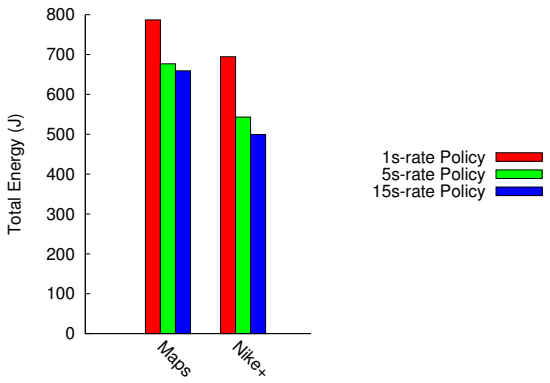


Figure 10: Total energy consumed by Google Maps and Nike+ Running after applying three different duty-cycle policies to the GPS provider. For both apps, the update rate is inversely proportional to energy savings.

Tamer does not acquire any wakelocks, but freeloads the system's active state from other wakeup sources.

6 Related Work

We are not the first to propose control of functionality in exchange for battery savings. Tamer builds upon a number of contributions to mobile power management.

Collateral Related Work. Efficient power management in mobile platforms is a challenging research problem due to the multitude of hardware configurations and power states. To improve energy consumption, we need to understand how hardware components draw power on behalf of applications. There is a myriad of tools that help quantify a device's energy expenditure. PowerScope [17] is one of the first works in the mobile domain to map energy to a program's structure. PowerScope employs linear regression and statistical sampling to apportion energy to hardware and applications. A series of recent profilers for smartphones complements PowerScope, including PowerTutor [50], ARO [38], AppScope [22], WattsOn [29] and eprof [36]. Sesame [12] and V-edge [47] go a step further and propose self-calibrating models that dispense the use of external power monitors, relying instead on internal battery data to model energy expenditure. Our analysis mainly adopts battery-drop rate as a proxy for energy consumption, but we expect such advanced contributions to be integrated by OEMs in future devices.

Saving Energy From Background Tasking. Android task killers once were the solution for background power savings, but their effectiveness is now a point of contention [27]. Task killers force background applications to quit, assuming that their removal from memory will reduce the energy footprint of released resources. Such assumption is incorrect as there is little correlation between memory and CPU usage in Android [4]. Excessive task killing may lead to the opposite effect: by discarding cached data, Android must reload apps from storage. A

killed app may restart itself immediately after being killed, using up CPU time and draining even more battery.

Rather than killing background tasks, popular apps like JuiceDefender [26] and Easy Battery Saver [1] can configure the access to power-greedy components, such as the radio and GPS, on a schedule basis. Although effective in many cases, some apps do not behave properly when they cannot, for instance, connect to the Internet. Some apps may even produce more energy overhead as they insist on accessing a resource made unavailable. Tamer circumvents such problems by mainly throttling asynchronous actions: the expecting app does not block while waiting for an event arrival or dispatch. Greenify [15] is an Android tool for hibernating apps, preventing the arrival or dispatch of events once an app switches to background. The original functionality is only restored when the app returns to the foreground. Greenify is effective in blocking misbehaving and start-at-boot apps, but its treatment of background computing is coarse and not applicable to notification-based apps that mostly run in the background (e.g., mail readers, instant messengers, calendars, etc.) Tamer is applicable in such cases as it throttles, but does not completely eliminate, background functionality.

Android also includes its own controls for background-task management. The `AutoSync` feature controls the automatic data synchronization between client apps and online accounts. With a mere tap, users can choose between disabling the synchronization of a specific feature of a selected account (e.g., photo uploads for Google+) or totally prevent any background synchronization for all registered accounts. `AutoSync` is mostly applicable to apps adopting Google Cloud Messaging (GCM), an API piece from Google Mobile Services. GCM provides a lightweight mechanism that third-party servers can use to notify mobile applications of available content to be fetched. As long as the application is subscribed to receive GCM messages, the Android device does not need to run continuously. Instant messengers, for instance, use GCM to receive notifications of new incoming messages. Tamer complements this service by offering a similar control to applications that do not adopt the GCM approach. Moreover, Tamer allows for the management of a variety of background events, whereas `AutoSync` focuses mainly on networking.

Partial inspiration for deep event monitoring stems from applications such as BetterBatteryStats [25] and Wakelock Detector [45]. Both apps report wakelock-usage statistics that developers can use to understand the root cause of battery drainage. Tamer complements such apps by empowering users to take action after they pinpoint the origin of abnormal energy consumption.

Carat [33] and eStar [30] use data collected from thousands of smartphone and tablet users to model the battery drainage of applications. By combining rich context information of multiple devices with energy awareness, it is possible to determine whether the energy used by an application deviates from its expected consumption. These tools are conservative in controlling energy expenditure, with both systems suggesting users to kill or uninstall culprit apps. eStar further recommends energy-efficient alternatives to power-hog apps, if they exist. Tamer let users keep their apps while modifying the culprit's behavior to reduce energy consumption.

7 Discussion

As any prototype, Tamer has limitations. Tamer's main utility comes from policy definition, which, at its current state, will not appeal to the end user. In the following, we elaborate on how to circumvent this usability issue. We also suggest improvements that are left as future work.

7.1 Policy Guidelines

In §5, we demonstrated how a wise policy selection can partially inhibit the surge of energy-hungry events. Our experience defining policies arose from intuition, reading source code (when available) and, in some cases, multiple attempts. At its current state, Tamer would better serve as a backend for higher-level power management tools than as an end-user app. With such limitations in mind, a user willing to run Tamer as it is, would benefit from the following guidelines to explore the event space and define effective policies.

Choosing events to control. Handhelds may carry tens or even hundreds of applications that generate thousands of events. We should not imply that policy definition must consider all of them equally. First, users prefer some apps over others. Second, event triggering does not follow a uniform distribution. As a rule of thumb, users should start with policies targeting the most frequent events. Tamer's monitoring module periodically outputs an event summary that can assist in such cases.

Cutting the red wire. Even after selecting the most prominent events for policy testing, there are no guarantees that the policy will work without side effects. Side effects may include an increase in the frequency of correlated events, the rise of unexpected events, and abnormal application behavior. Blocking alarm events recklessly could, for instance, totally defeat the purpose of a calendar app. Because most apps are only available in binary form, understanding the purpose of an event is not always clear and neither is uncovering its dependencies. Techniques used in black-box testing, such as cause-effect graphs can help. Events may also show a temporal correlation with others. To uncover temporal dependency, we generated event timelines from Tamer's monitoring output.

Use common sense. The Android OS defines two categories of applications: system and user. The former includes programs that are deemed critical, are deeply integrated into the OS, and cannot be uninstalled. Exam-

ples include the dialer, browser, and network manager, to name a few. Users apps are replaceable programs that can be freely removed and installed from the app store. As part of Tamer's design, we adopted the support of generic events. Consequently, system- and user-app events are treated equally. Policies that alter the frequency of system events may result in unwanted or abnormal behavior. Users should be mindful when defining policies involving critical events to avoid such situations. Removing support for system events would prevent such unfortunate occasions, but the definition of a system app is blurry. GMS, for example, comes pre-installed as a system package on many devices. Carriers also sell devices with *bloatware* installed as system apps. As demonstrated, systems apps present great opportunities for energy savings.

7.2 Potential Improvements

Event batching over cancellation. Our current implementation dismisses event continuation if there is a need for throttling. Alternatively, we could reschedule the asynchronous delivery of such events to coalesce multiple wakeups into one, saving even more energy. Although promising, event coalescing may lead to unexpected results that require deeper investigation. Some apps assume a fixed frequency of events. A pedometer may use the time difference between position fixes to estimate speed. Batching multiple fixes into one delivery may create havoc if the tracker does not discard outdated values. Nevertheless, coalescing has found its way in other domains. The Linux tickless kernel [44] reduced the precision of software timers to allow the synchronization of process wakeups, minimizing the number of CPU power-state transitions. From its Lollipop release, Android started to batch alarms that occur at reasonably similar times, turning them inexact. Xu et al.'s recent work on coalescing events to save energy in the context of email synchronization [48] is another successful example of careful event-handling for mobile devices. As long as developers do not assume guarantees on event delivery and commutativity, we believe coalescing should supersede cancellation as an energy-saving feature.

Native code support. Tamer controls applications by wrapping function calls from the Android Java API. Applications that make heavy use of native code, like games, multimedia apps and ELF libraries, could acquire wakelocks, spawn threads and perform background tasks using C/C++ code, thus bypassing our control system. Extending support to native code would require a similar effort on analyzing and instrumenting `libc` function calls.

Support for other mobile OSes. Background processing is not exclusive to Android, although handled differently by other mobile OSes. Apple's iOS 7+ regards background processing as a privilege [7]. Other than network transfers, common background tasks have limited time to completion and must respect the device's will to sleep, doing their processing in chunks after the device wakes up to handle phone calls, notifications and other interruptions. Windows Phone enforces background tasks to be lightweight by applying quotas to resources like CPU, memory, and network usage while apps are running behind the scenes [28]. Event-frequency control may not produce the same gains on Apple's and Microsoft's mobile devices given their stricter stance on deploying background tasks (mainly in the name of battery savings).

Feedback control. Tamer works as an open-loop controller, not using feedback to gauge whether the system needs more adjustments. During the design stage of this project, we discarded the closed-loop approach as it would require knowledge of application semantics as well as user perception of performance degradation. Modeling these two elements are hard problems beyond our scope.

8 Conclusion

This paper presented Tamer, an OS mechanism that interposes on task wakeups in Android and allows event handling to be monitored, filtered, and rate-limited. We demonstrated that Tamer substantially reduces the background energy use in popular Android applications. With Tamer, a device spends more time in low-power mode, which increases the battery lifetime significantly.

While this work shows Tamer's effectiveness as a mechanism, future work is needed to understand how to best construct policies that improve battery life while preserving application functionality. In future work, we will investigate techniques for determining if functionality is negatively impacted when exploring user visible elements (e.g., UI differences) between runs of an application with different policies. We will also explore which policies are most likely to have substantial battery savings in practice. With the combination of such techniques, we will strive to devise policies that improve battery life while retaining normal application functionality.

Acknowledgements

We thank Tim Nelson, Hammurabi Mendes, the anonymous reviewers, and our shepherd, Lin Zhong, for their feedback. Marcelo was funded in part by a generous gift from Intel Corporation.

References

[1] 2Easy Team. Easy Battery Saver. http://www.2easydroid.com.

[2] Android Developers. Best practices for background jobs. https://developer.android.com/training/best-background.html.

[3] Android Developers. Location strategies. https://developer.android.com/guide/topics/location/strategies.html.

[4] Android Developers. Managing your app's memory. https://developer.android.com/training/articles/memory.html.

[5] AndroidCentral.com. Google Services battery drain. http://forums.androidcentral.com/google-nexus-4/302559-google-services-battery-drain.html.

[6] Andrus, J., Dall, C., Hof, A. V. H., Laadan, O., and Nieh, J. Cells: A virtual mobile smartphone architecture. In *ACM SOSP'11*.

[7] Apple Inc. App programming guide for iOS. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html.

[8] ARM Limited. big.LITTLE technology: The future of mobile. http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[9] AT&T. Application resource optimizer (ARO). http://developer.att.com/application-resource-optimizer.

[10] CyanogenMod Community. http://www.cyanogenmod.org.

[11] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security'11*.

[12] Dong, M., and Zhong, L. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *ACM MobiSys'11*.

[13] Electronic Arts Inc. Bejeweled Blitz. http://play.google.com/store/apps/details?id=com.ea.BejeweledBlitz_na.

[14] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI'10*.

[15] Feng, O. Greenify. https://play.google.com/store/apps/details?id=com.oasisfeng.greenify.

[16] Flinn, J., and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *ACM SOSP'99*.

[17] Flinn, J., and Satyanarayanan, M. PowerScope: A tool for profiling the energy usage of mobile applications. In *IEEE WMCSA'99*.

[18] Gigaom. Google's killer Android L feature: Up to 36% more battery life thanks to Project Volta. http://gigaom.com/2014/07/02/googles-killer-android-l-feature-up-to-36-more-battery-life-thanks-to-project-volta.

[19] Google. Battery Historian. https://github.com/google/battery-historian.

[20] Google. Our mobile planet. http://think.withgoogle.com/mobileplanet/en.

[21] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., and Agarwal, Anant ad Rinard, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS'11*.

[22] Jung, W., Kang, C., Yoon, C., Dongwon, K., and Cha, H. AppScope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *USENIX ATC'12*.

[23] Kim, K., and Cha, H. WakeScope: Runtime wakelock anomaly management scheme for Android platform. In *ACM EMSOFT'13*.

[24] Kim, W., Gupta, M. S., Wei, G.-Y., and Brooks, D. System level analysis of fast per-core DVFS using on-chip switching regulators. In *IEEE HPCA'08*.

[25] Knispel, S. BetterBatteryStats. https://play.google.com/store/apps/details?id=com.asksven.betterbatterystats.

[26] Latedroid. JuiceDefender – battery saver. http://www.juicedefender.com.

[27] LifeHacker.com. Android task killers explained: What they do and why you shouldn't use them. http://lifehacker.com/5650894.

[28] Microsoft. Supporting your app with background tasks. https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh977056.aspx.

[29] Mittal, R., Kansal, A., and Chandra, R. Empowering developers to estimate app energy consumption. In *ACM MobiCom'12*.

[30] Mobile Enerlytics LCC. eStar: Because mobile devices are not mobile if they are plugged in. http://mobileenerlytics.com.

[31] Monsoon Solutions Inc. Power monitor. http://www.msoon.com/LabEquipment/PowerMonitor.

[32] Nike, Inc. Nike+ Running. http://play.google.com/store/apps/details?id=com.nike.plusgps.

[33] Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. Carat: Collaborative energy diagnosis for mobile devices. In *ACM SenSys'13*.

[34] Paek, J., Kim, J., and Govindan, R. Energy-efficient rate-adaptive GPS-based positioning for smartphones. In *ACM MobiSys'10*.

[35] Pathak, A., Hu, Y. C., and Zhang, M. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *ACM HotNets'11*.

[36] Pathak, A., Hu, Y. C. H., and Zhang, M. Fine grained energy accounting on smartphones with eprof. In *EuroSys'12*.

[37] Pathak, A., Jindal, A., Hu, Y. C., and Midkiff, S. P. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *ACM MobiSys'12*.

[38] Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. Profiling resource usage for mobile applications: a cross-layer approach. In *ACM MobiSys'11*.

[39] Qualcomm. Trepn Profiler. http://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler.

[40] Reddit. NlpWakeLock and NlpCollectorWakeLock discussion. https://www.reddit.com/r/Android/comments/1rvmlr/nlpwakelock_and_nlpcollectorwakelock_discussion/.

[41] rovo89. Xposed development tutorial. https://github.com/rovo89/XposedBridge/wiki/Development-tutorial.

[42] rovo89. Xposed module repository. http://repo.xposed.info.

[43] Roy, A., Rumble, S. M., Stutsman, R., Levis, P., Mazières, D., and Zeldovich, N. Energy management in mobile devices with the Cinder operating system. In *EuroSys'11*.

[44] Siddha, S., Pallipadi, V., and Van De Ven, A. Getting maximum mileage out of tickless. In *Ottawa Linux Symposium'07*.

[45] UzumApps. Wakelock Detector. https://play.google.com/store/apps/details?id=com.uzumapps.wakelockdetector.

[46] Vekris, P., Jhala, R., Lerner, S., and Agarwal, Y. Towards verifying android apps for the absence of no-sleep energy bugs. In *HotPower'12*.

[47] Xu, F., Liu, Y., Li, Q., and Zhang, Y. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *USENIX NSDI'13*.

[48] Xu, F., Liu, Y., Moscibroda, T., Chandra, R., Jin, L., Zhang, Y., and Li, Q. Optimizing background email sync on smartphones. In *ACM MobiSys'13*.

[49] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. ECOSystem: Managing energy as a first class operating system resource. In *ASPLOS'02*.

[50] Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., and Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS'10*.

U-root: A Go-based, firmware embeddable root file system with on-demand compilation

Ronald G. Minnich
Google

Andrey Mirtchovski
Cisco

Abstract

U-root is an embeddable root file system intended to be placed in a FLASH device as part of the firmware image, along with a Linux kernel. The program source code is installed in the root file system contained in the firmware FLASH part and compiled on demand. All the u-root utilities, roughly corresponding to standard Unix utilities, are written in Go, a modern, type-safe language with garbage collection and language-level support for concurrency and inter-process communication.

Unlike most embedded root file systems, which consist largely of binaries, U-root has only five: an init program and 4 Go compiler binaries. When a program is first run, it and any not-yet-built packages it uses are compiled to a RAM-based file system. The first invocation of a program takes a fraction of a second, as it is compiled. Packages are only compiled once, so the slowest build is always the first one, on boot, which takes about 3 seconds. Subsequent invocations are very fast, usually a millisecond or so.

U-root blurs the line between script-based distros such as Perl Linux[24] and binary-based distros such as BusyBox[26]; it has the flexibility of Perl Linux and the performance of BusyBox. Scripts and builtins are written in Go, not a shell scripting language. U-root is a new way to package and distribute file systems for embedded systems, and the use of Go promises a dramatic improvement in their security.

Introduction

Embedding kernels and root file systems in BIOS FLASH is a common technique for gaining boot time performance and platform customization[25][14][23]. Almost all new firmware includes a multiprocess operating system with a full complement of file systems, network drivers, and protocol stacks, contained in an embedded file system. In some cases, the kernel is only booted long enough to boot another kernel; in others, the kernel that is booted and the file system it contains constitute the operational environment of the device[15].

These so-called "embedded root file systems" also contain a set of standard Unix-style programs used for both normal operation and maintenance. Space on the device is at a premium, so these programs are usually written in C using, e.g., the BusyBox toolkit[26]; or in an interpretive languages, such as Perl[24] or Forth. BusyBox in particular has found wide usage in embedded appliance environments, as the entire root file system can be contained in under one MiB.

Embedded systems, which were once standalone, are now almost always network connected. Network-connected systems face a far more challenging security environment than even a few years ago. In response to the many successful attacks against shell interpreters[11] and C programs[8], we have started to look at using a more secure, modern language in embedded root file systems, namely, Go[21][16].

Go is a new systems programming language created by Google. Go has strong typing; language level support for concurrency; inter-process communication via channels, a la Occam[13], Limbo[17], and Alef[27]; runtime type safety and other protective measures; dynamic allocation and garbage collection; closures; and a package syntax, similar to Java, that makes it easy to determine what packages a given program needs.

The modern language constructs make Go a much safer language than C. This safety is critical for network-attached embedded systems, which usually have network utilities written in C, including web servers, network servers including sshd, and programs that provide access to a command interpreter, itself written in C. All are proving to be vulnerable to the attack-rich environment that the Internet has become. Buffer overflow attacks affecting C-based firmware code (among other things) in 2015 include GHOST and the so-called *FSVariable.c* bug in Intel's UEFI firmware. Buffer overflows in Intel's

UEFI and Active Management Technology (AMT)have also been discovered in several versions in recent years. Both UEFI[12] and AMT[4] are embedded operating systems, loaded from FLASH, that run network-facing software; attacks against UEFI have been extensively studied[9]. Most printers are network-attached and are a very popular exploitation target[6].

Firmware is not visible to most users and is updated much less frequently (if at all) than programs. It is the first software to run, at power on reset. Exploits in firmware are extremely difficult to detect, because firmware is designed to be as invisible as possible. Firmware is extremely complex; UEFI is roughly equivalent in size and capability to a Unix kernel. Firmware is usually closed and proprietary, with nowhere near the level of testing of kernels. These properties make firmware an ideal place for so-called advanced persistent threats[10][18][5]. Once an exploit is installed, it is almost impossible to remove, since the exploit can inhibit its removal by corrupting the firmware update process. The only sure way to mitigate a firmware exploit is to destroy the hardware.

Even the most skilled programmers make simple mistakes that in C can be fatal, especially on network-connected systems; nowadays, even the lowest-level firmware in our PCs, printers, and thermostats is network-connected. These mistakes are either impossible to make in Go or, if made, are detected at runtime and result in the program exiting. Perhaps surprisingly, the case for using a high-level, safe language like Go in very low level embedded firmware might be stronger than for user programs, because exploits at the firmware level are nearly impossible to detect and mitigate.

The challenge to using Go in a storage-constrained environment such as firmware is that advanced language features lead to big binaries. Even a date program is about 2 MiB. One Go binary, implementing one function, is twice as large as a BusyBox binary implementing many functions. As of this writing, a typical BIOS FLASH part is 16 MiB. Fitting many Go binaries into a single BIOS flash part is not practical.

The Go compiler is very fast and its sheer speed points to a solution: to compile programs only when they are used. We can build a root file system which has almost no binaries except the Go compiler itself. The compiled programs and packages can be saved to a RAM-based file system.

U-root is our proof of concept of this idea. U-root contains only 5 binaries, 4 of them from the Go toolchain, and the 5th an init binary. The rest of the programs are contained in BIOS FLASH in source form, including packages. The search path is arranged so that when a command is invoked, if it is not in /bin, an installer is invoked instead which compiles the program into /bin; if the build succeeds, the command is executed. This first invocation takes a fraction of a second, depending on program complexity; after that, the RAM-based, statically linked binaries run in about a millisecond.

U-root blurs the boundary between script-based root file systems such as Perl Linux[24] and binary-based root file systems such as BusyBox[26]; it has the flexibility of Perl Linux and the performance of BusyBox. Scripts are written in Go, not a shell scripting language, with two benefits: the shell can be simple, with fewer corner cases; and the scripting environment is substantially improved, since Go is more powerful than most shell scripting languages, but also less fragile and less prone to parsing bugs.

The U-root design

The u-root boot image is a build toolchain and a set of programs in source form. When first used, a program and any needed but not-yet-built packages are built and installed, typically in a fraction of a second. On second and later uses, the binary is executed. The root file system is almost entirely unformed on boot; /init sets up the key directories and mounts, including common ones such as /etc and /proc.

Since the init program itself is only 132 lines of code and is easy to change, the structure is very flexible and allows for many use cases.

- Additional binaries: if the 3 seconds it takes to get to a shell is too long (some applications such as automotive computing require 800 ms startup time), and there is room in FLASH, some programs can be precompiled into /bin.

- Build it all on boot: if on-demand compilation is not desired, a background thread in the init process can build all the programs on boot.

- Selectively remove binaries after use: if RAM space is at a premium, once booted, a script can remove everything in /bin; those things that are used will be rebuilt on demand.

- Always build on demand: it is possible to run in a mode in which programs are never written to /bin and always rebuilt on demand; this mode is surprisingly comfortable to use, given that program compilation is so fast[1].

- Lockdown: if desired, the system can be locked down once booted in one of several ways: the entire /src tree can be removed, for example, or just the compiler toolchain can be deleted.

How u-root works

U-root is packaged as an LZMA-compressed initial RAM file system (initramfs) in cpio format, contained in a Linux compressed kernel image, a.k.a. bzImage. The bootloader (e.g. syslinux) or firmware (e.g. coreboot) loads the bzImage into memory and starts it. The Linux kernel sets up a RAM-based root file system and unpacks the u-root file system into it. This initial root file system contains a Go toolchain (4 binaries), an init binary, the u-root program source, and the entire Go source tree, which provides packages needed for u-root programs.

All Unix systems start an init process on boot and u-root is no exception. The init for u-root sets up some basic directories, symlinks, and files; builds a command installer; and invokes the shell. We describe this process in more detail below. The boot file system layout is shown in Table 1.

The src directory is where programs and u-root packages live. The go/bin directory is for any Go tools built after boot; the go/pkg/tool directory contains binaries for various architecture/kernel combinations. The directory in which a compiler toolchain is placed provides information about the target OS and architecture; for example, the Go build places binaries for Linux on x86_64 in in /go/pkg/tool/linux_amd64/. Note that there is no /bin or many of the other directories expected in a root file system. The init binary builds them. The u-root root file system has very little state.

For most programs to work, the file system must be more complete. We save space in the image by having init create additional file system structure at boot time: it fills in the missing parts of the root filesystem. It creates /dev and /proc and mounts them. It creates an empty /bin which is filled with binaries on demand. We show it in Table 2.

Note that in addition to /bin, there is a directory called /buildbin. Buildbin and the correct setup of $PATH are the keys to making on-demand compilation work. The init process sets $PATH to `/go/bin:/bin:/buildbin:/usr/local/bin`. Init also builds the `installcommand`, using the go bootstrap builder; and creates a complete set of symlinks as shown. As a final step, init execs sh.

There is no /bin/sh at this point; the first sh found in $PATH is /buildbin/sh. This is a symlink to installcommand. Installcommand, once started, examines argv[0], which is sh, and takes this as instruction to build /src/cmds/sh/*.go into /bin and then exec /bin/sh. There is no difference between starting the first shell and any other program.

Hence, part of the boot process involves the construction of an installation tool to build a binary for a shell which is then run. If a user wants to examine the source

Table 1: The initial layout of a u-root file system. All Go compiler and runtime source is included under /go/src; all u-root source under /src; and the compiler toolchain binaries under /go/pkg.

/src	cmds/	
		builtin/builtin.go
		cat/cat.go
		cmp/cmp.go
		comm/comm.go
		cp/cp.go
		date/date.go
		dmesg/dmesg.go
		echo/echo.go
		freq/freq.go
		grep/grep.go
		init/init.go
		installcommand/installcommand.go
		ip/ip.go
		ldd/ldd.go
		losetup/losetup.go
		ls/ls.go
		mkdir/mkdir.go
		mount/mount.go
		netcat/netcat.go
		ping/ping.go
		printenv/printenv.go
		rm/rm.go
		script/script.go
		seq/seq.go
		sh/{cd.go,parse.go,sh.go,time.go}
		srvfiles/srvfiles.go
		tcz/tcz.go
		tee/tee.go
		uniq/uniq.go
		wc/wc.go
		wget/wget.go
		which/which.go
	pkg/	dhcp/ (dhcp package source)
		netlib/ (netlib package source)
		golang.org (import package source)
/go	src/	Packages and toolchain
	pkg/	tool/linux_amd64/{6a,6c,6g,6l}
	misc/	...
	tool/	...
	bin/	go
	include/	...
/lib/	libc.so	Needed for tinycore linux packages
	libm.so	

Table 2: Layout after /init has run. /buildbin contains symlinks to enable the on-demand compilation, and other standard directories and mount points are ready.

/	Root file system from Table 1
/buildbin/ (built/installed by /init) (/init creates symlinks)	(created by /init) installcommand builtin->installcommand cat->installcommand cmp->installcommand comm->installcommand cp->installcommand date->installcommand dhcp->installcommand dmesg->installcommand echo->installcommand freq->installcommand grep->installcommand init->installcommand ip->installcommand ldd->installcommand losetup->installcommand ls->installcommand mkdir->installcommand mount->installcommand netcat->installcommand ping->installcommand printenv->installcommand rm->installcommand script->installcommand seq->installcommand sh->installcommand srvfiles->installcommand tcz->installcommand tee->installcommand uniq->installcommand wc->installcommand wget->installcommand which->installcommand
/bin	/init creates
/proc	/init mounts /proc
/tcz	/init creates for tinycore binaries
/dev	init creates minimal needed devices
/etc	init writes resolv.conf

to the shell, they can cat /src/cmds/sh/*.go; the cat command will be built and then show those files.

U-root is intended for network-based devices, and hence good network initialization code is essential. U-root includes a Go version of the `ip` and `dhcp` programs, along with the docker `netlink` package and a dhcp package. Support for WIFI configuration is underway.

The u-root shell

A shell is a key part of any boot system. Shells run commands, where a command is a sequence of one or more programs, potentially tied together with pipes or other operators. Shells may run scripts from a file. Scripts are usually simple sequences of commands, each command invoking just one program, but the shell language may allow more complex commands in a script. Shells have built-in commands, i.e. commands that do not invoke a program, but are recognized by the shell and executed directly. Builtins are used when the command must change the shell state, as in the `cd` command; when the cost of starting a program is felt to be too high relative to the operation the command performs; because the shell source is not available or it is too hard to change the shell; or for convenience, i.e. users would rather write in the shell language instead of C. Most shells can be extended via a builtin facility, which usually looks like a function definition style syntax. The shell scripting language is usually the same language used for builtins.

Every boot loader in common use today has some sort of shell capability. That these shells have many limitations is a given, but at the same time they need to look as much as possible like a standard shell.

U-root provides a shell that is stripped down to the fundamentals: it can read commands in, using the Go scanner package; it can expand (i.e. glob) the command elements, using the Go filepath package; and it can run the resulting commands, either programs or shell builtins. It supports pipelines and IO redirection. At the same time, the shell defines no language of its own for scripting and builtins; instead, the u-root shell uses the Go compiler. In that sense, the u-root shell reflects a break in important ways with the last few decades of shell development, which has seen shells and their language grow ever more complex and, partially as a result, ever more insecure[19] and fragile[11].

The shell has several builtin commands, and the user can extend it with builtin commands of their own. Before we discuss user-defined builtins, we will describe the basic source structure of u-root shell builtins.

All shell builtins, including the ones that come with the shell by default, are written with a standard Go init pattern which installs one or more builtins. Shown in

Figure 1 and 2 is the shell builtin for time.

Builtins in the shell are defined by a name and a function. One or more builtins can be described in a source file. The name is kept in a map and the map is searched for a command name before looking in the file system. The function must accept a string as a name and a (possibly zero-length) array of string arguments, and return an error. In order to connect the builtin to the map, the programmer must provide an init function which adds the name and function to the map. The init function is special in that it is run by Go when the program starts up. In this case, the init function just installs a builtin for the time command.

Scripting and builtins

To support scripting and builtins, u-root provides two programs: `script` and `builtin`. The `script` program allows users to specify a Go fragment on the command line, and runs that fragment as a program. The `builtin` program allows a Go fragment to be built into the shell as a new command. Builtins are persistent; the builtin command instantiates a new shell with the new command built in. Scripts run via the script command are ephemeral.

We show a usage of the script command in Figure 3.

This script implements printenv. Note that it is not a complete Go program in that it lacks a package statement, imports, a main function declaration, and a return at the end. All the boilerplate is added by the script command, which uses the Go imports package to scan the code and create the import statements required for compilation (in this case, both fmt and os packages are imported). Because our shell is so simple, there is no need to escape many of these special characters. We have offloaded the complex parsing tasks to Go.

Builtins are implemented in almost the same way. The builtin command takes the Go fragment and creates a standard shell builtin Go source file which conforms to the builtin pattern. This structure is easy to generate programmatically, building on the techniques used for the script command.

A basic hello builtin can be defined on the command line:

```
builtin hello \
'{ fmt.Printf("Hello\n") }'
```

The fragment is defined by the {} pair. Given a fragment that starts with a {, the builtin command generates all the wrapper boiler plate needed. The builtin command is slightly different from the script command in that the Go fragment is bundled into one argument. The command accepts multiple pairs of command name and Go code

```
// Package main is the 'root' of the
// package hierarchy for a program.
// This code is part of the main
// program, not another package,
// and is declared as package main.
package main

// A Go source file lists
// all the packages on which
// it has a direct dependency.
import (
  "fmt"
  "os"
  "time"
)

// init() is an optional function.
//If init() is present in a file,
// the Go compiler and runtime
// arrange for it to be called
// at program startup.
// It is hence like a constructor.
func init() {
        // addBuiltIn is provided by
        // the u-root shell for
        // the addition of builtin
        // commands. Builtins must
        // have  a standard type:
        // o The first parameter is
        //    a string
        // o The second is a string
        //    array which may be 0
        //   length
        // o The return is the Go
        //    error type
        // In this case,
        // we are creating a builtin
        // called time which calls
        // the timecmd function.
  addBuiltIn("time", timecmd)
}
```

Figure 1: The code for time builtin, Part I: setup

```
// The timecmd function is passed
// the name of a command to run,
// optional arguments,
// and returns an error. It:
// o gets the start time using Now
//   from the time package
// o runs the command using the
//    u-root shell runit function
// o computes a duration using
//    Since from the time package
// o if there is an error,
//    prints the error to os.Stderr
// o uses fmt.Printf to print
//    the duration to os.Stderr
// Note that since runtime always
// handles the error, by printing
// it, it always returns nil.
// Most builtins return the error.
// Here you can see the usage
// of the imported packages
// from the imports statement above.
func timecmd(name string, args []
    string) error {
  start := time.Now()
  err := runit(name, args)
        if err != nil {
    fmt.Fprintf(os.Stderr, "%v\n",
        err)
  }
  cost := time.Since(start)
  fmt.Printf(os.Stderr, "%v", cost)
        // This function is special
            in that
        // it handles the error, and
            hence
        // does not return an error.
        // Most other builtins return
            the
        // error.
  return nil
}
```

Figure 2: The code for the shell time builtin, Part II.

```
script \
{ fmt.Printf("%v\n", os.Environ()) }
```

Figure 3: Go fragment for a printenv script. Code structure is inserted and packages are determined automatically.

fragments, allowing multiple new builtin commands to be installed in the shell.

Builtin creates a new shell at `/bin/sh` with the source at `/src/cmds/sh/`. Invocations of `/bin/sh` by this shell and its children will use the new shell. Processes spawned by this new shell can access the new shell source and can run the builtin command again and create a shell that further extends the new shell. Processes outside the new shell's process hierarchy can not use this new shell or the builtin source. When the new shell exits, the builtins are no longer visible in any part of the file system. We use Linux mount name spaces to create this effect[22]. Once the builtin command has verified that the Go fragment is valid, it builds a new, private namespace with the shell source, including the new builtin source. From that point on, the new shell and its children will only use the new shell. The parent process and other processes outside the private namespace continue to use the old shell.

Environment variables

The u-root shell supports environment variables, but manages them differently than most Unix environments. The variables are maintained in a directory called /env; the file name corresponds to the environment variable name, and the files contents are the value. When it is starting a new process, the shell populates child process environment variables from the /env directory. The syntax is the same; $ followed by a name directs the shell to substitute the value of the variable in the argument by prepending /env to the path and reading the file.

The shell variables described above are relative paths; /env is prepended to them. In the u-root shell, the name can also be an absolute path. For example, the command `script $/home/rminnich/scripts/hello` will substitute the value of the hello script into the command line and then run the script command. The ability to place arbitrary text from a file into an argument is proving to be extremely convenient, especially for script and builtin commands.

Using external packages and programs

No root file system can provide all the packages all users want, and u-root is no exception. We must have the ability to load external packages from popular Linux distros. As a proof of concept, we created a tool to load external packages from the TinyCore Linux distribution, a.k.a. tinycore. A tinycore package is a mountable file system image, containing all the package files, including a file listing any additional package dependencies.

To load these packages, u-root provides the tcz command which fetches the package and and needed depen-

dencies. Hence, if a user wants emacs, they need merely type tcz emacs, and emacs will become available in /usr/local/bin. The tinycore packages directory can be a persistent directory or it can be empty on each boot.

The tcz command is quite flexible as to what packages it loads and where they are loaded from. Users may specify the host name which provides the packages; the TCP port on which to connect; the version of tinycore to use; and the architecture. The tcz command must loopback mount each package as it is fetched, and hence must cache them locally. It will not refetch already cached packages. This cache can be volatile or maintained on more permanent storage. Performance varies depending on the network being used and the number of packages being loaded, but seems to average about 1 second per package on a WIFI-attached laptop.

U-root also provides a small web server, called srvfiles, that can be used to serve locally cached tinycore packages for testing. The entire server is 18 lines of Go.

Using u-root: current targets

There are three current targets for u-root. All three are available in a Docker image we provide.

The first two targets are used to test u-root to make sure it will work before it is loaded into its real target, a firmware image.

Chroot test

The first test target is a chroot environment. A chroot is a file system tree which must have at least one binary. A standard Unix command, chroot, uses the chroot system call to set the root for a child process and then execs a named binary from the tree. Note that the chroot only applies to the program being run, and does not affect any other programs.

A script provided with u-root builds an image of the file system shown in Table 1, including locating and installing the Go source tree and toolchain. The script also builds the init binary which the kernel runs as the first user-mode process.

The chroot environment simulates a full boot environment. The chroot startup process, running on a linux instance in VmWare Fusion on a Mac laptop takes about 3 seconds, including compiling the two binaries (installcommand and sh) and the packages they need, about 250 files. Once the startup process is done the user sees the u-root shell prompt and can run tests.

Kernel image

Once the file system tree has been verified via the chroot, it can be used as the input to the process of creating a so-called initramfs. An initramfs is a file system image that is built directly into a bootable Linux kernel image. When the kernel starts, it locates the initramfs, sets up a RAM file system, and extracts the intramfs into the RAM file system. At that point, the kernel can exec /init.

U-root includes the script to create this image. The result is a file, which can then be used as input to the kernel build process. The user can then boot the kernel directly in QEMU (via qemu –kernel) or drop the bzImage into a boot disk image and test that, either via qemu or booting on real hardware.

Firmware image

The end goal of u-root is to create an embedded firmware image. Linux can not run from power on reset directly; something needs to configure the platform and then load Linux from FLASH to RAM, and for that we use coreboot. We build a kernel as shown earlier, containing an initramfs, which in turn contains u-root; this in turn is built into a firmware image.

Users can take two steps to test coreboot. The first (and optional given enough confidence) is to add the kernel bzImage as a payload to a coreboot built to run in qemu. We provide a working qemu image in a Docker container to make this easy, as well as a script to add the bzImage to the coreboot image. Once the image is built, users start qemu with this image as the bios: qemu –bios coreboot.rom

Once the Linux image is known good, the user can embed it in a real coreboot image for a real mainboard. A current known limitation is that the board must contain a 16 MiB FLASH part. We have tested on the Asrock E350M1. In that case, we first tested on QEMU, then took the Linux image unchanged, merged it into the coreboot for the hardware, and it worked with no changes, indicating that QEMU provides a very accurate verification environment for the hardware target. If a given u-root build works in QEMU it will almost certainly work on hardware.

Discussion

Building the image

Building is a straightforward process, which requires a kernel source tree, Go source tree, the u-root source, and coreboot source. Build times vary depending on what infrastructure is used. In general, the kernel and Go build steps are measured in minutes, and the u-root and coreboot build times are measured in seconds. These orders of magnitude have changed little in the last 5 years.

Usability

We have been using u-root for a few months. The system provides a very usable firmware command line environment, comparable to what we have used with U-boot, UEFI, and Open Firmware. The u-root and its shell provide familiar tools and capabilities. The ability to start background tasks, shell pipelines, and redirect output to files is both useful and unusual in an embedded environment. In terms of scripting, it is more sophisticated than any shell we know of, given that our scripting language is Go.

The script command is more powerful than we first realized. It is possible to run it under any shell, or from any program. It can use much more powerful Go fragments than we have shown here, including whole Go programs, since its internal parsing is just the standard Go "imports" library. The script command fills in the blanks as needed, but only as needed.

The builtin command is perhaps the most powerful tool in u-root. It allows users to build fully customized shells for a specific purpose and then just as quickly discard them. The new shell is ready and running in less than a second. We could apply this technique to the problem of startup scripts for embedded environments. To support standard functions in init scripts, distributions provide several hundred files, and for each invocation of the shell, some set of these files are included over and over again. The time it takes to read, parse and run these standard scripts is a large part startup time[1].

With the scripting and builtin tools, users do not need to write full Go programs to get a new capability. We are finding that the basic set of tools we have is enough, and we are writing new tools as Go fragments.

Once the network is up, tools like emacs can be loaded either via a local disk, local network package server, or a remote server such as tinycorelinux.org.

Having the source always available in any sort of firmware environment is both unusual and very useful. At times, we have forgotten how some of our commands work. Having the source at hand has proven very helpful.

Future development

While we envisioned u-root as a boot time environment, we have seen increasing interest in wider use. Some users have requested common features as tab completion. The shell parser is written in such a way that it should allow for easy addition of tab completion.

History is another question, as it adds more state, complexity, and parsing to the shell. These additions in turn decrease reliability and open up paths for exploits. We are experimenting with new models of history maintenance that do not require the complexity of current systems. We might, for example, maintain history in a private per-process or per-user directory. Finding commands becomes easy, and with our extension to the shell variable model, running an old command becomes easy: $/home/rminnich/history/5 would run the fifth command. Another attraction of this model is that conversion of an old command into a shell script is easy. History stops being special to one shell and instead is common to all programs that can traverse files. Any program can see history and use the commands.

Instead of rereading the same scripts over and over, init could build a special shell at boot time that pulls in builtins to extend the shell. The scripts themselves would continue to be human-readable, but the performance of booting would be much faster, combining the perceived advantages of upstart-style scripting with systemd-level performance.

Related work

There are two main components to this work: on-demand compilaton and embedding a kernel and root file system in FLASH. Both ideas have been used at different times. Our work combines the two so that we can use Go. We review the earlier work below.

On-Demand Compilation

On-Demand compilation is one of the oldest ideas in computer science.

Slimline Open Firmware (SLOF)[7] is a FORTH-based implementation of Open Firmware developed by IBM for some of its Power and Cell processors. SLOF is capable of storing all of Open Firmware as source in the FLASH memory and compiling components to indirect threading on demand[2].

In the last few decades, as our compiler infrastructure has gotten slower and more complex, true on-demand compilation has split into two different forms. First is the on-demand compilation of source into executable byte codes, as in Python. The byte codes are not native but are more efficient than source. If the python interpreter finds the byte code it will interpret that instead of source to provide improved performance.

Java takes the process one step further with the Just In Time compilation of byte code to machine code[20] to boost performance.

[1] A good discussion can be found in http://free-electrons.com/doc/training/boot-time/boot-time-slides.pdf

Embedding kernel and root file systems in FLASH

The LinuxBIOS project[14][1], together with clustermatic[25], used an embedded kernel and simple root file system to manage supercomputing clusters. Due to space constraints of 1 MiB or less of FLASH, clusters embedded only a single-processor Linux kernel with a daemon. The daemon was a network bootloader that downloaded a more complex SMP kernel and root file system and started them. Clusters built this way were able to boot 1024 nodes in the time it took the standard PXE network boot firmware to find a working network interface.

Early versions of One Laptop Per Child used LinuxBIOS, with Linux in flash as a boot loader, to boot the eventual target. This system was very handy, as they were able to embed a full WIFI stack in flash with Linux, and could boot test OLPC images over WIFI. The continuing growth of the Linux kernel, coupled with the small FLASH size on OLPC, eventually led OLPC to move to Open Firmware.

AlphaPower shipped their Alpha nodes with a so-called Direct Boot Linux, or DBLX. This work was never published, but the code was partially released on sourceforge.net just as AlphaPower went out of business. Compaq also worked with a Linux-As-Bootloader for the iPaq.

Car computers and other embedded ARM systems frequently contain a kernel and an ext2 formatted file system in NOR FLASH, i.e. FLASH that can be treated as memory instead of a block device. Many of these kernels use the so-called eXecute In Place[3] (XIP) patch, which allows the kernel to page binaries directly from the memory-addressable FLASH rather than copying it to RAM, providing a significant savings in system startup time. A downside of this approach is that the executables can not be compressed, which puts further pressure on the need to optimize binary size. NOR FLASH is very slow, and paging from it comes at a significant performance cost. Finally, an uncompressed binary image stored in NOR FLASH has a much higher monetary cost than the same image stored in RAM since the cost per bit is so much higher.

UEFI[12] contains a non-Linux kernel (the UEFI firmware binary) and a full set of drivers, file systems, network protocol stacks, and command binaries in the firmware image. It is a full operating system environment realized as firmware.

The ONIE project[23] is a more recent realization of the Kernel-in-FLASH idea, based on Linux. ONIE packs a Linux kernel and Busybox binaries into a very small package. Since the Linux build process allows an initial RAM file system (initramfs) to be built directly into the kernel binary, some companies are now embedding ONIE images into FLASH with coreboot. Sage Engineering has shown a bzImage with a small Busybox packed into a 4M image. ONIE has brought new life to an old idea: packaging a kernel and small set of binaries in FLASH to create a fast, capable boot system.

Conclusions and future work

U-root is a root file system targeted to embedded firmware environments. In response to the increasing security challenges facing embedded systems in the always-connected Internet of Things, we have chosen to write all the u-root programs in Go, a modern, type safe language with garbage collection. The safety of the Go language and runtime reduce many of the security risks of writing network-facing services. The performance of the Go compiler makes on-demand compilation practical: most commands compile in a fraction of a second and, once compiled, run in about a millisecond.

The U-root file system, on boot, contains only 5 binaries. The rest of the root file system contains source to programs which are compiled on demand. We have found the system to be fast and usable. The images can be tested in emulation environments, of increasing fidelity to the firmware target, and have been tested on hardware running coreboot.

Our initial intent was to use u-root to build firmware images, but we are finding that we would like to use it more broadly. The structure of the Go toolchain naming scheme lends itself to heterogeneous environments: save for init, the toolchain binaries have a directory path name that includes the name of the target OS and architecture, e.g. linux_amd64, linux_arm, and so on. A single u-root image can contain many Go toolchains with no path conflicts. Were we to install more toolchains in the root file system, and move /init to the same directory containing the toolchain, a single u-root file system image could be used on many different OS and architecture combinations. We could build a u-root image, for example, that worked on all linux variants for different architectures. We are exploring this model now.

Availability

U-root is available as a git repo from

```
github.com/rminnich/u-root
```

To make trying it out easier, we have created a docker container,

```
docker.io/rminnich:18
```

The container includes all the u-root, linux kernel, and coreboot source needed to test three environments: the chroot, kernel and initramfs, and coreboot with qemu. There are scripts and logs of sessions in / which users can use to guide and verify their testing of the software.

Because u-root changes frequently, users should pull an update in /u-root once they have done initial testing.

Acknowledgements

Thanks to Maya Gokhale for her many helpful suggestions on this paper.

References

[1] AGNEW, A., SULMICKI, A., MINNICH, R., AND ARBAUGH, W. A. Flexibility in rom: A stackable open source bios. In *USENIX Annual Technical Conference, FREENIX Track* (2003), pp. 115–124.

[2] (AUTHOR OF SLOF), S. B. Personal conversation.

[3] BENAVIDES, T., TREON, J., HULBERT, J., AND CHANG, W. The enabling of an execute-in-place architecture to reduce the embedded system memory footprint and boot time. *Journal of computers 3*, 1 (2008), 79–89.

[4] BOGOWITZ, B., AND SWINFORD, T. Intel® active management technology reduces it costs with improved pc manageability. *Technology@ Intel Magazine* (2004).

[5] CELEDA, P., KREJCI, R., VYKOPAL, J., AND DRASAR, M. Embedded malware-an analysis of the chuck norris botnet. In *Computer Network Defense (EC2ND), 2010 European Conference on* (2010), IEEE, pp. 3–10.

[6] CUI, A., COSTELLO, M., AND STOLFO, S. J. When firmware modifications attack: A case study of embedded exploitation. In *NDSS* (2013).

[7] DALY, D., CHOI, J. H., MOREIRA, J. E., AND WATERLAND, A. Base operating system provisioning and bringup for a commercial supercomputer. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), IEEE, pp. 1–7.

[8] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., ET AL. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 475–488.

[9] KALLENBERG, C., AND BULYGIN, Y. All your boot are belong to us intel, mitre. cansecwest 2014.

[10] KALLENBERG, C., KOVAH, X., BUTTERWORTH, J., AND CORNWELL, S. Extreme privilege escalation on windows 8/uefi systems.

[11] KOZIOL, J., LITCHFIELD, D., AITEL, D., ANLEY, C., EREN, S., MEHTA, N., AND HASSELL, R. *The Shellcoder's Handbook*. Wiley Indianapolis, 2004.

[12] LEWIS, T. Uefi overview, 2007.

[13] MAY, D. Occam. *ACM Sigplan Notices 18*, 4 (1983), 69–79.

[14] MINNICH, R. G. Linuxbios at four. *Linux J. 2004*, 118 (Feb. 2004), 8–.

[15] MOON, S.-P., KIM, J.-W., BAE, K.-H., LEE, J.-C., AND SEO, D.-W. Embedded linux implementation on a commercial digital tv system. *Consumer Electronics, IEEE Transactions on 49*, 4 (Nov 2003), 1402–1407.

[16] PIKE, R. Another go at language design. Stanford University Computer Systems Laboratory Colloquium.

[17] RITCHIE, D. M. The limbo programming language. *Inferno Programmer's Manual 2* (1997).

[18] SACCO, A. L., AND ORTEGA, A. A. Persistent bios infection. In *CanSecWest Applied Security Conference* (2009).

[19] SAMPATHKUMAR, R. *Vulnerability Management for Cloud Computing-2014: A Cloud Computing Security Essential*. Rajakumar Sampathkumar, 2014.

[20] SUGANUMA, T., OGASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. Overview of the ibm java just-in-time compiler. *IBM systems Journal 39*, 1 (2000), 175–193.

[21] TEAM, G. The go programming language specification. Tech. rep., Technical Report http://golang. org/doc/doc/go spec. html, Google Inc, 2009.

[22] VAN HENSBERGEN, E., AND MINNICH, R. Grave robbers from outer space: Using 9p2000 under linux. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 83–94.

[23] VARIOUS. No papers have been published on onie; see onie.org.

[24] VARIOUS. No papers were published; see perllinux.sourceforge.net.

[25] WATSON, G. R., SOTTILE, M. J., MINNICH, R. G., CHOI, S.-E., AND HERTDRIKS, E. Pink: A 1024-node single-system image linux cluster. In *High Performance Computing and Grid in Asia Pacific Region, 2004. Proceedings. Seventh International Conference on* (2004), IEEE, pp. 454–461.

[26] WELLS, N. Busybox: A swiss army knife for linux. *Linux J. 2000*, 78es (Oct. 2000).

[27] WINTERBOTTOM, P. Alef language reference manual. *Plan 9 Programmer's Man* (1995).

Notes

[1] In fact, at one point, due to a configuration error, we were using this mode all the time without realizing it. Go compiles are that fast.

LPD: Low Power Display Mechanism for Mobile and Wearable Devices

MyungJoo Ham
Frontier Computer Science Lab,
Software R&D Center,
Samsung Electronics

Inki Dae Chanwoo Choi
Software Platform Team,
Software R&D Center,
Samsung Electronics

{myungjoo.ham, inki.dae, cw00.choi}@samsung.com

Abstract

A plethora of mobile devices such as smartphones, wearables, and tablets have been explosively penetrated into the market in the last decade. In battery powered mobile devices, energy is a scarce resource that should be carefully managed. A mobile device consists of many components and each of them contributes to the overall power consumption. This paper focuses on the energy conservation problem in display components, the importance of which is growing as contemporary mobile devices are equipped with higher display resolutions. Prior approaches to save energy in display units either critically deteriorate user perception or depend on additional hardware. We propose a novel display energy conservation scheme called *LPD* (Low Power Display) that preserves display quality without requiring specialized hardware. *LPD* utilizes the display update information available at the X Window system and eliminates expensive memory copies of unvaried parts. *LPD* can be directly applicable to devices based on Linux and X Windows system. Numerous experimental analyses show that *LPD* saves up to 7.87% of the total device power consumption. Several commercial products such as Samsung Gear S employ *LPD* whose source code is disclosed to the public as open-source software at `http://opensource.samsung.com` and `http://review.tizen.org`.

1 Introduction

The popularity of mobile devices such as smartphones, tablets, and smart watches is steadily increasing and their market size has grown explosively in recent years. Tetherless mobile devices use batteries as the main energy source and power is one of the scarcest resources that should be carefully managed; energy consumption is directly translated to the usability and the value of mobile products. In addition, imprudent use of energy may lead to excessive heat dissipation, which in turn, causes a safety issue of low temperature burns [13]. One easy solution for the power saving problem is to equip better and/or larger batteries in mobile devices. However, the advancements in battery technology failed to match the ever increasing functionalities and computational demands of mobile devices [17].

A mobile device consists of many components and functions each of which consumes energy. This paper deals with energy conservation in display components. As the display resolutions increase, the energy required to operate a device grows accordingly. For example, even though the physical scales of displays have not grown much bigger, resolution has increased from 800x480 to 2560x1440. The memory bandwidth increases almost ten times and so is the energy consumed. Energy conservation in memory access for display components has received less research attentions than other components such as processors and communication interfaces although display components consume significant share of energy [5].

Energy conservation schemes prone to deteriorate the performance or QoE (Quality of Experience) of devices. Because human beings are sensitive to the degradation in visual quality, vigilant attentions to preserve the original visual quality must be accompanied in designing power saving techniques for display units. Adjustments of color depth [8], brightness level [6, 11], or refresh rate [14] may significantly affect user perception such that the quality assurance team often rejects careless schemes. Of course, there are display energy saving schemes that preserve the original quality. AFBC (ARM Frame Buffer Compression) [3, 10], Transaction Elimination [4, 16], and frame buffer compression [20] are examples of such approaches. However, most of these schemes depend on specialized hardware and their applicability is quite limited.

We aim to develop a display energy conservation scheme that neither requires the addition of specialized hardware nor deteriorate the visual quality. The proposed scheme, low power display (*LPD*), does not require any hardware modifications to the traditional and popular `i80` display architecture, Intel's 8080 like command interface for display panels. *LPD* also does not deteriorate the user experiences because it conserves the true quality of every pixel.

The main idea of *LPD* is rather simple; to reduce memory accesses and data transfers by identifying the updated regions. The idea of preserving unchanged part and encoding only changed part is widely used in motion picture encoding [22] and display rendering. The problem is how to identify the updated regions. Comparing the two consecutive frame buffers directly requires too much energy or additional hardware. Instead of direct frame buffer comparison, we exploit the knowledge that the OS already possesses. In other words, *LPD* extends the design domain from HW-kernel to HW-kernel-middleware. In Linux and Tizen, a window system (X Server) and a compositing window manager (Enlightenment in the case of Tizen) know the changed regions. LPD accesses changed regions only and transfers the retrieved regions to the display controller and display panel. Therefore, LPD reduces the memory bandwidth as well as bus utilization which in turn reduces power consumption.

LPD also has the potential to enhance the performance of other functions because *LPD* reduces main memory bandwidth and the saved bandwidth can be distributed to other memory hungry functions. Unlike the previous schemes, the computation overhead of *LPD* is minimal; it requires a few simple integer arithmetic instructions without any loops or complex computation. Finally, *LPD* is orthogonal to other display power saving mechanisms [4, 8, 11, 14, 16] such that LPD can be applied with these methods.

To reconstruct a whole display image from updated regions only, the display panel should have an internal RAM that stores the previous frame. Such a feature is commonly available in mobile devices; `i80`, one of the de facto standard display interfaces supports an internal RAM. We confirmed that many mobile devices such as Galaxy S4 and Galaxy Note 3 use the `i80` interface.

We implemented *LPD* and *LPD* has been embedded in commercial products. An earlier version of *LPD* has been shipped with Gear 2. Field tests with real products under real-world use scenarios showed that *LPD* reduce up to 7.87% of the total device power consumption when 1% of frame is updated. Full capability of *LPD* has been implemented and embedded to Gear S. We disclose the source code of full *LPD* implementation to the public at `http://tizen.org` and `http://opensource.samsung.com`. The source code is under the GPL license as a feature of Direct Rendering Manager (DRM), which significantly lessens the maintenance and porting cost for further deployment.

The main contribution of this paper is as follows:

- Improve energy efficiency of display device components that were not properly addressed while
  - preserving the transparency of applications,
  - maintaining traditional hardware architectures,
  - minimizing changes to the operating systems,
  - limiting the overhead to virtually non-existing,
  - not deteriorating the quality of pixels,
  - and allowing most of previous display power optimizing schemes orthogonally coexisting.
- The proposed scheme is fully developed and released as open source software in commercial products.

This paper is organized as follows. The next section presents the related work of display power saving. Section 3 explains the hardware architecture and the rationale of *LPD*. Section 4 shows the design and implementation detail of *LPD*. Section 5 describes the experiments and their results. Section 6 discusses follow-up research that may further enhance *LPD*. Section 7 concludes the paper.

2 Related Work

Several researchers have attacked the power consumption of display-related device components. In this section, we introduce their work and we show why we still need a new mechanism.

Adjust color depth: Choi et al. [8] have suggested a display power saving mechanism that dynamically alters color depth according to the color distribution of a frame buffer. This method scans the whole frame buffer, which usually is performed by an additional hardware to avoid excessive CPU overhead and power consumption. The mechanism is especially effective with high quality high resolution displays while it inevitably deteriorates the picture quality.

Dynamic backlight brightness: backlight is the dominant power consumption source in display systems and several backlight reduction mechanisms have been

devised [1, 6, 7, 11, 19]. Backlight reduction should be accompanied with careful pixel color adjustment to keep the fidelity of images. For example, if a frame is filled with dark pixels, we may reduce the backlight brightness while compensate the gamma values of pixels to brighter colors. Enhancing such approaches further, [21] suggested to partition a screen into multiple regions with separated backlights and adjust the backlights and colors independently for each block for extra power saving.

Dynamic backlight reduction schemes have limitations. Chang et al. [6] sacrificed brightest pixels to reduce the backlight brightness. This optimization degraded the picture quality significantly such that the degradation can be detected by naked eyes. Backlight reduction schemes also require additional full scan of each frame buffer. Full frame scan inevitably provokes additional memory transactions and power consumption. LCD (Liquid Crystal Display) where the responses of each color to brightness are non-linear spawns another complicated control problem [1]. Significant latency increment is another roadblock for the adoption of the technique [6] to latency critical applications such as games, screen scrolling, and typing [19]. Most critically, the brightness control schemes cannot be applied to AMOLED (Active-Matrix Organic Light-Emitting Diode). AMOLED displays, dispense with backlights, are considered to be energy efficient and more suitable for mobile devices [12]. A similar approach for AMOLED displays [18], which tries to adjust pixel colors, may consume much energy due to the physical characteristics of AMOLED; if a pixel changes its color too drastically in a short time, this causes much energy consumption to drive the pixel.

Dynamic display refresh rate: Kim et al. [14] have suggested to dynamically scale the refresh rate of displays. We have applied the technique as a device driver of DVFS framework (devfreq) in the Linux kernel [9], but failed to meet the requirement of picture quality maintained by our quality assurance teams. With further optimizations, the techniques can be effective and applied with *LPD* orthogonally.

Compression: another approach is the frame buffer compression [3, 10, 20]. Compression reduces data size and thus decreases bus traffic and memory operations. Compression is usually performed by an additional non-standard hardware because compressing the whole frame buffer for every frame incurs heavy computational overheads [20]. Compression also incurs power consumption; even with a dedicated FPGA based hardware [20], compressing and decompressing

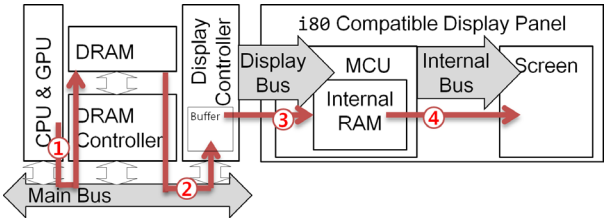


Figure 1: i80 Display Hardware Architecture

frame buffers of 640x480 with 18 bit color depth has consumed additional 30 mW. ARM's Adaptive Scalable Texture Compression (ASTC) [2, 15] provides higher compression rate than other conventional frame buffer compression mechanisms. However, ASTC is limited to textures for GPUs and uses lossy compression mechanisms.

Skip duplicated transmissions: the prior methods that are most similar to *LPD* are the mechanisms that skip transmissions of duplicated parts. Whelan et al. [24] saves the whole frame buffer at the display controller and allows skipping the transfer of a new frame from the main memory to the frame buffer if there are no changes. The benefit of skipping is achievable only when there is not even a pixel of change in a frame [24]. We implemented a variant of this scheme in Samsung Gear series products. In this implementation, we can turn the whole CPU off (suspend-to-RAM) along with the display controller while the screen kept on.

Another similar approach is Transaction Elimination developed by ARM [4, 16]. Transaction Elimination allows a GPU to skip transmitting unchanged parts of its frame buffers to the main memory based on CRC signatures. This approach requires ARM's Midgard GPU architecture. Transaction Elimination reduces data transfer to the main memory only maintaining the data transfers from the main memory to the display panel via the display controller. On the contrary, *LPD* can reduce the data transfers from the main memory to the display panel and does not require using a specific GPU.

3 Background

Figure 1 shows the hardware architecture and *LPD* procedure. Arrows with circled numbers represent image data transmission between hardware components. *LPD* requires a display panel with the i80 display interface and a display controller supporting "partial mode". In the partial mode, the display controller fetches a rectangular subset of the frame buffer from the DRAM to its buffer (step 2). The rectangular subset



(a) Current frame (b) Next frame (c) Updated region

Figure 2: Example of a Display Content Change

should contain all updated parts.

Figure 2 shows an example of a rectangular subset of updated parts. Let us assume that the display content is updated from the current frame (Figure 2(a)) to the next frame (Figure 2(b)). In the example, there are two updated components; the red second hand and the alarm shown on the upper right corner. These updated parts are represented by two light gray boxes in Figure 2(c) and the rectangular subset, outlined by a dotted blue rectangle, is a larger box that contains the two updated components.

After user programs draw images with CPU and GPU (step 1) on the frame buffer in the main memory (DRAM), graphical middleware, X Window and Composite Manager, process the raw image and send the processed data to the kernel. Referring the processed data, the kernel along with related device drivers configures the display controller. Next, the display controller read the updated part from DRAM to its buffer (step 2). The display controller transfers the updated part to the internal RAM of the display panel via a hardware-to-hardware line called "display bus" (step 3). Finally, the display panel lays out the contents in the internal RAM on the screen. *LPD* enhances step 2 and step 3 procedures. Step 2 involves with main memory read, transfer on the main bus, and write into the buffer in the display controller. Step 3 consists of buffer read, transfer on the display bus, and write into the internal memory. Note that the final transmission to the screen (step 4) contains the whole frame buffer and is not reduced by *LPD*.

3.1 Simple Analysis of Expected Power Saving

In this section, we describe the rationale that led us to the design of *LPD* with a simple analysis of the expected power saving. Let P_u ($0 \leq P_u \leq 1$) be the proportion of the updated rectangle to the whole frame. Also, let f be the frame rate and S be the size of a whole frame, which is usually the product of width, height, and color depth. Then, T_L, the memory bandwidth that *LPD* consumes to transfer the updated rectangle from the main memory to the internal RAM through the display controller, is

$$T_L = P_u \cdot S \cdot f \tag{1}$$

The traffic without *LPD*, T_0, between the same components is:

$$T_0 = S \cdot f \tag{2}$$

Note that the bandwidth of DRAM read and main bus transmission, display bus transmission and internal RAM write are the same because we assume no compression or modifications in the transmission chain from the DRAM to the internal RAM. *LPD* is orthogonal to such operations and any benefits obtained by compression can be equally applied to *LPD* as well as to non-*LPD* schemes.

The updated contents should be readily available in the DRAM when the display controller accesses the DRAM because the controller is not aware of processor caches; i.e., there is no cache coherency support between CPU and controller. It also means that caches of processors cannot be involved and every bit read, moved, or written with the display controller or the display panel is a direct memory-to-device or device-to-device operation. Therefore, we can assume that the power consumed in memory read, transfers on the main bus, and transfers on the display bus are not affected by caching. P_L and P_0, the power consumed by *LPD* and non-*LPD* schemes, respectively, are given as

$$\begin{aligned} P_L &= C \cdot (T_L) \\ P_0 &= C \cdot (T_0) \end{aligned} \tag{3}$$

, where C is a coefficient representing the sum of the energy consumption rates of all involved operations. The total power saved by *LPD*, P_{save} is:

$$\begin{aligned} P_{save} &= P_0 - P_L \\ &= C \cdot T_0 - C \cdot T_L \\ &= C \cdot (1 - P_u) \cdot S \cdot f \end{aligned} \tag{4}$$

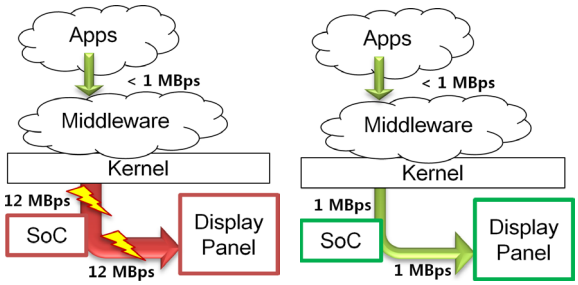
This shows that the power saving is proportional to P_u, the proportion of updated regions. As we can see in Eq. (4), *LPD* enjoys greater savings with devices with higher resolutions and higher frame rates. Note that mobile displays have undergone disruptive technology advances in the last decade and this trend may continue in near future; recent mobile phones have displays of 1920x1080 resolution or higher at 60 fps.

In Section 5, we show the effectiveness of *LPD* with a series of experiments with Samsung Gear 2. We also show how the model driven in this section fits with the experimental results.



(a) Current frame (b) Next frame (c) Torn frame

Figure 3: Screen Tearing Example



(a) Without *LPD* (b) With *LPD*

Figure 4: The Concept of *LPD*

3.2 Overhead of Brute Force Mechanisms

In this section, we analyze the potential overhead of *LPD* similar mechanisms implemented in a brute force style. Instead of using the processed information provided by middleware, these methods identify the updated regions by frame by frame comparison.

Method 1. Compare each pixel to identify updated regions. This requires reading two frames and the required memory bandwidth, M_r, is

$$M_r = S \cdot f \cdot 2 \tag{5}$$

The maximum benefit due to reduced transfer is achieved when there are no updated regions. The maximum benefit is $M_r/2$ and the overhead overwhelms the benefit.

Method 2. Compare CRC values of frame buffer blocks. This is what Transaction Elimination does [4, 16] with an additional hardware for GPU to main memory transmissions. If we perform the same operation with software, we need to read a whole frame once and should calculate CRC at the speed of memory bandwidth. The overhead still overwhelms the benefit as well.

As indicated above, brute force mechanisms that identify the differences based on frame-by-frame comparison are inappropriate. Note also that hardware-based approaches [4, 16] incur inevitable overheads of gate count, energy, and licenses.

3.3 Screen Tearing and Tearing Effect

Screen tearing may appear if the image transfer from the display controller to the internal RAM is not properly synchronized with the display refresh by the MCU. Figure 3 shows an example of screen tearing.

One scenario that causes the screen tearing of Figure 3 is as follows. While the MCU is scanning its internal RAM containing the current frame for the display refresh, the display controller transfers the next frame overwriting on to the internal RAM. If the speed of display controller transfer is faster than the display refresh, part of the internal RAM that is not displayed may be updated with the next frame. As a result, the screen shows a mix of both frames as depicted in Figure 3(c).

A display panel generates a tearing effect (TE) signal to notify the kernel that the panel has completed drawing the image from its internal RAM. A display controller should start sending the next frame to the display panel after receiving the TE signal and should complete the transmission before the MCU starts to refresh the next frame. In other words, steps shown in Figure 1 should be synchronized with the TE signal.

In Section 4.2.3, we discuss the issue of screen tearing in a greater detail. Screen tearing becomes more serious with *LPD* as the transfer latency becomes less deterministic and device drivers are required to add operations with exact timing. Section 4.2.3 describes the synchronization mechanism that *LPD* uses to mitigate the issue.

4 Design and Implementation

Figure 4 shows the design concept of *LPD*. *LPD* utilizes the information already known to applications and middleware to reduce the amount of information handled by hardware components. Suppose a device with 320x320 resolutions. With 4 bytes per pixel and 30 frames per second, the amount of information required to transfer is about 12 MB/s. If we further assume that 8% of a frame is updated on the average, then the required bandwidth for the updated regions is about 1 MB/s. Without *LPD*, the required bandwidth from the software stack to the display panel is still 12 MB/s because full frames are transferred regardless of updated regions. With *LPD*

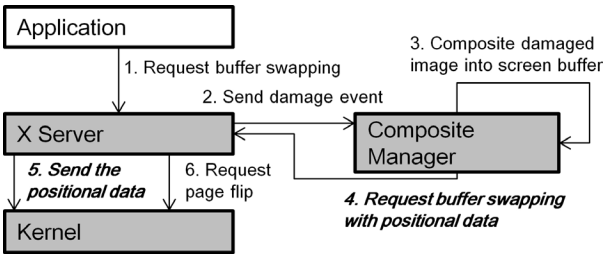


Figure 5: Interaction of Middleware and the Linux Kernel for *LPD* implementation

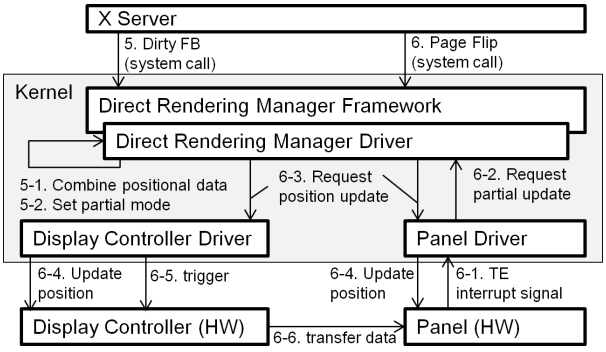


Figure 6: Interaction of Linux Kernel and Hardware

as shown in Figure 4(b), the bandwidth is reduced to 1 MB/s. A System-on-Chip (SOC) usually has processors, main memory, and a display controller. Let us examine the procedure of *LPD* from the top to the bottom and the issues we have encountered in the course of *LPD* implementation.

4.1 Userspace Middleware Interaction

Figure 5 shows how the window system (X Server) and the composite manager in userspace interact with applications and the kernel. The numbers in the interaction vectors denote the sequence of events. The shaded boxes and descriptions in italic are the components affected and interactions modified by *LPD*, respectively. Such modifications allow the kernel to have the information required to identify the updated regions. *LPD* does not require additional modifications in applications or other middleware components.

The sequence of interactions in userspace flows as follows:

1. An application requests a buffer swap to the X Server.
2. The X Server notifies a damage event to the composite manager. Each damage event contains the positional data of an updated region.
3. The composite manager composes the screen image with the damage event information provided in step 2.
4. The composite manager requests a buffer swap to the X Server. In *LPD*, this request includes positional data of updated regions. In non-*LPD*, this request does not include any information.
5. In *LPD*, the X Server transfers the positional data to the kernel with the "Dirty FB" kernel interface described in Section 4.2.1. In non-*LPD*, this step is skipped.
6. The X Server requests a page flip to the kernel so that image data can be sent to the screen with "Page Flip" interface described in Section 4.2.2.

As shown, the modification in the middleware is minimal and the backward compatibility of the modified userspace components is preserved. Because the composite manager has been already tracking the updated regions (or damaged regions in their notation) in order to optimize rendering performance, we simply modified the composite manager to report back what it already comprehends as one aggregated updated region. Then, the X Server just relays the information. With such simple and straightforward notifications, we can enjoy the benefit of reduced data bandwidth. It is worth to note that *LPD* incurs constant computational and space complexity.

4.2 Kernel Interaction

As mentioned earlier, we use two userspace-to-kernel interfaces: Dirty FB and Page Flip. The detailed in-kernel operations of the two interfaces are described in Figure 6. The Dirty FB triggers sub-steps 5-1 and 5-2 and the Page Flip interface involves with sub-steps 6-1 to 6-6. Note that some interfaces are not software-driven interactions. For example, 6-1 is an interrupt from hardware and 6-6 is a hardware-to-hardware transmission.

The two kernel interfaces, Dirty FB and Page Flip, are not new or non-standard interfaces but are standard Linux kernel interfaces that had been kept in the mainline. We also do not change the semantics of the interfaces. Note that being a standard interface is not coincident with the popular or frequent use of the interface. Both the Dirty FB interface and Page Flip interface are seldom used or not fully used.

We obeyed the syntax and semantics of Linux mainline interfaces. *LPD* is easily upstream-able and reusable

by other device drivers in various kernel versions by different vendors. The upstream-ability and the induced compatibility add yet another benefit to *LPD*: maintainability, which enables us to let the open source community maintain *LPD* along with later versions of Linux kernel and additional device drivers. We expect that we can upstream all the required pieces to the mainline Linux kernel soon.

4.2.1 Dirty FB

Dirty FB, a kernel-userspace interface, allows the X Server to send multiple sets of updated regions (rectangle forms consist of the left-top and right-bottom coordinates) to the kernel DRM driver before the X Server issues Page Flip. Without *LPD*, the X Server does not need to use the Dirty FB interface because the X Server assumes that a whole frame is updated. The operation of Dirty FB consists of the following steps as shown in Figure 6.

Step 5: The X Server sends one or multiple updated regions to the kernel DRM driver.

Step 5-1: The DRM driver merges input regions into a single rectangle that contains all updated regions. The larger box with a dotted blue outline in Figure 2(c) represents the aggregated single rectangle.

Step 5-2: The DRM driver remembers the coordinates of the aggregated update region and uses "partial mode" for the next frame transmission.

Most embedded display controllers can transfer image data of a single rectangular region to the display panels in one single transfer. For each TE interrupt signal, the display controller can conduct one transfer only and there is only one TE interrupt signal per display refresh. Therefore, in order to avoid image quality deterioration due to frame drops, *LPD* should combine multiple updated regions into one.

Let the left-top coordinate and the right-bottom coordinate of each updated region be $L = (L_x, L_y)$ and $R = (R_x, R_y)$, respectively. Each updated region can be expressed by a pair of L and R. Then, L' and R', the left-top and the right-bottom coordinates of the aggregated updated region covering n updated regions are derived as:

$$\begin{aligned} L' &= (min(L_{x1}, \ldots, L_{xn}),\ min(L_{y1}, \ldots, L_{yn})) \\ R' &= (max(R_{x1}, \ldots, R_{xn}), max(R_{y1}, \ldots, R_{yn})) \end{aligned} \quad (6)$$

, where $L_i = (L_{xi}, L_{yi})$ and $R_i = (R_{xi}, R_{yi})$ are the left-top and the right-bottom coordinates of the i-th updated region.

4.2.2 Page Flip

In non-*LPD*, the window system requests a frame buffer change via the Page Flip interface. An invocation of Page Flip updates the memory address to the requested frame buffer of the display controller hardware. Then, the display controller may access the requested frame buffer by setting a trigger bit after a TE signal is issued.

If the display controller is in a partial mode (*LPD* enabled), the Page Flip behavior is slightly different because we cannot simply switch frame buffers for each frame. Instead of transferring the whole frame buffer, the controller transfers the updated region only. In the partial mode, configured by *LPD*, a Page Flip request updates the relevant registers (sub-step 6-4) that include the memory base and the offset address to the updated region, start and end positions of the overlay, and the line size. Note that the partial mode does not require the display controller to support input/output memory management unit (IOMMU). The partial mode only requires the controller to access a rectangular subpart of a frame buffer. It does not depend on whether the frame buffer is in a physically contiguous memory chunk (conventional DMA) or in a virtually contiguous memory chunk (DMA with IOMMU).

In the partial mode, like the Page Flip request, a TE interrupt signal (sub-step 6-1) initiates the update of MCU registers that includes the start and end coordinates of the internal RAM. Note that a Page Flip request activates *LPD* if Dirty FB has been called after the previous Page Flip request. Otherwise, the kernel DRM subsystem assumes that the user wants to replace the whole contents.

As shown in Figure 6 with the sub-steps from 6-1 to 6-4, the TE interrupt (6-1) allows the panel driver to request a partial update to the kernel DRM driver (sub-step 6-2). Then, the kernel DRM driver requests position updates to both display control driver and panel driver (sub-step 6-3) that commonly are sub device drivers of the DRM driver. Then, these two sub drivers update positional information of their corresponding hardware (sub-step 6-4). Lastly, the display controller driver commands the display controller (sub-step 6-5) to initiate the data transfer (sub-step 6-6).

4.2.3 Prevention of Screen Tearing

While implementing *LPD* on experimental devices, we have experienced screen tearing. Without *LPD*, because frame data transmission times are long and deterministic, careful manipulation of the display controller is not required and the tearing is not an issue. A mechanism to

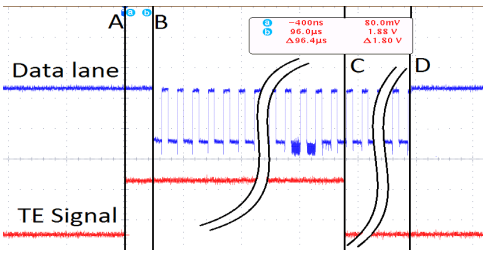


Figure 7: Data and TE Signals

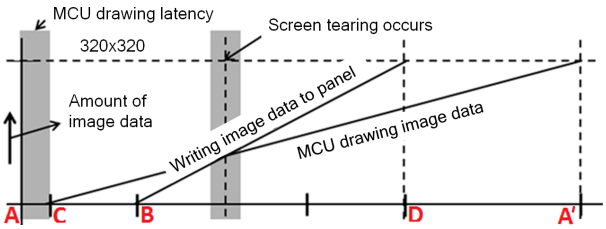


Figure 8: Screen Tearing with Faster Image Data Transfer

prevent screen tearing under varying frame data transfer latency has been implemented and included in *LPD*.

LPD uses the TE signal and its handler to prevent screen tearing. Figure 7 shows the timing of data lane signal of the display bus controller and the TE interrupt signal of the display panel. The TE signal notifies when to transfer the next frame.

When the TE signal occurs at point A in Figure 7, the MCU of the display panel has completed drawing the contents of its internal RAM to the screen. About 96 μs later, at point B, the display controller initiates the data transfer to the display panel. Point C denotes the time when the MCU of the display panel has completed drawing the contents. Point D denotes the time when the display controller has completed writing the contents to the display panel.

LPD classifies the events that cause screen tearing into two classes.

- Case 1. The display controller speed is slower than the drawing speed of the MCU.
- Case 2. The display controller speed is faster than the drawing speed and an image data transfer (at point B) starts while drawing the previous frame (point C already occurred). This case is depicted in Figure 8. The markers (A to D) in both figures denote the same types of events. A' denotes the next A event.

In order to prevent the first case, *LPD* completes configuring every related device between A and B and sets the display controller faster than the drawing speed of MCU. In order to prevent the second case, *LPD* ensures that B starts after A and before C.

Another issue with *LPD* arises when multiple hardware overlays are applied. Samsung Gear 2 supports up to five overlays although it mostly uses only one. If we use multiple hardware overlays simultaneously, the display controller sends a merged image from multiple virtual frame buffers (hardware overlays) to the panel. The current implementation of *LPD* does not support aggregate updated regions across multiple hardware overlays. Therefore, if multiple hardware overlays are used, the transfer mode should be fixed to full screen mode before the display controller starts to transfer image data to the display panel. *LPD* configures the transfer mode to partial mode (*LPD* enabled) if a single hardware overlay is used and configures to full screen mode if multiple overlays are used. *LPD* checks if the partial mode may be enabled or the full screen mode should be enabled based on the Page Flip request. In order to support multiple hardware overlays, *LPD* should be updated to track the origin point of each hardware overlay.

5 Experiments

We have examined the functionality and performance of *LPD* by conducting experiments on Samsung Gear 2. The hardware specifications of Gear 2 are as follows:

- Display type & size: AMOLED, 1.63 inch
- Resolution: 320x320
- Frame rate: 30 FPS
- Application Processor (SoC): Exynos 3250
  - CPU: Dual ARM Cortex A7 1.0 GHz
  - GPU: Mali-400 MP
  - Main Memory: 512 MiB LPDDR3 DRAM

We have conducted two different sets of experiments. The first set of experiments is performed with a synthetic power consumption benchmark; a benchmark application runs directly on the Linux kernel without the X server window system. In the first set of experiments, we varied the size of updated regions.

The second set of experiments involved with publicly released Tizen wearable applications: W-launcher, Heart Rate, Setup-wizard, and Voice Memo. In most cases, these applications draw objects of sizes: 320x320, 192x169, 96x80, and 64x34, respectively. The purpose of the second set of experiments is to validate the

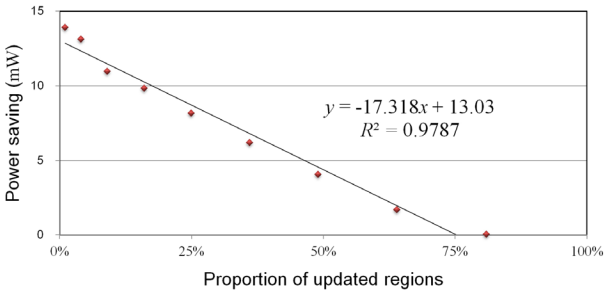


Figure 9: The Power Saving

effectiveness of *LPD* in the real-world environments on commercialized products. Note that *LPD* does not require any modifications in applications and *LPD* is applicable to any Tizen devices or X Window systems with Linux DRM and the `i80` display interface.

For each test case, we have conducted three experimental runs. Each experimental run consists of a continuous execution for 30 seconds. In order to get the average value of a continuous execution, an in-house power measurement device accumulates the energy consumed via the battery connection (supplied by V_{BATT}) and shows the average power over the 30 seconds.

The in-house power measurement device samples the current every 0.2 ms with the range of 0.6 mA to 4 A in the 0.01 mA granularity and with less than 1% of error. The measurement device sends the data to a tablet or a PC via a Bluetooth connection in real-time and allows the tablet or the PC to visualize or later analysis of the accumulated data. We have supplied 4.0 V to the device constantly in order to make the measurement and analysis simple.

Due to the technical difficulty, we have measured the whole power consumption of the device, not the power consumed by the display system only. The power consumed by other non-related devices such as GPU and network adapters is included. Thus, any visible power saving in the experiments is significant enough to motivate the adaption to commercial products; engineers invest huge effort and time to get additional minutes of battery life. If it is an extra hour for a 72-hour device, the responsible engineer may even be called a *hero* by his or her colleagues.

5.1 Synthetic Workload Benchmark Result

Table 1 and Figure 9 show the amount of power that *LPD* saves in the first set of experiments with the synthetic workload benchmark. As shown in the column

Table 1: Power-wise Synthetic Benchmark Result

Updated regions	Control (mW)	LPD (mW)	Power saving (mW)	Power saving (%)
320x320	209.03	208.99	0.04	0.02%
288x288	203.08	203.04	0.04	0.02%
256x256	196.84	195.18	1.66	0.84%
224x224	192.01	187.95	4.06	2.11%
192x192	187.83	181.64	6.20	3.30%
160x160	184.27	176.10	8.17	4.43%
128x128	181.43	171.60	9.84	5.42%
96x96	179.06	168.10	10.96	6.12%
64x64	177.35	164.25	13.10	7.39%
32x32	176.47	162.58	13.89	7.87%

of Updated regions of Table 1, we have executed the test application with various sizes of updated regions while fix the frame rate to 30 FPS. The results illustrate that the less data (= less updated region size) the display controller transfers, the more power saving we can get. In order to show the power-wise overhead of *LPD*, we have experimented with the full screen update that corresponds to the "320x320" row in Table1. In this test case, *LPD* cannot provide any benefit but incurs overheads only. However, the test results indicate that the power-wise overhead induced by *LPD* is ignorable. Surprisingly, *LPD* reduces 0.02% of power consumption. Because *LPD* adds a few CPU cycles per frame, we suspect that errors in power meters or variances in the experiments such as the temperature are responsible for this result.

Figure 9 shows power saving as a function of the update region size. We fit the observation point with a linear line in order to see if the amount of power saving is linearly related to the size of updated region as Eq. (4) suggests. The linear equation embedded in Figure 9 has the goodness-of-fit value of 0.98. Such a value implies that the model fits with the experimental results well.

The power saving of *LPD* appears to be more than expected if the proportion of updated regions is very small: the two left-most points in Figure 9. We speculate that the difference is due to the DVFS mechanism on the memory bus and memory interface. A DVFS mechanism for the memory bus and interface [9] can further save power by lowering the voltage and the frequency if the memory transmission is reduced. With the lower voltage and frequency, the energy consumption is no more linear to the memory bandwidth.

Based on the experiments, we can conclude that *LPD*

Table 2: Power Reduction of Real-World Applications

App	*LPD* power saving		Reduced traffic
	mW	%	kiB/s
W-launcher	0.22	0.20	-
Heart rate	0.39	0.58	134.6
Setup-wizard	1.14	1.52	7178.0
Voice memo	2.70	2.98	7165.5

is successful in saving energy when the proportion of updated region is small. Especially, if *LPD* is applied to smart phones or tablets equipped with higher resolution displays, the energy saving will be greater as suggested by this experiment and the model summarized in Eq. (4).

5.2 Experiments with Real-World Applications

Table 2 shows how much power *LPD* saves with actual applications running on Samsung Gear 2. In Table 2, the two sub-columns of "*LPD* power saving" show power saving in absolute values (mW) and in relative values (%). The column of "Reduced traffic" shows the memory bandwidth reduction. Table 2 suggests that *LPD* reduces power consumption of commercial applications running on a commercial product as well.

5.3 Overhead of *LPD*

LPD requires a few additional lines of codes in the middleware and kernel device drivers based on DRM. Therefore, *LPD* incurs additional overhead. We can infer the energy overhead of *LPD* by activating *LPD* for the cases where *LPD* is completely useless; i.e., the whole screen is updated every frame. For example, in Table 1, the case that "320x320" is updated represents such a case. As shown in Table 1, the energy overhead induced by *LPD* is -0.0462 mW. This result implies that the overhead of *LPD* is too scanty that the overhead is obscured by environmental variances. This is consistent with the amount of instructions added for the implementation of *LPD*; i.e., only several lines of trivial arithmetic instructions without loops or context switches are added to device drivers and middleware.

6 Future Work and Implications

LPD has been released with the X Window-based Tizen 2.3 commercial device, Samsung Gear S. However, in later versions, Tizen plans to use Wayland instead of the X Window System [23]. In order to keep the benefit of *LPD* for later Tizen versions, we will need to implement *LPD* on top of Wayland.

Further enhancement of *LPD* may draw out additional power conservation. That is, *LPD* may improve further by utilizing the characteristics of the DVFS-capable display bus such as MIPI-DSI. MIPI-DSI controller has various control modes: HSM (High Speed Mode), LPM (Low Power Mode), and ULPM (Ultra Low Power Mode). With a lot of display updates such as video playing, MIPI-DSI needs to operate at HSM, which supports bandwidth from 80 Mbps to 1 Gbps. Eliminating the transfer of unchanged regions, *LPD* may be able to reduce the bandwidth less than 80 Mbps. Then, MIPI-DSI can operate in the LPM mode, which consumes significantly less power than HSM.

LPD is an excellent example of vertical optimization that involved with several layers of the system. By allowing the kernel to accept simple yet performance critical hints that are readily available at middleware, we are able to use the given hardware more efficiently with minimal modifications and without any kernel hacks that deteriorates the maintainability of software. As an example of vertical optimization, *LPD* suggests that operating system architects should be well aware of the information that its upper layers have–the updated regions of the window system–and what its lower layers want–the `i80` display panel in *LPD*. *LPD* suggests that well-designed co-operation between multiple layers is extremely important.

LPD depends on the ability of the window system to recognize updated regions of the screen. The current implementation of the X Server depends on the correct operation of applications. That is, if an application declares that the whole screen is updated even though only parts of the screen are actually updated, and then *LPD* cannot save power. This implies that educating application developers for proper implementation or provision of a proper SDK tool is critical in deploying *LPD* and power saving. This indicates further need for vertical optimization going up through SDK, tools, and applications. Another aspect is that the UX design is extremely important in power saving; i.e., the updated region size of each frame matters significantly. We may conjecture that vertical power optimization should be extended even to UI/UX designs, which is already becoming important with the adoption of AMOLED displays; AMOLED consumes power differently depending on the colors and brightness of pixels.

7 Conclusion

LPD can lessen power consumption induced by memory operations and data transfers related with frame buffers. The first implementation of *LPD* has been applied to Samsung Gear 2 for the experimentation purpose. After confirming the stability and usability of *LPD*, we have successfully commercialized it for Gear S and released the complete source code for the public access. Even though we confined *LPD* to wearable Tizen devices only, contributing *LPD* to the mainline Tizen might be a trivial process. We are also ready to upstream *LPD* to the Linux kernel community and the infrastructural patch for *LPD* has been submitted and merged to the DRM tree for Linux 3.16. The main body of *LPD* is to be upstreamed to the Linux kernel community afterwards. Because *LPD* is not a compatibility breaking kernel hack, but a mainline upstream-able kernel feature, any Linux-based devices with the popular `i80` display interfaces can use *LPD* to save power.

The experimental results have shown that *LPD* saves significant amount of energy for wearable devices. If we save 5% of total energy for a device with 72 hours of life time, we extend additional 3 hours and 36 minutes of the life time. Besides, as discussed in Section 3, the energy saved by *LPD* might be larger for mobile devices with higher resolutions. More significantly, *LPD* does not require any modifications in hardware as long as the device has the de facto standard, `i80`. *LPD* does not incur noticeable overhead in CPU and *LPD* does not affect the visual quality of the display at all. Finally, *LPD* may be used with other display power saving mechanisms independently without any modifications in user applications.

8 Acknowledgments

We would like to thank Dr. Jong-Deok Choi, Dr. Hyogun Lee, and Dr. Sang-bum Suh for the support and advices. We would also like to express our special thanks to YoungJun Cho, a kernel graphics expert, who has been participated in the implementation and test of *LPD* for the commercialization of Samsung Gear series. We would like to show our gratitude to the other Tizen kernel and system framework developers for their commitment in the development of Tizen and its products. Comments from the anonymous reviewers, Dr. Chong-kwon Kim, and Geunsik Lim were extremely helpful in revising the paper.

9 Availability

LPD has been used for Samsung Gear S product running Tizen. Both userspace and kernel codes for Gear S, including *LPD*, are available for the public. You can access the kernel code for Gear 2 with *LPD* in the same site as well:

`http://opensource.samsung.com/`

If readers want to look at, understand, and contribute the *LPD*-related code, they may want to access the repositories of Tizen after creating an account at `http://tizen.org/`, which is opened to the public and operated by the Linux Foundation:

`http://review.tizen.org/`

References

[1] ANAND, B., THIRUGNANAM, K., SEBASTIAN, J., KANNAN, P. G., ANANDA, A. L., CHAN, M. C., AND BALAN, R. K. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 57–70.

[2] ARM LTD. Adaptive scalable texture compression. `http://www.arm.com/products/multimedia/mali-technologies/adaptive-scalable-texture-compression.php`. Accessed: 2015-01-30.

[3] ARM LTD. Arm frame buffer compression. `http://www.arm.com/products/multimedia/mali-technologies/arm-frame-buffer-compression.php`. Accessed: 2015-01-06.

[4] ARM LTD. Transaction elimination. `http://www.arm.com/products/multimedia/mali-technologies/transaction-elimination.php`. Accessed: 2015-01-06.

[5] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 21–21.

[6] CHANG, N., CHOI, I., AND SHIM, H. Dls: Dynamic backlight luminance scaling of liquid crystal display. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI) 12*, 8 (August 2004), 837–846.

[7] CHENG, W.-C., HOU, Y., AND PEDRAM, M. Power minimization in a backlit tft-lcd display by concurrent brightness and contrast scaling. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (Washington, DC, USA, 2004), DATE '04, IEEE Computer Society, pp. 10252–.

[8] CHOI, I., SHIM, H., AND CHANG, N. Low-power color tft lcd display for hand-held embedded systems. In *Proceedings of Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)* (August 2002), pp. 112–117.

[9] CORBET, J. Better device power management for 3.2. *LWN* (Nov. 2011). `http://lwn.net/Articles/466230/`.

[10] CROXFORD, D., JONES, S., AND FLORDAL, O. Adaptive frame buffer compression, Nov. 2013. US Patent App. 13/898,510.

[11] GATTI, F., ACQUAVIVA, A., BENINI, L., AND RICCO', B. Low power control techniques for tft lcd displays. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (New York, NY, USA, 2002), CASES '02, ACM, pp. 218–224.

[12] HUNG, L., AND CHEN, C. Recent progress of molecular organic electroluminescent materials and devices. *Materials Science and Engineering: R: Reports 39*, 5 (2002), 143 – 222.

[13] JAPANESE SUPREME COURT. Precedent of a civil case of a consumer vs. panasonic. `http://www.courts.go.jp/app/files/hanrei_jp/686/080686_hanrei.pdf`, 2011. Accessed: 2015-01-06.

[14] KIM, H., CHA, H., AND HA, R. Dynamic refresh-rate scaling via frame buffer monitoring for power-aware lcd management. *Softw. Pract. Exper. 37*, 2 (Feb. 2007), 193–206.

[15] NYSTAD, J., LASSEN, A., POMIANOWSKI, A., ELLIS, S., AND OLSON, T. Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2012), EGGH-HPG'12, Eurographics Association, pp. 105–114.

[16] OTERHALS, J., CROXFORD, D., ERICSSON, L., NYSTAD, J., AND LILAND, E. Graphics processing systems, May 2011. US Patent App. 12/923,518.

[17] PANASONIC. 18650 cell scope / high power density trend. `http://www.slideshare.net/GIPC2011/20140207-panasonic-b2-b-gipc`, 2014. Accessed: 2015-01-27.

[18] PARK, J., CHUN, B., CHOI, Y., AND LEE, J. Method of reducing power consumption and display device for reducing power consumption, Aug. 14 2014. US Patent App. 13/928,334.

[19] PASRICHA, S., LUTHRA, M., MOHAPATRA, S., DUTT, N., AND VENKATASUBRAMANIAN, N. Dynamic backlight adaptation for low-power handheld devices. *IEEE Des. Test 21*, 5 (Sept. 2004), 398–405.

[20] SHIM, H., CHANG, N., AND PEDRAM, M. A compressed frame buffer to reduce display power consumption in mobile systems. In *Proceedings of Proceedings of the Conference on Asia South Pacific Design Automation (ASP-DAC)* (January 2004), pp. 818–823.

[21] SINGHAR, A. Smart backlights to minimize display power consumption based on desktop configurations and user eye gaze, Apr. 1 2014. US Patent 8,687,840.

[22] THE MOVING PICTURE EXPERTS GROUP. Mpeg standards. `http://mpeg.chiariglione.org/standards`. Accessed: 2015-04-28.

[23] VANCUTSEM, J. Tizen ivi 3.0-m1 released. `https://lists.tizen.org/pipermail/ivi/2013-July/000563.html`, 2013. Accessed: 2015-01-19.

[24] WHELAN, R., AND GRINDSTAFF, M. Low power display refresh, Oct. 7 2004. US Patent App. 10/407,758.

Memory-Centric Data Storage for Mobile Systems

Jinglei Ren
Tsinghua University[*]

Chieh-Jan Mike Liang
Microsoft Research

Yongwei Wu
Tsinghua University[*]

Thomas Moscibroda
Microsoft Research

Abstract

Current data storage on smartphones mostly inherits from desktop/server systems a flash-centric design: The memory (DRAM) effectively acts as an I/O cache for the relatively slow flash. To improve both app responsiveness and energy efficiency, this paper proposes MobiFS, a memory-centric design for smartphone data storage. This design no longer exercises cache writeback at short fixed periods or on file synchronization calls. Instead, it incrementally checkpoints app data into flash at appropriate times, as calculated by a set of app/user-adaptive policies. MobiFS also introduces transactions into the cache to guarantee data consistency. This design trades off data staleness for better app responsiveness and energy efficiency, in a quantitative manner. Evaluations show that MobiFS achieves 18.8× higher write throughput and 11.2× more database transactions per second than the default Ext4 filesystem in Android. Popular real-world apps show improvements in response time and energy consumption by 51.6% and 35.8% on average, respectively.

1 Introduction

App experience drives the success of a mobile ecosystem. Particularly, responsiveness and energy efficiency have emerged as two new crucial requirements of highly interactive mobile apps on battery-powered devices.

Recent work has shown the impact of data storage on app experience. Storage I/Os can slow down the app responsiveness by up to one order of magnitude [11, 19, 28, 36], and can substantially impact the device's energy consumption either directly or indirectly [29, 38, 50].

Modern mobile platforms typically inherit their data storage designs from desktops and servers. For example, Android and Windows Phone 8 currently default to the Ext4 and NTFS filesystem, respectively. However, these data storage designs neither reflect the different requirements nor exploit the unique characteristics of smartphones. The limited number of foreground apps, increasingly adequate DRAM capacity, and the networked nature of most apps open up a new design space that is not applicable to desktops and servers.

In this paper, we advocate a **memory-centric design** of data storage on smartphones. To elevate energy efficiency and app responsiveness as first-class metrics, we switch from the traditional flash-centric design to a memory-centric design. The flash-centric design, derived from desktops/servers, assumes the persistent flash medium as the primary store, and regards the memory (DRAM) as a temporary cache. Most recent optimizations [16, 20, 22, 36, 37, 38, 51] still follow this traditional philosophy. In contrast, we re-examine the underlying assumption of mobile storage design. Our memory-centric design views the memory as a long-lived data store, and the flash as an archival storage layer. Concretely, (1) frequent writebacks of in-memory dirty data become unnecessary; (2) individual file data synchronization calls (typically, `fsync`), which are costly, can be safely aggregated and scheduled out of the critical path of app I/O. Both changes have significant implications on app responsiveness and energy efficiency [11, 19, 28, 36]. Instead, we incrementally checkpoint app data at optimal variable intervals that adapt to app behavior, user interactions and device states.

Our key idea is to **trade off durability for energy efficiency and app responsiveness**. To realize the tradeoffs in a quantitative way, we interpret durability as a continuous variable, instead of a binary discrete variable (durable or not). Intuitively, given a specific probability of system failure, the less stale the persistent version is, the more "durable" the data is. Therefore we use data staleness as the metric of durability. Besides, these tradeoffs rely on the decoupling of durability and consistency in storage. Recent efforts have explored a similar decoupling in different domains [6, 35], but they do not apply to our memory-centric design, and they do not show how to optimize the tradeoffs for mobile apps. Overall, there is a lack of systematic and quantitative studies on these tradeoffs in mobile systems.

The gains from our design mainly come from its adaptability to mobile app behavior and usage. The tra-

[*]Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Beijing; Research Institute of Tsinghua University, Shenzhen.

ditional `fsync` and fixed flush intervals are not suited for optimizing energy performance in mobile systems and often negatively impact user experience. Instead, our measurements motivate an *app/user-adaptive* design of checkpointing for mobile apps. Concretely, we answer the algorithmic questions regarding *what* in-memory data to checkpoint (i.e., save to flash), and *when* to do so. Our solution determines the ideal checkpointing times for each app independently, by considering both device states and user interactions.

Meanwhile, loosening the timing of checkpointing is feasible in the mobile context because smartphones exhibit favorable properties. First, being self-contained (e.g., powered by batteries), smartphones are less exposed to losing data on volatile memory caused by external factors (e.g., power loss). Second, both hardware and software advances lower the data loss probability due to system crashes. Only 6% of users experience system failures more than once per month, according to our online survey. Third, most mobile apps are networked, meaning that their data are recoverable from remote servers (e.g., Gmail, Facebook, Browser). We investigate 62 most popular free apps on Google Play, and only 8 are vulnerable to local data loss (Section 3)[1].

To manipulate the checkpointing time, we change the semantics of POSIX `fsync` to be asynchronous. For some apps and databases (e.g., SQLite) that rely on synchronous `fsync` to guarantee data consistency, we design *Versioned Cache Transactions* (VCTs) to enforce atomic transactions on the filesystem cache. Combining adaptive checkpointing and VCT ensures data consistency while minimizing overhead caused by periodically frequent writebacks.

We implement our filesystem, *MobiFS*, at the system call layer for three reasons. First, this position is below upper-layer apps and databases, so it allows MobiFS to intercept and manage all storage I/O. At the same time, we can selectively enable/disable our features for individual apps[1]. Second, our solution does not alter standard filesystem interfaces, so no changes to upper-layer apps are necessary. Third, our solution is agnostic to underlying flash management implementation. For example, it can be integrated with Ext4 [33], Btrfs [45] or the latest F2FS [26].

In summary, we make multiple **contributions**. (1) We establish the feasibility and significance of the memory-centric design for mobile storage. (2) We exploit, in the mobile context, the tradeoffs between data staleness, energy efficiency and app responsiveness. MobiFS introduces transactions to the page cache of regular filesystems, without modifying app storage interfaces. (3) We propose a new measure to quantify the trade-off between data staleness and energy efficiency, and characterize various I/O patterns of apps. These empirical results drive a policy framework that organizes and balances multiple factors – data staleness, energy, and responsiveness. (4) We implement a fully working prototype integrated with both Ext4 [33] and Btrfs [45]. Experiment results suggest up to 35.8% reduction in energy consumption and 51.6% improvement on app responsiveness, as compared to the default Android filesystem. It also achieves 18.8× higher write throughput and 11.2× more database transactions/sec.

[1] Apps with critical data (e.g., unreproducible photos) can still opt to use a regular flash partition.

2 Background

Filesystem and Page Cache. A typical filesystem consists of three main components: (1) the interface routines to serve system calls; (2) the in-memory cache of hot data, typically the *page cache*; (3) the management of the persistent media. Our work revisits two of these parts: the system call routines and the cache.

Traditional filesystems with POSIX [15] interfaces use two ways to minimize data staleness and guarantee consistency while optimizing I/O performance. (1) Asynchronous `write`[2] moves data into the page cache. Dirty pages are written to flash after a small fixed time interval (default is 5 seconds in Android). (2) Synchronous `fsync` immediately enforces data persistence of the specified file. Databases rely on `fsync` to maintain consistency. Take write-ahead logging for example: the database first records updates in a separate log, without affecting the main database file, and then invokes `fsync` over the log. This ensures a consistent state of the log file in persistent media. Finally, logged changes are applied to the main database file.

Data Consistency and Staleness. A system failure may lead to data loss in the page cache, with essentially two negative outcomes: inconsistency and staleness. Consistency in this paper refers to *point-in-time consistency* [44], meaning that the persistent data always corresponds to a point of time T in the write history – all writes before T are stored in flash and all writes after T are not. Asynchronous `write` can not guarantee consistency. Specifically, when data is kept in the page cache, it may be overwritten and results in writes being reordered. If only partial cache is flushed before a system crash, the in-flash data could violate point-in-time consistency.

Meanwhile, *data staleness* is typically less of a concern for most apps in the case of a system crash. Data staleness is the "distance" between the current volatile in-memory data and the persistent in-flash data. This distance can be measured either with respect to versions [4] or time [43].

[2] For clarity, `write` only refers to an asynchronous one without special flags such as O_SYNC.

3 Insights

The memory-centric approach has become feasible on smartphones, as advances in both hardware and software make its underlying assumptions tenable.

Insight 1 *Memory capacity on smartphones is ample enough for app data storage.*

The DRAM capacity on modern smartphones has grown significantly (8× since 2010, from 512 MB to 4 GB), with 2 GB being the standard today. This amount of memory is already sufficient to run Windows XP on a desktop. Although app data requirement has also been increasing, it has been doing so in a slower pace. For example, typical web page requests have increased in size by only 94% during the same time period [1]. Moreover, smartphone users tend to run a small number of active apps/services at the same time due to the limited screen size. Further evaluation can be found in Section 7.2.

Insight 2 *Storing app data on smartphone memory is not as risky as it sounds.*

First, smartphones have a battery power supply. Such battery-backed RAM (BBRAM) is regarded as reliable in the desktop/server setting [12, 47, 49]. Second, the reliability of smartphones has improved to the extent that memory data loss due to system failures is sufficiently rare. This observation is based on our online survey about the frequency of mobile system crashes (not app crashes) experienced by average users. Among all 117 users responding to the survey, only 6% encounter more than one failure per month, and the average frequency is once per 7.2 months. Third, most apps store data on online services or the cloud anyway.

Our detailed case study of the top 62 free apps in the Google Play app store (covering all categories, representative of most popular and frequently used apps) well supports the observation above. At one extreme, there are apps that are always in sync with online servers, e.g., Facebook, Google Maps, Glide video texting, Fitbit and most games (so does Apple's Game Center). At the other extreme, some apps rely on local data exclusively or extensively, e.g., WhatsApp (for privacy protection) and Polaris Office. Data of these apps is vulnerable before being saved to flash. Meanwhile, there are apps in between these two extremes. For example, Skype may store messages on the server for "30 to 90 days" to synchronize states across multiple devices, so the data loss risk is negligible.

Overall, only 8 apps are counted as vulnerable to local data loss, for which a system crash may largely affect user experience. Users/developers have the flexibility of configuring these apps to use a regular flash partition with traditional `fsync`. Note that these exceptions only raise a slight *configuration* burden, rather than a *programming* burden. In our experience, an app-level configuration option is more practical and easy-to-use than enforcing new programming interfaces.

Insight 3 *Reducing the amount of data flushed to flash is one key to save app energy.*

First, the write energy dominates the app I/O energy. Prior measurements [5] have shown that reading consumes about 1/6 energy of writing for the same amount of data. Meanwhile, our system-call traces of Google Play top 10 apps suggest that the data amount of reads is only 41% of writes on average. Therefore, the overall read energy is only 6.3% of write energy.

Second, the amount of data to flush, rather than the number of batches, is the dominant factor of write energy. In our experiment, writing 40 MB data in batches ranging from 4 to 40 MB results in a net energy consumption difference within 1.5% on a Samsung smartphone. In addition, standby is not a good state for data flushing, because fixed overhead can be amortized if the device is active. Up to 129% extra energy is used if data is flushed after the device switches to standby.

In conclusion, considering that the total write data issued from an app is externally determined, we can focus on how much data is overwritten before flushing, as an indicator of the app's energy efficiency.

Insight 4 *Relaxing the timing of flushes is a key to app responsiveness.*

Flushing impacts app responsiveness in two ways: (1) When flushing in a `fsync` call, the app has to wait until the data is saved to the slow flash. This situation is encountered frequently [16, 28] as databases rely on `fsync`. (2) When flushing is invoked for background writeback, it competes for CPU cycles with active app workloads, as shown in [19, 36] and from our evaluation.

In either case, the timing of flushes plays a key role: if flushing is out of the `fsync` path to avoid user interaction/CPU peaks, its negative impacts on the app responsiveness would be minimal. Our memory-centric view leverages this insight.

Insight 5 *App/user-specific I/O access patterns suggest adaptive policies to balance the staleness-energy tradeoff, which can be achieved in a quantitative way.*

I/O access patterns can vary widely among apps/users. We follow three steps to quantify this variability as well as the key tradeoff. **(1)** We define a data staleness metric that is suitable to our context. Traditional definitions are with respect to either time [43] or versions [4]. However, the time-based staleness is hardly associated with energy efficiency, and there is no strict data versioning in a regular filesystem. Instead, we define the *data staleness*, s, as the total amount of data that an app has ever written since the last checkpoint. If an app writes two pages of data to the same address, the data staleness is increased by two

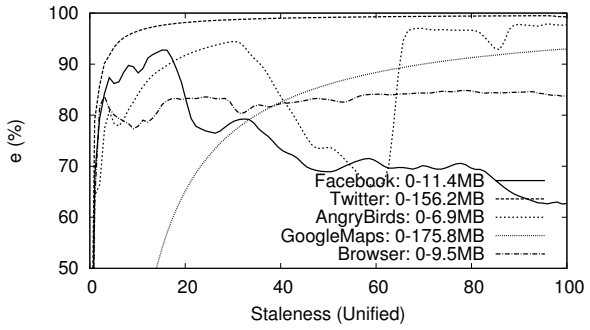


Figure 1: Different shapes of e curves suggest app-specific I/O patterns. The staleness ranges are unified to 0-100, and actual values are noted beside the app names.

pages (similar to the version-based staleness in this case). **(2)** Based on INSIGHT 3, we define the *energy efficiency ratio*, $e = o/s$, where o is the total amount of data that has been overwritten since the last checkpoint. As $o < s$, e is within [0, 1). A larger e indicates a larger proportion of pending write data to be merged before flushing and hence higher energy efficiency. **(3)** Based on the two definitions above, we draw the e curve over s to capture the extent to which increased staleness improves energy efficiency. The different shapes of e curves in Figure 1 suggest the optimal flushing time is different for different apps (and also for different users). Ideally, data in memory should be flushed when e reaches the maximum.

4 Design

The design of MobiFS is guided by insights in the previous section. As Figure 2 shows, MobiFS consists of five major components: (1) the *page cache*, which stores file data in memory; (2) the *write log*, which maintains a write history; (3) the *transactions*, which group entries in the write log and protects them from inconsistency due to overwriting/reordering; (4) the *checkpointer*, which is based on an underlying flash store to persist transactions atomically; (5) the *policy engine*, which marks transaction boundaries, detects user interactions, and decides the timing and target transactions to checkpoint.

MobiFS is designed to work with the existing page cache shared with the OS. For each `write` to the page cache, MobiFS first updates the write log, by appending a new entry or updating an existing entry with the target page address. We also maintain a *page reverse-mapping* from the dirty page back to the write log. Based on the write log and page reverse-mapping, MobiFS establishes atomic transactions. A transaction defines the scope of overwriting and reordering. Finally, the policy engine guides the checkpointer to save transactions without interfering with user interactions. It makes app-specific decisions, according to each app's behavior statistics and

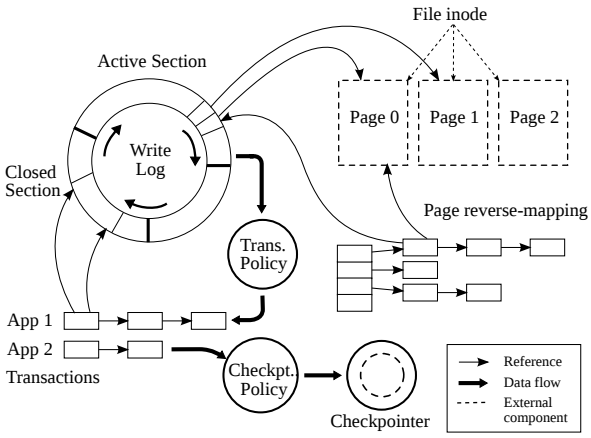


Figure 2: Architecture of MobiFS

current device states.

In a typical set-up, apps have their own write log and transactions, and they share the page reverse-mapping, the policy engine and the checkpointer process(es).

4.1 Write Log

The write log is a chronological record of all writes from an app. The write log is divided into sections. New entries are inserted to the *active* section at the end of the write log, and the *closed* section contains entries that are ready for checkpointing.

The scope of a write log covers all directories accessible by an Android app (`/data/data/[PackageName]`, etc). Note that SQLite is an embedded database for individual apps, whose files are also covered by write logs.

One observation that enables app-specific optimization is the relatively static and isolated app data paths, which avoids the consistency issue of cross-app coordination. However, there might be cases where several apps have access to the same data file, e.g., a galley app and a file management app can manipulate the same set of pictures. In such a case, apps are responsible for handling situations where files are manipulated by a third party. In fact, this is the expectation of mobile operating systems such as Android.

4.2 Versioned Cache Transaction

The write log can have different versions of a cached page. *Versioned Cache Transaction* (VCT) captures this information. A VCT goes through three states in its lifetime. When it is *open*, it accepts new entries in the active section. These entries reference the latest version of a cached page. When *closed*, all entries in the active section are moved into the closed section and protected from further modifications. A VCT in the closed section can not be re-opened. When it is *committed*, all pages referenced by entries in a closed section are

flushed atomically. After the commit, the VCT and its entries are evicted from the write log.

Overwriting and reordering are only allowed within a single VCT. As the checkpointer will guarantee the durability and atomicity of a committed VCT, such optimizations will not leave any inconsistent state in flash.

When a `write` comes, MobiFS has to handle three situations: (1) If the reverse-mapping of the target page does not exist, this means the app writes to a page that is not in the write log. MobiFS appends a new entry, and associates it with a new reverse-mapping. (2) If the reverse-mapping exists and points to the closed section, this means the app writes on a page within a protected VCT. MobiFS copy-on-writes over the target page. A new entry is appended for the modified copy. (3) If the reverse-mapping exists and points to the active section, this means the target page is not in a closed VCT and can be overwritten directly.

4.3 Crash Recovery

VCT boundaries do not necessarily coincide with `fsync`. In a crash, MobiFS relies on the underlying flash management component to recover any partially checkpointed VCT. Take our Ext4 variant for example. It either rolls back to the last transaction, or replays the journal to persist the latest one. In this design, apps and databases always see the data state corresponding to a point of time in history. This is true even after recovering from a system crash. We note that MobiFS guarantees consistency, but not the typical definition of durability.

4.4 Policy Engine

Two categories of policies are running in the policy engine: the *transactioning policy* and the *checkpointing policy*. The former addresses when to close a VCT, while the latter addresses when to save which VCTs into flash. Our general rule is to do checkpointing during the idle time (e.g., when a user is reading the screen content).

The concrete transactioning and checkpointing algorithms we implement in MobiFS are described in Sections 5.2, 5.3 and 5.4, respectively. However, note that our policy engine is an extensible framework, so alternative algorithms may be used.

4.5 Checkpointer

The checkpointer has two responsibilities. First, it invokes an underlying flash component to save data in flash. Second, as MobiFS is loaded, the checkpointer checks a target partition and attempts recovery of any inconsistent data. The flash management component of many filesystems like Ext4 and Btrfs can be easily adopted to implement the checkpointer. The checkpointer exposes four interfaces:

- BEGIN_TRANSACTION, invoked at the beginning of a VCT commission.
- APPEND_ENTRY, invoked for each entry in the target VCT after a successful invocation of the above.
- END_TRANSACTION, invoked at the end of a VCT commission after all its entries are appended.
- WAIT_SYNC, used if flushing is asynchronous.

The underlying flash management component should guarantee the durability and atomicity of the data written between BEGIN_TRANSACTION and END_TRANSACTION.

5 Policy

In this section, we describe our policy design and specific algorithms employed in MobiFS.

5.1 Overview

The policy design has to balance several contradictory requirements of mobile systems: data staleness, energy efficiency, and app responsiveness. We organize their relations into a modular extensible policy framework.

The policy framework assembles three modules. (1) Individual transactions are made ready for checkpointing according to the e curve, in favor of energy efficiency (Section 5.2). This does not rely on any unrealistic assumption of user operation distribution. Instead, we use a second module to predict dynamic user behaviors, so that (2) Transactions may get delayed and queued before checkpointing, in favor of app responsiveness. (Section 5.3). (3) Coordination of multiple apps is managed by a scheduling model (Section 5.4).

We make energy- and responsiveness-optimizing decisions independent, avoiding complex multi-objective optimization with simplistic assumptions. This keeps the algorithms concise as well as effective for practical systems. Many heuristics used in this section are derived from substantial first-hand experience.

5.2 Transactioning Algorithm

Increasing data staleness improves the chance of data overwriting (thus, energy saving), but it pays the price of a higher data loss risk. Hence we face the question: To what extent should MobiFS trade off data staleness for energy efficiency? MobiFS decides by evaluating the energy saving *per* data staleness unit, namely the e ratio (Section 3). Intuitively, the peak of the e curve is the best tradeoff point, as it maximizes the energy saving. Different from related efforts that set a fixed large staleness threshold [32, 35, 43], our philosophy is to reduce data loss risk unless there is a reason (improving energy efficiency) to do otherwise.

The goal of the tradeoff point location (TPL) algorithm is to determine the log entry that marks the end of the current VCT. Each `write` increments the data staleness value, which corresponds to a point in the e curve. Whenever a VCT is closed, the new curve starts at $e = 0$

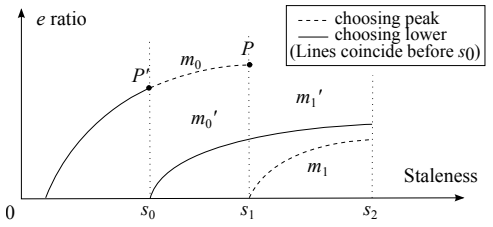


Figure 3: The e curves produced by choosing different transactioning points.

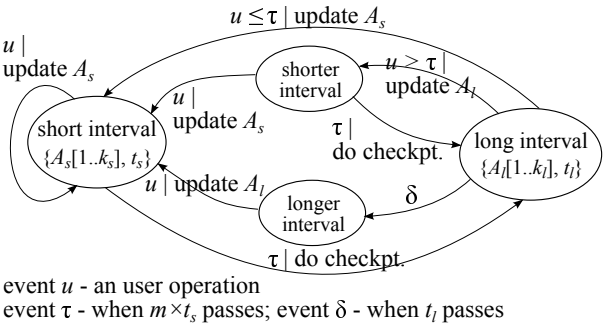


Figure 4: Finite-state machine for interval prediction

(c.f. Figure 3). Ideally, the best point to close a VCT is the peak point with highest e, as explained next. Suppose, for contradiction, that an algorithm decides to close a VCT at P' with $x = s_0$, rather than at the peak P which is $x = s_1$. Then, we can improve this algorithm by shifting the closing point to P, while keeping the subsequent closing point $x = s_2$ the same as with the supposed algorithm. We can show that by doing so, we increase the *probability* of data overwriting[3]. Without loss of generality, we let $s_0 < s_1$. The amount of overwritten data during $[s_0, s_1]$ ($[s_1, s_2]$) is m_0 (m_1) for our strategy; it is m_0' (m_1') for the supposed algorithm. Then we have $m_0 - m_0' > m_1' - m_1$ for the following reasons: $[s_0, s_1]$ still sees our curve quickly rising, so there should be much data to overwrite, yet if it is cut by the supposed algorithm, much data loses the chance to overwrite. In contrast, as P is the peak, the original curve would go down in $[s_1, s_2]$, meaning that little overwritten data is found in there. A simple transformation of the above formula leads to $m_0 + m_1 > m_0' + m_1'$. Therefore, by cutting at P which is necessary to confine data staleness, our strategy has less a chance to overwrite data.

In practice, we have to deal with additional challenges. To mitigate fluctuations in the curve that may lead the algorithm to a locally optimal point, we use linear fitting within a sliding window. The algorithm remembers the latest k points, fits a line, and judges the peak via the gradient of the line. We choose linear fitting, instead of higher order curve fitting, because the algorithm runs on every write so that its complexity should not impose high CPU overhead. Meanwhile, we set a staleness (or time) limit to prevent the opposite – unbounded waiting for a peak. Evaluation of this algorithm is in Section 7.5.

5.3 Interval Prediction

The goal of this algorithm is to predict the length of an interval within which the user is expected not to actively operate the smartphone. These are idle intervals when flushing should be scheduled. The algorithm is triggered when there are pending VCTs. To evaluate the effectiveness of such an algorithm, we call an user operation unexpectedly occurring within a predicted idle interval a *responsive conflict* (RC). Note that idle interval prediction errors can cause several RCs.

We use a state-machine-based prediction method that well balances the low conflict number (i.e., low potential impact on responsiveness) and the long predicted interval (i.e., large potential energy saving as more VCTs can be merged and flushed once). Our algorithm is based on the observation that users usually switch back and forth between short and long intervals, e.g., when reading News Feeds on Facebook, the user may quickly skim some posts before spending time to read one. It is not the goal of this paper to compose a full-fledged interaction model for app users. Instead, we establish the policy framework, and show that, for our purpose, a simple state-machine model is sufficient to learn user patterns and achieve good prediction qualities (evaluated in Section 7.4).

Figure 4 depicts our finite-state machine model. There are two central states: the `short interval` (`long interval`) state when the user operates with short (long) intervals. Each of the two states maintains a recent history of intervals $A[1 \ldots k]$, and uses the minimal value t as a prediction of the next interval. Each of them also has a timer, which is set for a corresponding timeout event whenever necessary. Subscripts "s" and "l" denote the two central states, respectively. Meanwhile, the other two intermediate states, `shorter interval` and `longer interval`, help to decrease or increase interval predictions.

Intuitively, the state machine works as follows. While staying in the `short interval` state, it will loop if the coming event is a user operation. However, if the user operation does not come before a timeout event τ (Figure 4), the machine assumes that the user may begin a long interval, so it changes to the `long interval` state. Afterwards, if the predicted time t_l successfully passes without user operation (event δ), the state machine enters the `longer interval` state which waits until a user operation happens. Otherwise in the `long interval` state, if a user operation comes later than τ, we assume the user

[3]This is not a rigorous mathematical proof. Counterexamples may exist, but overall it is sufficient for the policy design.

still operates with long intervals but the interval prediction should be decreased, so it goes into the `shorter interval` state; if the user operation comes so quickly (before τ) that we guess the user switches to short intervals, the state is directly set to `short interval`.

5.4 Transaction Scheduling

The scheduling problem arises when a user interacts with multiple (background) apps or switches between apps. A typical scenario can be playing a game while listening to radio, and a background service repeatedly checks emails. MobiFS may have multiple write logs with closed VCTs for commission. The scheduler needs to prioritize VCTs to checkpoint, balancing the goals of fairness, high responsiveness and energy efficiency. Our algorithm considers three factors in the decision: (1) Transaction length, or the number of pages to checkpoint. We judge whether the transaction can fit into the predicted interval. (2) Transaction affinity. Transactions from the same app have affinity, because they can be merged if checkpointed together, thereby often saving extra energy. (3) Transaction age, the number of intervals a VCT has previously been skipped by the scheduler.

We use a priority-based scheduling algorithm, with four rules ordered in descending precedence. The algorithm maintains three queues as a way to batch VCTs of the same age. These queues have varying priority in getting flushed. Whenever a VCT is selected to be scheduled, other VCTs of the same app are prioritized. For simplicity of discussion, we may directly use apps as the unit of scheduling thereafter.

Rule 1 (transaction affinity): Whenever the scheduling algorithm skips an app in a queue, the app is moved to a higher-priority queue (if there is one).

Rule 2 (transaction age): Apps are first enqueued in the lowest-priority queue, and promoted to higher-priority queues as time goes on (as described in Rule 1). When there is no feasible choice in all queues, we find a shortest VCT in the highest urgent queue to checkpoint.

Rule 3 (transaction length): An app in the candidate queue is feasible to checkpoint only when its first VCT's length is shorter than the available predicted interval.

Rule 4 (queue replenishment): If an app is unable to checkpoint all its VCTs within a scheduled time, VCTs left are moved to the lowest-priority queue.

6 Implementation

We implement a fully-working prototype in Android 4.1 (Linux 3.0.31), and integrate it with both Ext4 [33] and Btrfs [45]. The code base has 1,996 lines of C code, excluding the reused components from Ext4 or Btrfs. MobiFS does not need kernel recompilation for deployment.

6.1 Main Components

Write Log. We implement the write log with a circular array, as it well supports the required sorting operation. To save space, some logical entry fields (e.g., the page index and version number) are compacted to a single physical data type. The write log also embeds a `kobject` structure, such that MobiFS can export user-space interfaces under the `/sys` directory for easy configuration. Moreover, the log supports certain parallelism in operations by distinguishing protection for checkpointing and appending – the tail of the circular log is protected by a spinlock, and the head is protected by a mutex that only postpones `write`s when the tail grows to reach the head. Finally, to locate which log covers a certain file, we record the log index into the `i_private` field of the `inode` structure. When a new file is created, its `i_private` is derived from its parent directory.

Page Reverse-Mapping. One approach to implement the page reverse-mapping is adding a reference pointer to the `page` structure. Since `struct page` is already packed (e.g., 24+ flags reside in a 32-bit variable), this approach requires enlarging the structure size. Instead, we opt for a customized hash table, which uses a reader-writer lock for each bucket instead of a table-wide lock. Pages associated with entries have its `_count` field incremented so that the Linux Page cache will not evict them. This also means MobiFS must unpin pages before memory space runs low, to avoid out-of-memory problems.

Checkpointer. The checkpointer is a kernel thread, which sleeps if there is no VCT to checkpoint. Upon being woken up by the policy engine, it first runs the interval prediction over the recorded user interaction history. Then, it finds the appropriate VCTs to checkpoint, according to the VCT scheduling policy.

Policy Engine. The implementation of the policy engine needs to consider some of the kernel limitations. For example, the kernel does not directly support floating points due to FPU register overheads. Hence we have to multiply e by 10^3 in our OPL algorithm to preserve thousandth precision.

User Interaction Logger. We record screen events in a queue. An issue is that some single logical user operations, such as dragging, incurs multiple events with small intervals (< 0.01 ms). Therefore, we need a filter to combine these events to one logic operation.

6.2 Integration with Storage Components

Our prototype bases its flash I/O implementation on some existing filesystem components. To support durable and atomic transactions, there are mainly two methods, the write-ahead logging (WAL) and copy-on-write (COW). Ext4 and Btrfs are two typical filesystems that use the methods, respectively.

Ext4. Ext4 uses WAL to achieve durable and atomic transactions. All file writes are first performed in a journaling area, and then moved to the main flash data set. The integration with Ext4 needs to consider the write-twice nature of Ext4 journal, where all data is written on flash twice. This may diminish the gain from MobiFS' overwriting. Fortunately, empirical results suggest that MobiFS can still achieve significant energy savings.

Btrfs. Btrfs relies on COW to achieve durable and atomic transactions. Similar to WAL, COW does not directly update the target area on flash, but makes a new copy of the data for modification. While Btrfs is highly anticipated, it is still in an experimental phase. Therefore, our MobiFS integration with Btrfs (Btr-MobiFS) is not as mature as with Ext4.

7 Evaluation

We evaluate MobiFS by three main metrics - app/user adaptability (Section 7.3), app responsiveness (Section 7.4), and energy consumption (Section 7.5). Before discussing benefits, we estimate memory footprints of MobiFS for running individual apps (Section 7.2).

7.1 Methodology

The evaluation results consist of both trace-driven simulations and actual device measurement; both benchmarks and real apps. We use a Samsung Galaxy Premier I9260 smartphone (with dual-core 1.5 GHz CPU, 1 GB RAM, Android 4.1), and two Kingston microSD cards (with the default 128 MB journal and 4 KB block size). A Monsoon Power Monitor [3] measures device energy consumption. By default, MobiFS refers to our Ext4-based implementation, and Ext4 uses the default ordered mode on Android, journaling only metadata.

Simulation traces are collected from five users operating each of the following top apps (logged in with their own accounts) for five minutes: Facebook (FB), Pandora (PA), Angry Birds (AB), Netflix (NF), Twitter (TT), Google Maps (GM), Citrix Receiver (CR), Flipboard (FL), Web Browser (WB), and WeChat (WC). Traces include I/O system calls, page cache accesses and screen touch timestamps.

Benchmarks consist of the following: (1) AnTuTu's I/O and database benchmarks, (2) RL Benchmark for SQLite (with 13 workloads), and (3) MobiBench for simulating I/O characteristics of Android system. We also use an in-house benchmark that issues sequential `writes` of 8 MB data to the same region of a file 16 times, and invokes `fsync` once every two `writes`.

For experiments that monkey real apps, we choose Browser, Facebook and Twitter, because they are representative of three typical I/O characteristics: Browser incurs few `fsyncs` and is mainly influenced by Ext4 flushing; Twitter is the opposite, triggering more than 50 `fsyncs` per second; Facebook has a moderate number of `fsyncs`. We use monkeyrunner [2] to replay programmed user interaction paths. We test Browser with an in-lab Apache2 web server via 802.11n Wi-Fi to minimize noise introduced by network dynamics.

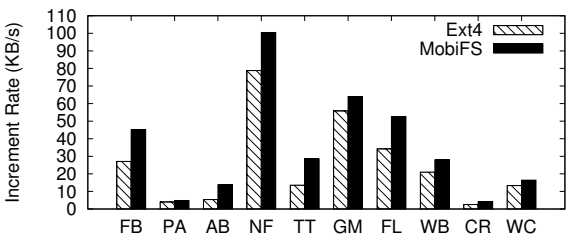


Figure 5: Memory footprint increment rates of Ext4 and MobiFS for different apps.

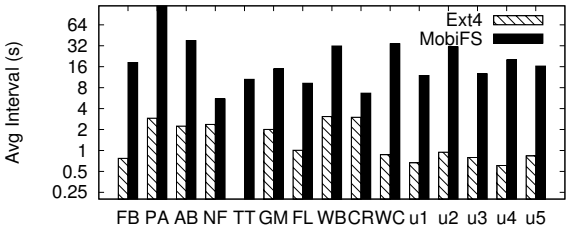


Figure 6: Adaptive checkpoint intervals of MobiFS for different apps/users. Right most user statistics (ux) are collected on Facebook.

7.2 Memory Footprint

We estimate the worst-case footprints of MobiFS according to our app I/O traces, as shown in Figure 5. It is assumed that MobiFS does not checkpoint VCTs. The y axis is the increment rate of the average memory footprint[4] introduced by the filesystem. On average, Ext4 footprints increase by 25.6 KB/s without restriction, while MobiFS incurs an increment of 35.8 KB/s. In other words, having an extra 100 MB memory, MobiFS can support an app running 17.4 minutes without flushing in the worst case (with 100.5 KB/s increment rate). Note that, when the footprint is beyond a threshold, MobiFS can deliberately execute checkpointing to release RAM space. Overall, considering that RAM is ample for apps nowadays, the footprint of MobiFS is acceptable.

7.3 App/User Adaptability

This section evaluates MobiFS' adaptability to both apps and users, as implemented by our tradeoff point location algorithm (Section 5.2).

[4]To reflect different shapes of memory footprint curves, we use integration to calculate the average. For the target increment rate α and the known integral I of the footprint curve over the time interval Δt, we suppose $\frac{1}{2}\alpha\Delta t^2 = I$, so $\alpha = 2I/\Delta t^2$.

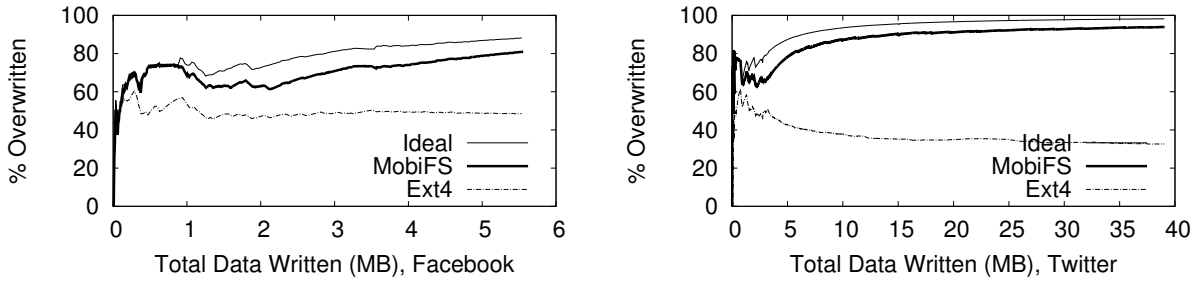


Figure 7: *e* curves against overall app written data.

Adaptability to Apps and Users. We run the tradeoff point location algorithm on the I/O traces and suppose immediate checkpointing without considering user interactions. Figure 6 shows the average of these calculated checkpoint intervals for each app. The average MobiFS checkpoint intervals fluctuate drastically, with a variance of 21.7×. Meanwhile, the geometric mean of MobiFS' average intervals is 17.5 times that of Ext4 flushes. We can see that not only does MobiFS largely extend the traditional Ext4 flush intervals, but also it is inherently adaptive to various apps.

Figure 6 also shows the average of checkpoint intervals for Facebook, as grouped by users. There is up to 2.6× variation of intervals among users for the same app. Such user-oriented adaptability is due to users exploring different contents, from different sources, and with different reading speeds.

Gains from Adaptability. Assuming no flushing should happen, the resulting *e* curve ("ideal") would present the highest potential for overwriting data. MobiFS tries to follow this ideal curve by adapting to individual apps and users. In contrast, Ext4 is limited by fixed flush intervals and traditional `fsync`s. To illustrate MobiFS' gains from adaptability, Figure 7 compares a variant of the *e* curve with Ext4. The *e* ratio here is calculated against the overall data staleness *s* from the beginning, instead of from the last checkpoint. The observation is that MobiFS follows the ideal curve quite closely, and this higher overwrite ratio translates to energy efficiency improvement.

7.4 App Responsiveness

There are two factors that MobiFS focuses on to improve app responsiveness: minimizing responsiveness conflicts, and improving the I/O throughput.

Responsiveness Conflicts. When a user-idle interval is predicted by our interval prediction algorithm (Section 5.3), MobiFS would try to schedule a checkpointing operation to take up the full length of the interval. RCs occur when one or more unexpected user operations happen during such supposedly idle interval. We use two metrics to evaluate the prediction quality:

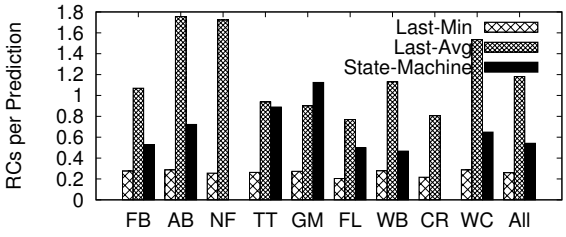


Figure 8: Responsive conflict ratios of three models.

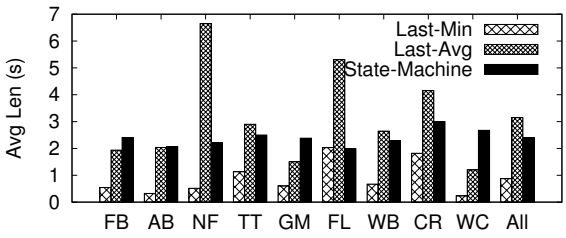


Figure 9: Average length of predicted intervals of three models.

(1) Average number of RCs per prediction; (2) Average predicted interval length. Longer intervals offer more opportunities for transaction merging which saves time/energy.

Figures 8 and 9 separately illustrate the performance of MobiFS' state-machine-based solution on the two metrics, according to our user operation traces. It is compared with the commonly used last min model (LMM)/last average model (LAM), which predicts using the min/average of the last *k* measured intervals. As LMM always takes a conservative prediction, it inflicts only 0.26 RCs per prediction, smaller than LAM which inflicts 1.18. Naturally, our state machine algorithm cannot outperform LMM on this metric, but it achieves 54.8% less than LAM. On the other hand, LAM predicts much longer intervals than LMM. On average, our state machine achieves 75.9% length of LAM, and is over 2.7× that of LMM. As it can remember the previous "long" intervals for prediction, our state machine

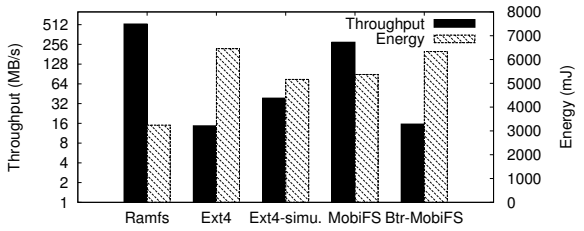


Figure 10: A baseline benchmark on different file-systems. Ext4 only journals metadata. Ext4-simu. is an Ext4 that simulates behaviors of MobiFS, i.e., it journals all data and only does so when MobiFS flushes.

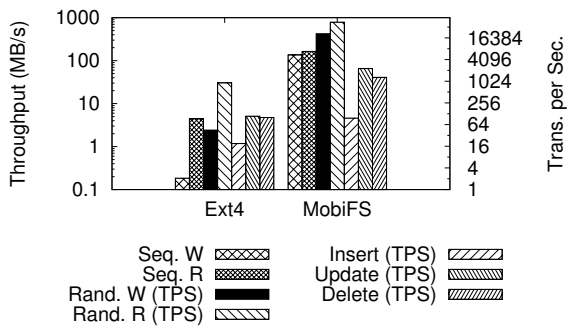


Figure 11: Itemized performance of MobiBench on Ext4 and MobiFS.

even outperforms LAM in 3 out of 9 apps. To sum up, MobiFS has a low RC numbers close to LMM, and realizes long interval length as LAM. It achieves the sweet spot between LMM and LAM.

I/O Throughput. As I/O largely affects app responsiveness, we evaluate typical I/O performance metrics of MobiFS on the device. Table 1 and Figure 11 show results from AnTuTu (ATT), RL and MobiBench. Results suggest that MobiFS can outperform Ext4 by up to 480× (e.g., random writes), and by one order of magnitude in typical database operations.

User-Perceived Latency We evaluate app responsive latency by the time required for monkeyrunner to finish a predefined user interaction path on the device. This method has advantages in (1) eliminating diversity in real user operations that are not mutually comparable and (2) reflecting user-perceived latency that excludes users' reaction time. As Figure 12 shows, monkeyrunner operates the browser to visit 50 websites, and the page loading time drops by 49.0% (-0.36 s/op) when switching from Ext4 to MobiFS. For Facebook, MobiFS reduces the time required to load the news feed five times by 53.6% (-0.85 s/op). Finally, for twitter, the time for loading #Discover tag ten times is reduced by 51.9% (-0.47 s/op). Overall, MobiFS significantly reduces the user-perceived latency of real apps.

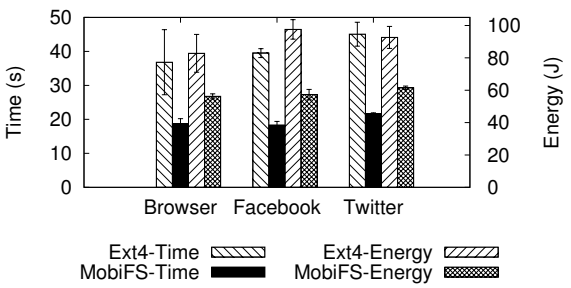


Figure 12: Responsiveness and energy consumption of apps on Android Ext4 and MobiFS.

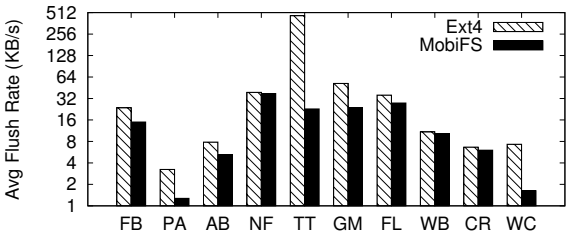


Figure 13: Flushed data of various apps on Ext4 and MobiFS.

7.5 Energy Consumption

MobiFS reduces energy consumption primarily by reducing the amount of data flushed to flash. This section first uses trace-driven simulation to quantify this reduction, and then evaluates the real device energy saving.

Reduction in Flushed Data. Figure 13 compares the amount of data flushed to the permanent storage media in the case of Ext4 and MobiFS, according to the traces. The flush data saving varies among apps, which depends on the number of overlapping writes that an app issues. By the geometric mean of all apps, 53.0% less data were flushed in the case of MobiFS, as compared to Ext4.

Our evaluation shows that MobiFS requires 66.4% more energy than regular Ext4 for the same flush size due to write-twice (Section 6.2), so the overall simulated energy cost of MobiFS is 78.3% of Ext4. Meanwhile, if we calculate average flush sizes, Ext4 flushes 4.29× the amount of data that MobiFS flushes, which means that MobiFS consumes 61.2% less energy than Ext4.

Device Energy Saving. We first consider the baseline energy figures in Figure 10. MobiFS logically flushes only half the amount of data compared to Ext4, but due to write-twice (Section 6.2) it should incur similar energy consumption with Ext4. In practice, however, both Ext4-simu. and MobiFS require less energy (over 16.8%), partially because internal data movement incurs less CPU processing than independent write system calls. On the other hand, although it does not write twice, Btr-MobiFS costs similar energy with Ext4, due to COW overheads.

Item		Ext4	MobiFS	Improve.
Perf.	ATT(score)	689.9±21.5	1817±51.0	+163%
	RL(sec.)	38.6±0.4	19.1±0.4	-50.1%
Energy* (J)	ATT	24.3±0.8	20.3±0.7	-16.4%
	RL	43.8±0.5	37.6±0.6	-14.2%

Table 1: Performance and energy of AnTuTu (ATT) and RL Benchmark on Android Ext4 and MobiFS. ATT scores favor the higher; RL time favors the lower.

Moreover, by comparing MobiFS with Ext4-simu., we can see that the energy cost of the unique components of MobiFS only counts 4.0%.

Results from benchmark tools on the device also justify MobiFS's contribution in reducing energy consumption. Table 1 shows energy savings under the AnTuTu and RL benchmarks, as compared to Ext4. Note that these workloads hardly manifest MobiFS' full potential, because our design is highly oriented to real app/user behaviors, as evaluated in the following experiment.

Figure 12 compares the energy cost of real apps in the case of MobiFS and regular Ext4. Specifically, the energy cost of the whole device drops on average by 32.1%, 41.3% and 33.6% with Browser, Facebook and Twitter, respectively. We can see that MobiFS substantially improves the energy efficiency of mobile apps.

8 Related Work

Latest Mobile Filesystems. F2FS [26] (on Moto X) observes 128% higher random write throughput than Ext4 [24]. DFS [17] improves the I/O performance by delegating storage management to the flash hardware. In contrast, our memory-centric solution can achieve nearly two orders of magnitude of improvements on read/write performance (Figure 11).

Revisiting `fsync`. MobiFS decouples the consistency and durability functionalities of `fsync`. The same methodology has been exploited to different extents. xsyncfs [40] stalls any user-visible output until the durability is accomplished. We have a more aggressive tradeoff for performance than xsyncfs, considering the unique features of mobile systems. OptFS [6] introduces `osync` and `dsync`. The former ensures only *eventual* durability. In a sense, we also follow this durability model. However, OptFS' mechanism ensures consistency of journaling disk writes by checksums, while we realize consistency in the page cache. It does not study policy design for mobile systems. Other similar work [32, 35, 43] simply uses a static time bound on staleness, and does not adaptively tradeoff in the same way as MobiFS does for mobile apps.

Memory Data Management. Main-memory databases [10, 12, 18, 41], adaptive logging [23], recoverable virtual memory [46], flash-oriented [9, 21, 31] and NVM-based [8, 14, 27, 49] storage systems optimize the performance of data flushing/writeback. NVM-based swapping [51] shows less performance improvement than our design. qNVRAM [30] implements a persistent page cache but requires new APIs to use. External journaling [16] requires extra storage devices, and does not optimize energy efficiency. Fjord [20] distinguishes apps mainly by cloud-related properties, and changes software configuration accordingly. Host-side flash caching [25] preforms a similar tradeoff between performance and staleness. Beyond all the above work, we advance at identifying minimal modifications to `fsync` and the page cache in a constrained mobile system, a systematic study of key tradeoffs, and a policy design with app/user-adaptive optimization.

Energy/Responsiveness Optimization. BlueFS [39] carefully chooses the least costly replica among multiple nodes. SmartStorage [38] sacrifices 4%-6% performance for energy efficiency by tuning storage parameters, while we achieve orders of magnitude of performance promotion along with energy saving. Capsule [34] only considers random or sequential access patterns. SmartIO [36] focuses on prioritizing reads over writes. Mobius [7] takes into account node location, network congestion, etc. While increasing I/O burstiness for energy efficiency [42, 48] shares a similar logic with us, we also consider adaptive strategies and asynchronous `fsync`. Simba [13] crafts a sync interface for both local and cloud data. Similar to Simba, we also provide consistency cross filesystem and database for local data. After all, our observation on the *e* curves and resulting multi-objective policy designs distinguish MobiFS from these above optimization works.

9 Conclusion

MobiFS identifies a fundamentally new sweet spot in the staleness-performance and staleness-energy tradeoffs that lie at the core of a filesystem for smartphones. Its new memory-centric rationale, along with app/user-adaptive incremental checkpointing, and the VCTs to support asynchronous `fsync`, provides a good reference for next-generation data storage design tailored for the mobile environment. Evaluations via user traces, micro-benchmarks, and real apps on the real device illustrate the sound policy design and practical benefits.

Acknowledgement

We thank our shepherd, Chia-Lin Yang, and the anonymous reviewers for their valuable feedback. This work is partially supported by National High-Tech R&D (863) Program of China (2012AA012600), National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), and Chinese Special Project of Science and Technology (2013zx01039-002-002).

References

[1] HTTP archive trends. `http://httparchive.org/trends.php`, 2014.

[2] The monkeyrunner tool. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`, 2015.

[3] Monsoon power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`, 2015.

[4] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012).

[5] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *USENIX ATC* (2010).

[6] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *SOSP* (2013).

[7] CHUN, B.-G., CURINO, C., SEARS, R., SHRAER, A., MADDEN, S., AND RAMAKRISHNAN, R. Mobius: unified messaging and data serving for mobile apps. In *MobiSys* (2012).

[8] COBURN, J., BUNKER, T., SCHWARZ, M., GUPTA, R., AND SWANSON, S. From aries to mars: Transaction support for next-generation, solid-state drives. In *SOSP* (2013).

[9] DAI, H., NEUFELD, M., AND HAN, R. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys* (2004).

[10] DEBRABANT, J., PAVLO, A., TU, S., STONEBRAKER, M., AND ZDONIK, S. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow. 6*, 14 (Sept. 2013).

[11] DESNOYERS, P. What systems researchers need to know about nand flash. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems* (2013), HotStorage '13.

[12] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *SIGMOD* (1984).

[13] GO, Y., AGRAWAL, N., ARANYA, A., AND UNGUREANU, C. Reliable, consistent, and efficient data sync for mobile apps. In *FAST* (2015).

[14] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994).

[15] IEEE AND THE OPEN GROUP. IEEE Std 1003.1-2008 (POSIX.1-2008). `http://pubs.opengroup.org/onlinepubs/9699919799/`, 2015.

[16] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX ATC* (2013), pp. 309–320.

[17] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. Dfs: A file system for virtualized flash storage. In *FAST* (2010).

[18] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow. 1*, 2 (Aug. 2008).

[19] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *FAST* (2012).

[20] KIM, H., AND RAMACHANDRAN, U. Fjord: Informed storage management for smartphones. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (2013).

[21] KIM, H., RYU, M., AND RAMACHANDRAN, U. What is a good buffer cache replacement scheme for mobile flash storage? In *SIGMETRICS* (2012).

[22] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android I/O: Multi-version b-tree with lazy split. In *FAST* (2014).

[23] KIM, Y.-S., JIN, H., AND WOO, K.-G. Adaptive logging for mobile device. *Proc. VLDB Endow. 3*, 1-2 (2010).

[24] KLUG, B. Moto x review. `http://www.anandtech.com/show/7235/moto-x-review/9`, 2013.

[25] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *FAST* (2013).

[26] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *FAST* (2015).

[27] LEE, E., KANG, H., BAHN, H., AND SHIN, K. Eliminating periodic flush overhead of file I/O with non-volatilebuffer cache. *IEEE Transactions on Computers*, 99 (2014).

[28] LEE, K., AND WON, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *The ACM SIGBED International Conference on Embedded Software* (2012), EMSOFT '12.

[29] LI, J., BADAM, A., CHANDRA, R., SWANSON, S., WORTHINGTON, B., AND ZHANG, Q. On the energy overhead of mobile storage systems. In *FAST* (2014).

[30] LUO, H., TIAN, L., AND JIANG, H. qnvram: quasi non-volatile ram for low overhead persistency enforcement in smartphones. In *6th USENIX Workshop on Hot Topics in Storage and File Systems* (2014), HotStorage '14.

[31] LV, Y., CUI, B., HE, B., AND CHEN, X. Operation-aware buffer management in flash-based systems. In *SIGMOD* (2011).

[32] MA, D., FENG, J., AND LI, G. Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *SIGMOD* (2011).

[33] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007).

[34] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys* (2006).

[35] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., NAREDDY, K., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., AND KHAN, O. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *NSDI* (2014).

[36] NGUYEN, D. T. Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (2014), UbiComp '14 Adjunct, ACM.

[37] NGUYEN, D. T., PENG, G., GRAHAM, D., AND ZHOU, G. Smartphone application launch with smarter scheduling. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication* (2014), UbiComp '14 Adjunct.

[38] NGUYEN, D. T., ZHOU, G., QI, X., PENG, G., ZHAO, J., NGUYEN, T., AND LE, D. Storage-aware smartphone energy savings. In *UbiComp* (2013).

[39] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the blue file system. In *OSDI* (2004).

[40] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *OSDI* (2006).

[41] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011).

[42] PAPATHANASIOU, A., AND SCOTT, M. Energy efficiency through burstiness. In *Fifth IEEE Workshop on Mobile Computing Systems and Applications* (2003), HotMobile '03.

[43] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional consistency and automatic management in an application data cache. In *OSDI* (2010).

[44] RECOVERY SPECIALTIES, LLC. Data consistency: Explained. `http://recoveryspecialties.com/dc01.html`, 2015.

[45] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The linux b-tree filesystem. *ACM Trans. Storage (TOS) 9*, 3 (2013).

[46] SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. Lightweight recoverable virtual memory. In *SOSP* (1993).

[47] WANG, A.-I., REIHER, P. L., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX ATC* (2002).

[48] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: a novel I/O semantics for energy-aware applications. In *OSDI* (2002).

[49] WU, M., AND ZWAENEPOEL, W. envy: a non-volatile, main memory storage system. In *ASPLOS* (1994).

[50] XU, F., LIU, Y., MOSCIBRODA, T., CHANDRA, R., JIN, L., ZHANG, Y., AND LI, Q. Optimizing background email sync on smartphones. In *MobiSys* (2013).

[51] ZHONG, K., WANG, T., ZHU, X., LONG, L., LIU, D., LIU, W., SHAO, Z., AND SHA, E.-M. Building high-performance smartphones via non-volatile memory: The swap approach. In *The ACM SIGBED International Conference on Embedded Software* (2014), EMSOFT '14.

WearDrive: Fast and Energy-Efficient Storage for Wearables

Jian Huang[†], Anirudh Badam, Ranveer Chandra and Edmund B. Nightingale

[†]Georgia Institute of Technology Microsoft Research

Abstract

Size and weight constraints on wearables limit their battery capacity and restrict them from providing rich functionality. The need for durable and secure storage for personal data further compounds this problem as these features incur energy-intensive operations. This paper presents WearDrive, a fast storage system for wearables based on battery-backed RAM and an efficient means to offload energy intensive tasks to the phone. WearDrive leverages low-power network connectivity available on wearables to trade the phone's battery for the wearable's by performing large and energy-intensive tasks on the phone while performing small and energy-efficient tasks locally using battery-backed RAM. WearDrive improves the performance of wearable applications by up to 8.85x and improves battery life up to 3.69x with negligible impact to the phone's battery life.

1 Introduction

The utility of a mobile device has long depended upon the tension between the device's size, weight and its battery lifetime. Smaller, lighter devices tend to be easier to carry. However, battery lifetime is mainly a function of size. A smaller device must therefore contain a smaller battery making energy a precious resource. The need for durable storage further compounds this problem. Slow flash storage wastes energy by keeping the CPU active for longer period of time [26, 27, 52], yet the use of a battery dictates that durable storage is vital to a device's utility. Likewise, data encryption is energy-intensive [31], but the sensitive nature of personal information that devices collect dictates using appropriate protection mechanism over a durable medium like flash that can be easily detached from a stolen device to retrieve personal data.

On wearables [43, 13, 5, 35], these trade-offs are magnified. Size matters even more since the device is worn on the body, therefore these devices have a very precious energy reserve. A watch that must be charged after a few hours is not very useful. Likewise, these devices generate precious sensor data (e.g., body sensor readings and location) that must be guaranteed against loss and theft.

In this paper, we explore a new approach to storage on wearable devices that does away with local durable storage while leveraging a nearby phone to protect against data loss and theft in an energy efficient manner. The system, called WearDrive, uses only memory on wearables for storage operations to provide performance and energy improvements. It exploits the battery in mobile devices to provide durability for the data in memory. It leverages low-power network connectivity available on wearables to exploit the capabilities of the phone. New data is asynchronously transmitted to the phone, which ultimately performs the energy-intensive operations of storing data with encryption in its local flash.

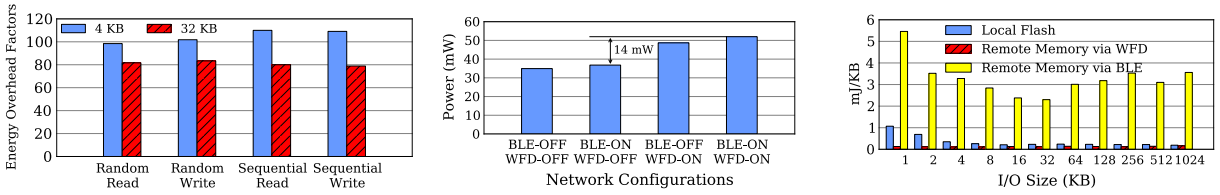
WearDrive targets the two most important application scenarios of wearables. The first scenario is the "extended display" that uses the wearable as a second display to allow applications on a nearby phone to run interactive but less-featured companion applications. Examples include companions that provide notifications for emails, social networks, etc. Providing *fast and durable storage* to such applications helps wearables conserve battery while remaining interactive.

The second scenario is sensor data analysis. Wearables are packed with sensors that take advantage of their location on a person's body. Exposing this data to the applications on the phone with *low-energy data sharing* can open up powerful applications. WearDrive targets these scenarios and reduces the need for a large battery and eliminates the need for flash on wearables. This paper makes the following contributions:

- A distributed battery-backed RAM based storage system called WearDrive is presented that can help applications span data and computation across the wearable and phone quickly and energy efficiently.
- A hybrid Bluetooth and Wi-Fi data transfer scheme is presented. It helps the wearable exploit the capabilities of the phone at a low-energy cost by shipping data and computation to it.
- Data-intensive wearable workloads are identified. A benchmarking tool named WearBench with several data-intensive scenarios is developed to benchmark applications spanning wearable and phone.

Experimental results show WearDrive helps applications obtain up to 8.85x better performance and consume up to 3.69x less energy compared to the state-of-the-art systems with little impact to the phone.

The rest of this paper is organized as follows: Section 2 presents the challenges that we address in this



(a) Software Storage Stack Vs. Storage Hardware (b) Standby Power Consumption (c) Flash Vs. Wireless Network

Figure 1: Motivating scenarios for WearDrive: (a) Mobile storage stacks are energy-intensive because storage software consumes 80–110x more energy than flash. (b) To maintain a connection to the phone for the wearable, WiFi-Direct consumes 10–15mW extra power, while Bluetooth Low-Energy requires only 1–2mW. (c) In terms of energy consumption of the whole system when sequentially writing 32 MB data set with various I/O granularities, it is more energy efficient to write to remote phone's memory via WiFi-Direct than to write data locally to flash on the wearable.

work. Design and implementation of WearDrive are described in Section 3. The benchmarking suite and the evaluation results are shown in Section 4. Section 5 presents the related work. Finally, we describe the conclusions from this work in Section 6.

2 Wearable Storage Challenges

Wearables present a new challenge for mobile system design. Constraints on size and weight limit the battery capacity, but their location on the body and proximity to the phone create new opportunities.

Small Batteries. Li-ion battery metrics like gravimetric energy density (Watt-Hours/kg) and volumetric energy density (Watt-Hours/liter) take 10+ years to double [9]. Therefore, wearables will still be restricted to battery capacities of 1–2 Watt-Hours for the next several years because of their size and weight constraints; today's phones have 7–11 Watt-Hours batteries [19, 44, 33]. Therefore, we propose that *the battery on the phone be traded for the battery on the wearable*.

Energy Overhead of Legacy Platforms. To simplify the hardware and software development of wearables, manufacturers have chosen to reuse the system-on-a-chip (SOC) design and mobile operating systems that were originally made for phones and tablets. For example, most smart-watches and smart-glasses [43, 14, 13, 5] follow this approach to reduce cost, and accelerate development of the platform and the applications. The focus of this paper is on such wearable devices. This means that wearables face a larger energy challenge compared to phones, because of their smaller batteries.

Our prior work [31] identified that mobile storage software consumes up to 110x more energy compared to flash hardware for accessing data as shown in Figure 1(a). The energy overheads are caused by three factors. First, mobile flash is slow and increases CPU idle time while waiting for IO completion [26]. Second, storage on mobile devices is accessed via managed runtime environments like the Darwin engine on Android and the CLR engine on Windows that add additional CPU overhead. Finally, encryption of data that happens using special CPU instructions is also energy intensive. *A fast and energy-efficient storage system with security and privacy guarantees is needed for wearables*.

New Applications. Nearly all existing applications of wearables fall into two categories: extended display and sensor analysis. Using a wearable as an extended display requires arbitrary mobile application state be shared across the wearable and phone. And for wearables, the users focus more on *new content* from contextual applications like email, messaging, social networks, calendar events, music controls, navigation companion and etc.

Wearables are rich sources of sensor data (Table 3) because of their location on the body. For example, watches can better monitor heart-rate and glasses can provide better video sensing. These sensors pave the way for a wide variety of useful applications including long term fitness/wellness tracking, detecting chronic health conditions like sleep-disorder, heart conditions, etc. Existing wearables unfortunately are severely crippled in terms of battery size and provide only limited data analytics. *A storage system capable of supporting these wearable workloads and exploiting their characteristics for performance and energy-savings is needed.*

Reaching the phone. Bluetooth Low Energy (BLE) enables wearables to maintain a constant connection to phone at a low-energy cost (Figure 1(b)). However, its low modulation rate imposes a large energy tax on large data transfers. An alternative is WiFi-Direct (WFD) which requires higher constant power to maintain a connection, but supports low-energy large data transfers with high modulation rates. Figure 1(c) shows the average energy per KB consumed by the whole system of the wearable (see Table 3) as it sequentially writes data to local flash or remotely to the phone via BLE/WFD. The experimental setup is the same as described in Section 4.2. Experimental results indicate that the energy overhead of writing data to remote memory via WFD is comparable to that of writing data to flash on the wearable.

The challenge is to build a mechanism to connect the wearable to the phone with a constant low-power connection overhead with a means to transfer data energy-efficiently. A hybrid connection and data-transfer mechanism can be built using BLE and WFD so that *data sharing between wearable and phone can be enabled at a low-energy cost*.

Slow flash. Mobile flash is slow and energy-intensive [26]. Faster flash technologies like SSDs require 25–100% more $/GB and 5x more energy per operation, and have a controller alone that is bigger than an entire SD card. Moreover, even SSDs are 10,000x slower than DRAM [1]. Furthermore, we demonstrate that data transfers over WiFi-Direct between two mobile devices consumes less energy than writing the same data to flash (Figure 1(c)). We propose that *wearables actively use only DRAM (local and remote) to drastically speed up storage operations*.

3 WearDrive Design

We begin by showing how applications minimize using flash and use mostly DRAM for *fast and durable* storage operations on wearables. We then present a new data management system that helps applications span extended-display and sensor data across the wearable and the phone. A new hybrid BLE/WFD data transfer mechanism is then described which helps WearDrive transmit data at a low-energy cost to the phone.

3.1 Storage with Battery-Backed RAM

To speed up storage operations, WearDrive actively uses DRAM as storage. However, WearDrive guarantees durability in spite of DRAM's volatility. DRAM on mobile platforms is continuously refreshed. The only time when the DRAM refresh stops is when the device is shutdown, the battery runs out of energy or it is removed. The first two scenarios provide an early warning sign allowing data in DRAM to be flushed to flash just in time before the refresh stops. Removing the battery while the system is running can lead to data loss even in today's systems. Moreover, most wearables' batteries are not removable. Therefore, we assume that DRAM can be treated as non-volatile on such devices. We call such DRAM as battery-backed RAM (BB-RAM).

BB-RAM coexists with DRAM to minimize OS changes. It grows and shrinks dynamically according to the memory pressure in the rest of the OS. DRAM is a precious resource on wearable devices. Most of the wearables we surveyed have less than 0.5GB of DRAM. While reserving a known and fixed region of physical memory as BB-RAM simplifies the implementation, it leads to fragmentation of DRAM and does not allow BB-RAM to dynamically expand and contract in accordance with application/OS requirements. WearDrive's BB-RAM design adapts to memory pressure and spans across non-contiguous physical memory pages.

WearDrive uses BB-RAM both on the wearable and phone to ensure high-performance of applications spanning both the wearable and the phone. Wearable uses the phone as the secondary storage for its data. All old data on the wearable's BB-RAM is retired to the phone's BB-RAM. All dirty data in wearable's BB-RAM is also sent to the phone when the wearable needs to shutdown. Likewise, phone uses its flash as the secondary storage for its data in BB-RAM.

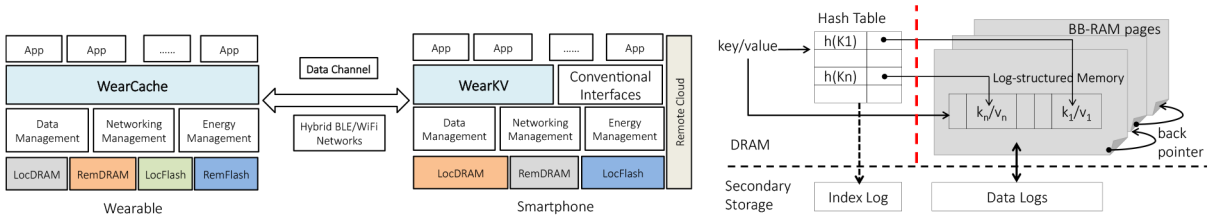
Data in BB-RAM is not lost even after an OS crash. WearDrive uses a firmware component to ensure that BB-RAM is backed to flash in case of an OS crash. Firmware needs additional support to identify the physical pages that are used as BB-RAM. For this purpose, WearDrive reserves a small known region of physical memory to store a bitmap in DRAM to represent whether that physical page belongs to BB-RAM or not. The firmware uses these bits to identify BB-RAM pages after an OS crash (before shutdown) and writes them to a reserved region on flash. This simple design allows BB-RAM to coexist with DRAM and also enables a firmware without any OS state awareness to ensure data durability. Recovering WearDrive's state after a crash solely from the set of BB-RAM pages that it spans across is a harder problem and we present its design in the next sections.

WearDrive uses BB-RAM only as long as there is enough battery life left to ensure durability of data in case of a crash. When battery level reaches a threshold, WearDrive stops using BB-RAM and treats all of DRAM as volatile. New and dirty data is first written to local flash to ensure durability. We set the threshold to 7% in WearDrive based on the observation that flushing 512 MB data from memory to flash sequentially costs about 5% of wearable's battery life on our reference wearable platform. However, this value can be adapted according to the hardware.

Warm reset. WearDrive is optimized for warm resets of the OS. If the available energy is above 7%, the firmware continues to refresh DRAM without scrubbing or cleaning any data. The OS then separates the pages in BB-RAM from regular DRAM using the bitmap and continues the boot process.

OS Deadlock. In case of a deadlock there is a chance that the data in BB-RAM will permanently be lost as the phone is completely drained out of battery. WearDrive uses a watchdog timer to detect if the OS is hung. When the battery life reaches the threshold, firmware schedules a BIOS-context process that wakes up once every sixty seconds and sets a bit in a known portion of memory that it expects the OS to reset every sixty seconds.

[1] Data surveyed from samsung.com, newegg.com and amazon.com



(a) WearDrive Overview

(b) KV-Store Design

Figure 2: (a) WearDrive expands wearable's memory and storage capacity by leveraging phone's capabilities. LocDRAM/RemDRAM represents local/remote DRAM, LocFlash/RemFlash are local/remote Flash. (b) BB-RAM pages are held in a linked list. The pages contain a sequential log of key-value pairs as they arrive. The hashtable stored in regular DRAM contains the index for the key-value store whose state can be efficiently recovered after failures.

If the OS fails to reset it during an iteration then the firmware assumes that the OS has hanged and flushes the data to flash by itself and disables the watchdog timer. The watchdog timer is also disabled as soon as the OS starts using DRAM as volatile.

3.2 Storing Data Across Devices

Since extended display and sensor data analysis scenarios need to span data across wearables and phone, we design WearDrive as a distributed storage system spanning across all devices. We find that in most extended-display scenarios, the wearable is treated as a helper for the full application on the phone because of the smaller screen size on watches, lack of touch screen on glasses and small battery size on both. For this reason, we design the component of WearDrive on the wearable as a cache (WearCache) and the component of WearDrive on the phone (WearKV) as the main storage of data (see Figure 2(a)). WearKV and WearCache both have a key-value store interface that mobile application developers are familiar with. We use the same KV-store system to implement both WearCache and WearKV.

3.3 KV-store Design

KV-store is optimized for BB-RAM. This ensures fast and durable operations for WearCache and WearKV when inserting new data. KV-store prioritizes new data. The focus of wearable applications is on the latest data generated by phone applications and also by the sensors. Examples include the user's interest in latest notifications and most recent sensor values that can provide statistics about a run or a workout session. Therefore, the KV-store is implemented as a sequential log of key-value pairs in BB-RAM with FIFO replacement. Figure 2(b) illustrates the design. Keys and values are arbitrary length data blobs. New values are inserted by appending the KV-pair to the head of the log and adding a hash table entry with pointers to the key and the value in the log.

KV-store stores data in BB-RAM and metadata in DRAM. The log of KV-pairs is stored in BB-RAM and the hash table is stored in regular DRAM. The rationale for this is that the hash table can be recovered from BB-RAM in case of a crash by scanning through the BB-RAM pages in the right order. In case of a clean shutdown, the hash table is serialized to secondary storage (Index Log in Figure 2(b)). This design choice makes effective use of the precious BB-RAM space.

KV-store can recover BB-RAM and DRAM state after a crash. Recall that the firmware flushes BB-RAM pages to local flash in case of a crash. To recover the hash table and the correct head of the log of KV-pairs, ordering of the BB-RAM pages is needed. The ordering of the BB-RAM pages in the log is determined by a four byte pointer stored at the tail of every BB-RAM page to the next BB-RAM page in the log as shown in Figure 2(b). Each KV-pair in BB-RAM is a sequence of five fields: four bytes length of the key followed by the key, followed by eight bytes of application identifier (described later) and then four bytes length of the value followed by the value. This FIFO of BB-RAM pages allows the KV-store to arbitrarily increase its size by appending new pages and decrease the size of the log by purging the KV-pairs at the tail to secondary storage. Moreover, the firmware remains simple, precious BB-RAM space is best utilized and recent data that is of interest for applications is prioritized during page replacement.

WearCache is the KV-store instance that lives on the wearable and caches all the latest data from applications and sensors. New data arrives in WearCache via two methods: when phone applications push data to their companion applications and when sensors generate new values. When WearCache runs out of BB-RAM, it flushes old data to WearKV on the phone in FIFO order as the focus of the wearable is always on new data. It does so by simply moving the tail forward in the log of KV-pairs on BB-RAM several KV-pairs at a time. This provides the functionalities of having recent data on the wearable, adapting to memory pressure, and providing an efficient replacement policy. An example application on today's watches that can leverage this storage model

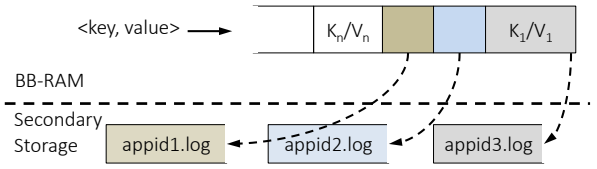


Figure 3: WearDrive creates individual logs per application and per sensor to isolate on secondary storage.

is a notification center for recent emails. The user's focus will be on the most recent emails while the older emails may be safely flushed to WearKV as the user may not access them on the wearable. Complex functionalities are implemented by the email application on the phone while a companion email application on the wearable keeps the design/UI simple with focus on latest data.

WearCache removes flash I/O overhead from the critical path of applications. The OS, application binaries and other application metadata continues to reside on local flash. However, data accessed in critical path resides in WearDrive. The key-value interface to WearDrive eases development as wearable applications already use the key-value interface for sharing data between the phone and wearable [3]. As future work, we wish to provide filesystem and database interfaces using BB-RAM.

WearDrive supports simple sensor data analytics on the wearable and complex data analytics on the phone. Small battery restricts wearables to analyzing sensor logs from short activities like the latest run/workout-session or other short activity. However, applications can perform rigorous analytics on the phone (several days worth of sensor logs at a time). Applications on the phone can proactively pull the sensor data from WearCache as and when a certain number of samples are available. For example, a fitness tracker on the phone can register with WearCache that the heart-rate logs from the wearable be pushed to the phone once every ten minutes. WearCache implements these requests in the following manner. For each sensor, WearCache pre-allocates a KV-pair. A certain amount of space is reserved for the value upfront. The sensor samples (configurable sampling rate) are gradually added to the pre-allocated value as they become available. Data is pushed to the phone and phone-applications are notified accordingly.

WearKV is the KV-store that resides on the phone and contains all the data of the wearable. It contains old extended display data and the entire log of sensor values. Old extended display data is fetched back to WearCache on demand (this is a rare event as wearables focus on new data). The phone with its larger battery can use the full sensor log to perform rigorous sensor data analysis. When WearKV runs out of BB-RAM, it flushes old data to flash where it creates a per-application and per-sensor sequential log as shown in Figure 3. It does so by leveraging the metadata information stored in the values where it records the device-ID, application-ID, sensor-ID and time stamp of creation.

Data in WearDrive crosses the memory/flash boundary only on the phone. Data encryption and other mechanisms put in place to ensure security and privacy of data are needed only for "truly" non-volatile media like flash that can be detached from the rest of the phone and have unprotected data stolen in a straightforward manner. Therefore, the heavy software cost [31] of storage is offloaded to the phone. Note that treating DRAM as non-volatile by using it as BB-RAM is at least as secure as the previous model where data was not encrypted in DRAM as DRAM which is part of the SOC is hard to detach from a device. BB-RAM is a mechanism to ensure that data in DRAM in never lost as opposed to making DRAM "truly" non-volatile.

Offline Capabilities. WearCache can function without the phone. WearCache can lock data on the wearable based on time of arrival such that it is not purged to the phone until explicitly deleted. Offline capabilities allow applications to lock data to be available locally so that functionality can be provided without the phone. An example is when the email companion application imposes a restriction that email from last three days be locked locally. KV-pairs are written to flash on the wearable only if WearCache runs out of BB-RAM and the applications impose an offline availability restriction. Offline requirements are specified in WearCache using time cutoffs per applications and per sensor (see Table 1). We compare the specified time with the timestamp stored in KV-pair's metadata. The qualified offline data is written to its local flash's logs. As time passes, WearCache will move the tail closer to the head on the flash log and overwrites older data that the application does not need.

3.4 Communication

Efficient reachability to the phone allows the wearable to be designed with less DRAM and slower flash thereby reducing their cost. Moreover, it allows the wearable to offload storage and computations to the phone. BLE 4.1 and 802.11a/b/g/n/ac are the network connectivity options for wearables. While a few smart-watches only have BLE, we envision that Wi-Fi will make it to all wearables as it enables efficient large data transfer.

Standalone BLE or WFD is not an ideal network connection. BLE consumes low power (1–3mW) for staying always connected to the phone while using a WFD to stay connected to the phone consumes 5x extra power (10–14mW) (Figure 1(b)). On the other hand, BLE consumes 10–20x extra energy for transmitting data when compared to WFD (Figure 1(c)). A mechanism to minimize the total energy of always staying connected and for transferring data is required.

API	Description
OpenWearDrive (FileName)	open a connection to WearDrive and obtains a handle, the data is represented using an opaque *FileName*.
CloseWearDrive (handle)	close the connection to WearDrive and flush any data from BB-RAM in the process to an appropriate location.
InsertKV (handle, key, value)	insert the new key/value to the *FileName* corresponding to the handle.
ReadKV (handle, key)	provide the value corresponding to the key in the *FileName* file.
MakeOffline (handle, date)	make all data of this file that arrived after a certain date available on the wearable even when the phone is not reachable. Date is specified relatively to the current time. This function is available only to WearCache.
DeleteOldData (handle, date)	provide a hint to WearDrive that data beyond a certain date can be deleted. Date is an absolute value. This function is available only to WearKV.
RegisterForSensor (DeviceID, SensorID)	register an application for values from the sensor represented by *(DeviceID, SensorID)*.
UnregisterFromSensor (DeviceID, SensorID)	unregister the application from a sensor.
RegisterCallBack (TimeGap, CallBackFunction)	make WearDrive issue the *CallBackFunction* in the context of registering application every *TimeGap* seconds with the newly available sensor values.
Compute (DeviceID1, SensorID1, ..., DeviceIDN, SensorIDN, TimeGap)	a function that does not access any global variables but accesses data in sensor logs that are accessible to the application. It can be executed on both wearable and phone.

Table 1: WearDrive API

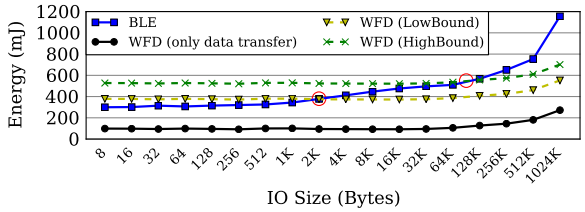


Figure 4: Energy consumption of data transfer via BLE and WFD. WFD is efficient if connection establishment, tail latency and connection-teardown are not included.

Using BLE for staying connected and short data transfers, and turning on WFD solely for large data transfers is a hybrid solution. This is practical because WearCache and WearKV know how much data is to be pushed. If it is beneficial then a control signal over BLE is sent to the other side to turn on WFD. Data transmission begins on BLE and switches over to WFD when it is available.

Knowing the right data transfer size for switching on WFD is crucial. To estimate the transfer size at which it pays-off to turn on the WFD, we conduct the following experiment: transferring data of various sizes on BLE and WFD. We keep BLE always on and send data of various sizes between two mobile devices whose power consumption is monitored using the Monsoon power monitor [36]. We then estimate the energy required for transferring the data via WFD. The energy estimates for WFD contains the energy needed for turning the WiFi chipset on and off. Figure 4 shows the transfer size at which using the hybrid protocol pays off.

The pay-off point for switching to WFD depends on signal quality. We present the results for two extreme modulation rates in 802.11n: the highest modulation rate and the lowest modulation rate. A crossover-point database is built for various modulation rates of BLE and WFD. We use the BLE signal strength to estimate the WiFi signal strength as they use the same band and radio over the same distance.

Picking the right time to turn off WFD is important. WFD consumes more power than BLE in idle state (i.e., standby power gap). However, network discovery, connection and powering-down are expensive, frequently turning WFD on/off would incur more energy usage than keeping it in idle state for workloads with small inter-arrival times. We use two solutions to solve this problem. The first is to have a running average of inter-arrival times and predict on the basis of the average-value if it is worth keeping the WFD on. The second is to explicitly help applications that can tolerate delay to batch data (efficacy evaluated in Section 4) for further energy saving.

3.5 Implementation Details

We implement WearDrive on Android 4.4 using Java, C and JNI [21]. It consists of the KV-store, the data transfer library and the code needed for ensuring durability of BB-RAM. WearDrive is accessed via the calls on *all devices* as shown in Table 1. `InsertKV` and `ReadKV` always append the application ID (stored in `handle`) to the key for inserting and reading data. This helps WearDrive isolate data between applications. Privacy is protected by not providing user-space access to BB-RAM. All data is accessed through user space buffers provided to the system calls.

Sensor values are aggregated by WearDrive on a per-sensor basis. Applications can register sensor logs for each sensor. WearDrive directly appends sensor samples to the pre-allocated KV-pair that is buffering the current set of sensor samples. When enough samples are available, WearDrive notifies the corresponding applications.

4 Evaluation

Evaluating wearable applications is hard because of the lack of a standard benchmarking tool that can generate

Workload	Parameters	Application examples
Extended Display	Size and inter-arrival time distribution of data	Email, news, instant messages, status updates from social networks, etc.
Sensors	sampling rate, monitoring period	Physical fitness, sleep quality, heart health monitoring, elder care, etc.
Audio/ Video	Encoding rate, quality, monitoring period	Dash-cam using glasses, sleep quality monitoring.

Table 2: Workloads included in WearBench.

representative workloads that span across wearables and phone. We present WearBench, a framework that is intended to test the impact of data generated by such wearable workloads on performance and energy.

4.1 WearBench

WearBench is an Android app that runs on the phone/wearable for generating the extended-display data and sensor data which represent wearable applications. WearBench runs on the phone when testing the wearable and vice versa so that WearBench does not interfere with the measurements. WearBench defines synthetic data-analytics that can be executed on sensor logs like calculation of running statistical features including average, standard-deviation, k-means, and hourly/diurnal/weekly pattern recognition algorithms – sampling rate and timeliness are configurable. WearBench can create notifications of varying sizes and different inter arrival time distributions. To the best of our knowledge, WearBench is the first framework for benchmarking wearable systems.

We identify several typical data-intensive workloads running on smart wearables (see Table 2). In order to cover a wide variety of users, we abstract the usage pattern as configurable parameters in WearBench.

The aim of our evaluation is to demonstrate the performance and energy benefits to wearable devices from using WearDrive. We also study the impact on the battery life of the phone. Table 4 summarizes the major benefits of WearDrive for wearable applications over the state-of-the-art methods.

4.2 Experimental Setup

We use a low-end mobile platform as a reference wearable device that runs Android 4.4. As shown in Table 3, our reference wearable device compares to Samsung Galaxy Gear smart-watches which have similar hardware and software configurations. While our reference has 1 GB RAM, we use only 512 MB on it for the system to match the amount of RAM on state-of-the-art wearables.

Monsoon power monitor [36] is used to profile energy consumption of the device. We instrument the reference wearable device's battery-leads such that it draws power from the Monsoon power meter instead of a battery. We

Type	Our Reference Wearable	Samsung Gear
Processor	1.2 GHz dual-core	1.2 GHz dual-core
Memory	1 GB RAM	512 MB RAM
Storage	4 GB eMMC flash	4 GB eMMC flash
Network	Bluetooth 4.0 LE, WiFi 802.11 b/g/n	Bluetooth 4.0 LE, WiFi 802.11 b/g/n
Sensors	accelerometer, barometer, compass, GPS, gyroscope, heart rate monitor, magnetometer, altimeter, barometer, UV light sensor, ambient light sensor, BLE and WiFi events, camera, microphone	accelerometer, gyroscope, compass, heart rate monitor, ambient light, UV light, barometer, GPS, microphone, BLE and WiFi events
OS	Android 4.4	Android 4.3+/Tizen

Table 3: Reference wearable device used for evaluation.

Typical Workloads	Battery-Life Improvements
Passive heart-rate monitoring (Section 4.4.1)	39%
Passive movement monitoring (Section 4.4.1)	54%
Taking pictures (Section 4.4.2)	16%
Taking pictures in burst mode (Section 4.4.2)	27%
Passive video monitoring (Section 4.4.3)	33%
Passive audio monitoring (Section 4.4.4)	50%
Batched Notifications (Section 4.6)	149%
Unbatched Notifications (Section 4.6)	24%

Table 4: WearDrive's benefits for typical wearable workloads compared to Google WearSDK.

perform comparative energy calculations by subtracting the base power of the system from the power used when a workload is executed. However, when reporting absolute energy required for a workload we include the base power of the system. We compare WearDrive with the following state-of-the-art storage solutions:

WearableOnly: The wearable applications use the capabilities on the wearable for storage. The phone is used only for Internet connection via tethering. All the computation is performed locally and all data is durably written to local flash. This is the way most fitness/health trackers are implemented on today's wearables.

WearSDK: Android Wear SDK released by Google [3] is one way to span data across wearable and phone. However, this SDK uses flash synchronously on either one of the devices to ensure durability. WearSDK provides a data layer for data synchronization between paired wearable and phone via BLE (i.e., WearSDK-BLE). We extend the data layer and make it support WFD (i.e., WearSDK-WFD) and our hybrid network protocol (i.e., WearSDK-HYN).

4.3 Local Memory vs. Local Flash

We first examine the advantages of BB-RAM over local flash with a set of microbenchmarks. We configure

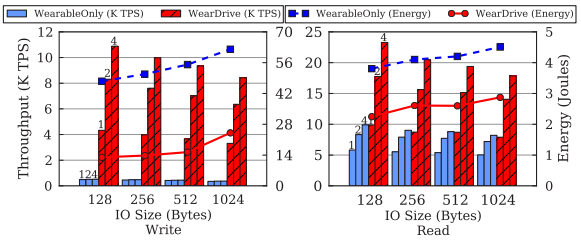


Figure 5: Performance and energy comparison of WearableOnly and WearDrive with varied number (1, 2, 4) of threads.

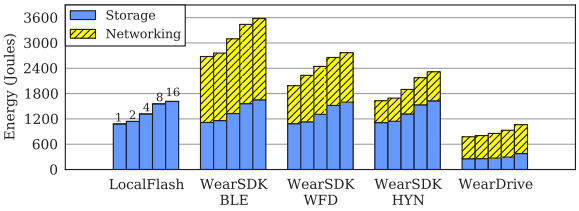


Figure 6: Energy used by various storage systems with varied number (1–16) of sensors sampling values continuously at 1Hz for 24 hours. A typical smart-watch battery contains between 3000–6000 Joules of energy.

WearBench to issue 100 K `InsertKV` and `ReadKV` operations. The size of the data written or read is varied uniformly from 128 bytes to 1 KB. Figure 5 compares the throughput for different data sizes. WearDrive outperforms WearableOnly by 6.65–8.85x on inserts where storage I/O from flash becomes the bottleneck, and 1.57–1.69x on reads where the CPU becomes (single thread) the bottleneck for our system and the flash IOPS for WearableOnly. Moreover, WearDrive's throughput scales linearly till four threads while WearableOnly is saturated by a single thread. Figure 5 also shows the total energy usage of these write/read operations. WearDrive consumes 2.58–3.69x and 1.57–1.70x less power than WearableOnly on inserts and reads respectively, as slow I/O operations on flash cause more CPU cycle wastage, and further increase the energy usage.

4.4 Passive Sensor Data Aggregation

In this section, we demonstrate the benefits of using local and remote BB-RAM for providing durability for sensor data recording over flash.

4.4.1 Fitness Tracker

Fitness/health tracking applications collect sensor values on a periodic basis and update statistics [8]. We use a fitness tracker application that samples various sensors at 1Hz and stores them to local flash periodically. We record the storage calls that this application makes for storing sensor logs, and incorporate the workload into WearBench for replaying.

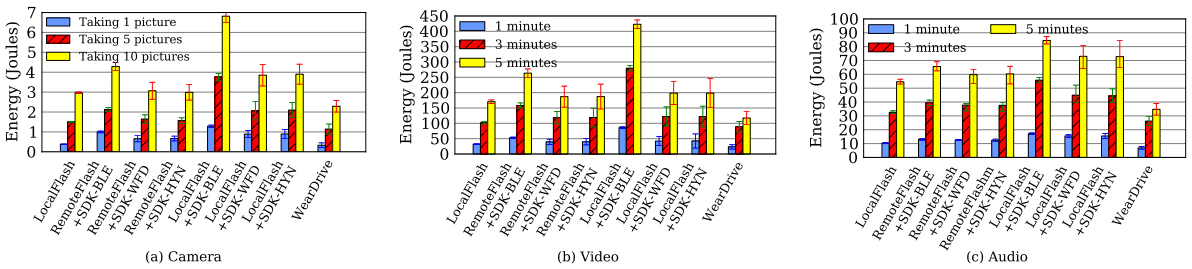
WearDrive aggregates sensor data in BB-RAM and ensures their durability. WearableOnly and WearSDK unfortunately cannot provide such guarantees unless they write every sensor sample through to flash, but they suffer severe performance losses in doing so. As a tradeoff between durability and performance, for these methods, we write the sensor samples to flash when data fills a sector (512 bytes). Every five minutes, all the new data is sent to the phone as sending data to phone at 1Hz leads to significant energy wastage because the network chip would never go into low power mode. Figure 6 shows the total amount of energy in Joules required each day only recording the sensor values. The overall trend across all the systems show that the number of sensors sampled does not severely impact the energy consumption of storage, indicating that the setup costs inside storage stack are the dominant factors for this workload.

WearDrive outperforms the other systems by up to 3.31x and provides better durability. When sampling 16 sensors every second for the whole day and writing them to flash, the storage system (hardware and software) requires 1760 Joules. Considering a typical smart-watch battery that contains 4000 Joules (1.1 Watt-Hour) of energy, writing sensor data to flash requires 44% of total battery life each day. WearDrive on the other hand consumes 28.25%, which is 1.54x more efficient. Moreover, we find that 89.5%, 68.1% and 58.75% of the battery life is respectively required by WearSDK-BLE, WearSDK-WFD and WearSDK-HYN. While HYN reduces the cost of transmitting data over the network to the phone, the bulk of the cost for these systems is still from using slow flash which wastes energy by delaying CPU and network from going to sleep sooner.

4.4.2 Time Lapse Photography

Time lapse photography applications for smart-glasses allow users to log their outdoors activities without the effort of carrying a bulky camera or phone in the hands. We incorporate a time-lapse photography storage workload in WearBench by recording the storage calls of a time-lapse application on Android which takes high-quality pictures at each timer event. Each picture has 2592x1944 dimensions with average size of 900 KB. A few pictures are taken once every few minutes. The results of the workload are shown in Figure 7(a) where average energy required on the wearable per round of photography are reported. LocalFlash stores the pictures only on the wearable. RemoteFlash stores the photos on the phone's flash with the various WearSDK networking solutions. We also test scenarios where photos are stored locally in flash but are also transmitted to the phone with WearSDK. Finally, WearDrive does local BB-RAM to remote BB-RAM copy with HYN.

Results indicate that storing pictures synchronously on



(a) Camera (b) Video (c) Audio

Figure 7: Energy usage on wearable of taking pictures, video recording, audio recording.

local flash is 1.78–2.97x more energy efficient than on remote flash. This implies that eliminating local durable storage is not enough for energy savings and that the energy cost of storage is from the CPU that has to be idle while the flash-IO completes. WearDrive's ability to provide durability for data with local BB-RAM to remote BB-RAM copy reduces the amount of time the CPU and network have to be active and this significantly improves energy efficiency while enabling applications on the phone to have access to the photos.

4.4.3 Passive Video Monitoring

Recording video while traveling, riding or commuting allows a person to have evidence in case of an accident. We add a prototype dash-cam scenario in WearBench that emulates storage calls due to recording of video. Video is shot at 30 FPS (frames per second) with 480p resolution. We buffer video on the wearable for 1, 3 and 5 minutes and then transmit it to phone. Figure 7(b) demonstrates that the energy required scales linearly with data size for large workloads indicating that setup costs in storage show up only for small workloads as in the case of fitness tracking applications.

4.4.4 Passive Audio Monitoring

Some sleep disorders like snoring, bruxism, etc can be diagnosed by passive audio recording on a wearable during the night [53]. We create a prototype passive audio monitoring workload in WearBench by recording all the calls made by an audio recorder application. Audio is sampled for a few minutes continuously several times when the wearable detects motion or noises. The sampling rate for audio is set to 16 KHz, the audio format is PCM 16 bits per sample. As shown in Figure 7(c), compared with the state-of-the-art solutions, WearDrive consumes 1.5x less power than LocalFlash by taking advantages of in-memory store and memory-to-memory data transfer. Combined with the previous results, this provides further evidence that WearDrive can provide benefits regardless of the sensor used as the energy overhead is largely a function of data size.

4.5 Impact on Smart-phone

In this section, we evaluate the energy usage on the phone side and show how WearDrive can improve the lifetime of wearables by leveraging only a negligible portion of phone's larger battery capacity. To understand the energy impact on the phone accurately in this context, we use the same reference hardware in Table 3 as a phone. Note that this is a hardware specification similar to most low-end phones on the market today. However, we use a 2000mAh battery as the reference battery when evaluating the energy impact on the phone.

Energy cost of storage: We reuse the fitness monitoring application workload from Section 4.4.1. Recall that for recording 16 sensors at 1Hz for 24 hours requires 28.25% of the battery life on the wearable instead of 44.0% when writing the data to the flash on the wearable. For this experiment, we find that the phone requires 1369 Joules of energy. This energy accounts for 5.1% of the battery on the phone but this leads to savings of 16% of the battery on the wearable. Considering the fact that the batteries on wearables are usually 5–7x smaller than on low-end phone, this is a valuable tradeoff to make. Moreover, having the data on the phone enables phone to perform analytics and provide more energy savings for the wearable device.

Energy cost of computation: We implement `Mean` and three commonly used data mining algorithms in WearBench: `k-NN` (k-Nearest Neighbor) for classification [24], `ID3` (Iterative Dichotomiser 3) for generating decision tree [20], and `k-means` for cluster analysis [23] for detecting patterns in streams of sensor data to find out when user's heart rate is high [4], when a user snores during the night [53], the levels of UV exposure [51], etc. WearableOnly refers to the baseline, in which records are stored in SQLite and data analytics run on wearables. WearDrive performs computation on the phone with the data in WearKV. The sensor data are aggregated over three days.

Table 5 shows that WearableOnly method of storing and computing on the wearable consumes a significant portion of wearable's battery life, ranging from 14.72% to 27.12%. For smaller data sets the data can be read

Algorithms	Mean		k-NN		ID3		k-means	
Schemes	% of battery life on		% of battery life on		% of battery life on		% of battery life on	
	wearable	phone	wearable	phone	wearable	phone	wearable	phone
WearableOnly	14.72%	-	18.85%	-	20.24%	-	27.12%	-
WearableOnly+InMem	0.83%	-	4.96%	-	6.56%	-	13.23%	-
WearDrive	0.87%	0.21%	0.87%	0.83%	0.87%	1.08%	0.87%	2.09%

Table 5: WearDrive saves wearable's battery by trading it with the phone's battery.The battery capacities of the wearable and phone used in the experiments are 300 mAh and 2000 mAh respectively.

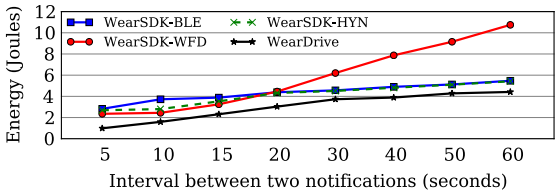
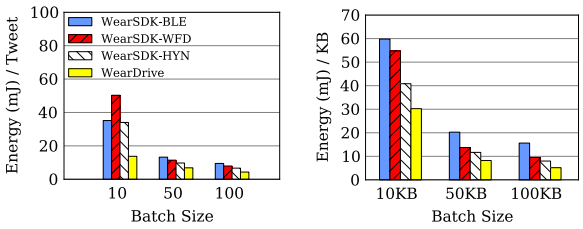


Figure 8: Energy usage of receiving 10 notifications (10KB size) with varied interval between notifications.



(a) Batching Tweets (b) Batching Email

Figure 9: Performance and energy usage of notification workload with different data size.

into memory all at once and computed over as opposed to reading data from flash in batches. We refer to this solution as WearableOnly+InMem. It reduces the energy usage dramatically, but it works only for small workloads that fit in memory. However, when sampled at a higher rate (required usually when the user is running or biking) of over 10Hz, sensor data beyond a few hours will not fit in the memory of the wearable. While such workloads may not fit in the phone's memory either, the phone's larger battery takes much smaller impact.

When the computation is shifted to the phone by WearDrive, it consumes a trivial portion (0.21%–2.09%) of phone's battery life, but reduces the energy usage on wearables to be only 0.87% of wearable's battery life for issuing the arithmetic functions. As future work, we wish to explore when offloading computation to the cloud pays-off with respect to energy. Offloading to the cloud incurs more energy overhead due to data transmission across a wide area with WiFi or LTE. For instance, uploading 8 MB data to Google Drive [11] consumes 3.14x more power than writing to local flash in our experiment setup (with perfect WiFi conditions).

4.6 Extended Display Workload

In this experiment, we demonstrate the benefits of WearDrive to efficiently store extended-display data durably. We use WearBench to emulate application patterns from representative workloads of Twitter [28], Instagram [17], and email [52] applications with various parameters (size and interarrival time).

Varying inter-arrival times. In order to model more notification workload patterns, we vary the interval between tweets from 5 to 60 seconds and measure the energy-impact from storing them durably on the wearable. Figure 8 shows these results.

WearDrive reduces energy usage by 1.2–2.9x compared with the default option of WearSDK-BLE for today's wearable applications. The benefits are made possible not only because of the performance benefits of BB-RAM, but also because of the energy-benefits of HYN. Faster storage operations help the CPU and network go back to sleep faster and reduce the energy footprint.

With HYN, WearDrive uses WFD when the average interval between notifications is small enough to warrant keeping WFD active (20 seconds for our hardware). When the interval is further increased, WearDrive will intelligently turn off WFD and use BLE to send notifications. The hybrid networking protocol also brings benefit to WearSDK (see WearSDK-HYN in Figure 8). For long intervals, WearDrive still performs better than WearSDK-BLE, because of its faster storage.

Effects of batching notifications. Buffering data on the phone gives HYN more opportunity to exploit the energy efficiency of the WFD protocol. We vary the size of the notifications pushed by the phone to wearable from 128 bytes to 1KB. The batch size that the data is sent ranges from 10 to 100. This experiment allows us to study the energy-benefits of delaying notifications from applications that are less interactive than instant messages, such as social networking updates and even email in some cases.

Figure 9(a) shows the results for tweets which are short social networking messages that can tolerate delay. Compared to WearSDK-BLE, WearDrive takes 2.93x less time, while saving energy by 2.23x. The overhead of

WearSDK is reduced with WFD and HYN for large number of notifications. For small number of notifications such as 10 notifications, HYN will use BLE instead of WFD for data transfer. The execution time of WearSDK-WFD is less than WearSDK-BLE and WearSDK-HYN, but its energy usage is larger as the overhead on WiFi discovery and connection offsets its benefit on data transfer. WearDrive is 1.81x more energy efficient than WearSDK-HYN because of BB-RAM's fast durability.

Likewise for email, as shown in Figure 9(b), the benefits of HYN when batching when possible are apparent. However, WearDrive is 2.49x more energy efficient than WearSDK-HYN because of the fast durability guarantee provided by BB-RAM. Overall, WearDrive helps extended-display applications not only by making the energy-batching tradeoff straightforward to exploit but also by providing benefits for applications that are interactive by enabling fast durability.

5 Related Work

WearDrive is built upon existing work on mobile storage systems, hybrid wireless networks and data management for Internet of Things (IoT).

Energy-efficient mobile storage: Kim et al. [26] provided the evidence that slow flash technologies such as SD and eMMC are the primary performance bottleneck for several classes of mobile applications. Our previous work [31] studied the energy overhead of mobile storage systems and found that the mobile software stack consumes more power than storage hardware. These findings motivate our work, as these overheads become more prominent on wearables where the battery is more constrained than on phones.

Recent optimizations to mobile storage [27, 22] address some of the performance problems, but flash is still 10,000x slower compared to DRAM. Emerging non-volatile memory (NVM) technologies like PCM [29, 30, 6] are not yet available in the market. Battery-backed RAM [34, 38, 32] is viable because batteries, DRAM and flash are pervasive in mobile systems. Luo et al. [34] proposed QuasiNVRAM that is a dedicated, known, contiguous region of physical memory to provide performance benefits for phone applications that use SQLite on Android. WearDrive's BB-RAM improves upon QuasiNVRAM by dynamically adapting to memory pressure, not losing data during any class of crashes and by exploiting application characteristics to provide energy and performance benefits.

Rio [45], BlueFS [39], EnsemBlue [40] Simba [1], Segank [47], Bayou [49] and PersonalRAID [46] are distributed file system techniques to share personal data efficiently across mobile consumer electronic devices. WearDrive is an energy-efficient storage system for data intensive wearable workloads like extended-display and sensor data analysis where the workload characteristic of focus on the newest data is exploited to provide a quick and energy-efficient mechanism to span data and computation across the wearable and the phone.

Data management for IoT: Time-series databases [18, 15, 50] enable computations over logs of sensor values. WearDrive is designed to provide time-series data from sensors to applications on the phone at a low-energy cost to enable such computations [37, 54, 42]. Android Wear SDK [3] provides a library to share data between wearables and phone via Bluetooth. WearDrive additionally takes energy-efficiency as its primary design consideration, exploits the recency-focused nature of wearable applications and provides a low-energy durable storage and communication mechanism. WearDrive can also provide sensor data to cloud-based fitness APIs [16, 12, 4] on the phone at a low-energy cost.

Energy-efficient hybrid networks: BlueFi [2], TailEnder [7], Turducken [48], WASP [25], CoolSpots [41] and Bluetooth high speed wireless [10] design heterogeneous networks for efficient data transfer. We draw upon these works and present an energy-efficient hybrid data transfer mechanism by exploiting application knowledge. We find that awareness of data transfer size coupled with the technique where BLE connection is used to predict WiFi's quality enables a mechanism to send data efficiently.

6 Conclusion

WearDrive demonstrates that battery-backed RAM (BB-RAM) can provide significant performance and energy benefits for wearable applications. It also shows how Bluetooth and WiFi can be used in combination to provide a low-energy communication link (HYN) between the wearables and the phone. BB-RAM in combination with HYN provides a quick and energy-efficient way for wearable applications to span data across all the devices on the body enabling new functionalities for users. We validate these benefits with various typical wearable applications using a new wearable benchmarking suite that we develop, and show that WearDrive is 1.16-1.55x more energy-efficient compared to existing solutions. WearDrive can leverage phone's capabilities to reduce energy usage of wearables by up to 15.21x, with trivial impact on phone for realistic wearable workloads.

Acknowledgments

We would like to thank Karsten Schwan and Moinuddin K. Qureshi for their valuable feedback on this work. We also thank our shepherd Theodore Ts'o as well as the anonymous reviewers. This research was performed when the lead author was an intern at Microsoft.

References

[1] AGRAWAL, N., ARANYA, A., AND UNGUREANU, C. Mobile data sync in a blink. In *Proc. HotStorage'13* (San Jose, CA, June 2013).

[2] ANANTHANARAYANAN, G., AND STOICA, I. Blue-Fi: Enhancing Wi-Fi Performance using Bluetooth Signals. In *MobiSys'09* (Krakow, Poland, June 2009).

[3] ANDROID WEAR API. https://developer.android.com/design/wear/.

[4] APPLE HEALTHKIT. https://developer.apple.com/healthkit/.

[5] APPLE WATCH. http://www.apple.com/watch/apple-watch/.

[6] BADAM, A. Impact of Persistent Random Access Memory on Software Systems. *IEEE Computer Society* (May 2013).

[7] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proc. IMC'09* (Chicago, Illinois, Nov. 2009).

[8] BARUA, D., KAY, J., AND PARIS, C. Viewing and Controlling Personal Sensor Data: What Do Users Want? *Persuasive* (2013), 15–26.

[9] BATTERY DENSITY TRENDS RESEARCH BY ARGONNE NATIONAL LABORATORY. http://ec.europa.eu/dgs/jrc/downloads/events/20130926-eco-industries/20130926-eco-industries-miller.pdf.

[10] BLUETOOTH HIGH SPEED WIRELESS TECHNOLOGY. https://www.bluetooth.org/en-us/marketing/high-speed-technology.

[11] GOOGLE DRIVE. http://drive.google.com.

[12] GOOGLE FIT SDK. https://developers.google.com/fit/.

[13] GOOGLE GLASS HARDWARE. https://support.google.com/glass/answer/3064128?hl=en.

[14] GOOGLE GLASS SOFTWARE. https://developers.google.com/glass/tools-downloads/system.

[15] GUPTA, T., SINGH, R. P., PHANISHAYEE, A., JUNG, J., AND MAHAJAN, R. Bolt: Data management for connected homes. In *Proc. NSDI'14* (Seattle, WA, 2014).

[16] HA, K., CHEN, Z., HU, W., RICHTER, W., PILLAI, P., AND SATYANARAYANAN, M. Towards Wearable Cognitive Assistance. In *Proc. 12th ACM MobiSys* (Bretton Woods, NH, June 2014).

[17] HOCHMAN, N., AND SCHWARTZ, R. Visualizing Instagram: Tracing Cultural Visual Rhythms. In *Proc. Sixth International AAAI Conference on Weblogs and Social Media* (June 2012).

[18] HUANG, S., CHEN, Y., CHEN, X., LIU, K., XU, X., WANG, C., BROWN, K., AND HALILOVIC, I. The next generation operational data historian for iot based on informix. In *Proc. SIGMOD'14* (Snowbird, Utah, June 2014).

[19] IPHONE TECHNICAL SPECIFICATIONS. http://www.gsmarena.com/apple_iphone_6_plus-6665.php.

[20] ITERATIVE DICHOTOMISER 3 ALGORITHM. http://en.wikipedia.org/wiki/ID3_algorithm.

[21] JAVA NATIVE INTERFACE. http://developer.android.com/training/articles/perf-jni.html.

[22] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proc. USENIX ATC'13* (San Jose, CA, June 2013).

[23] K-MEANS CLUSTERING ALGORITHM. http://en.wikipedia.org/wiki/K-means_clustering.

[24] K-NEAREST NEIGHBORS ALGORITHM. http://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

[25] KAPLAN, M., ZHENG, C., MONACO, M., KELLER, E., AND SICKER, D. WASP: A Software-Defined Communication Layer for Hybrid Wireless Networks. In *Proc. ANCS'14* (Los Angeles, CA, Oct. 2014).

[26] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting Storage for Smartphones. In *FAST'12* (San Jose, CA, Feb. 2012).

[27] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving Journaling of Journal Anomaly in Android IO: Multi-version B-tree with Lazy Split. In *FAST'14* (Santa Clara, CA, Feb. 2014).

[28] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a Social Network or a News Media? In *Proc. WWW'10* (Raleigh, NC, Apr. 2010).

[29] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proc. FAST'13* (San Jose, CA, Feb. 2013).

[30] LEE, K., KAN, J. J., AND KANG, S. H. Unified Embedded Non-Volatile Memory for Emerging Mobile Markets. In *Proc. ISPLED'14* (La Jolla, CA, Aug. 2014).

[31] LI, J., BADAM, A., CHANDRA, R., SWANSON, S., WORTHINGTON, B., AND ZHANG, Q. On the Energy Overhead of Mobile Storage Systems. In *FAST'14* (Santa Clara, CA, Feb. 2014).

[32] LOWELL, D. E., AND CHEN, P. M. Free Transactions with RioVista. In *Proc. 16th ACM SOSP* (Saint-Malô, France, Oct. 1997).

[33] LUMIA 930 TECHNICAL SPECIFICATIONS. http://www.microsoft.com/en/mobile/phone/lumia930/specifications/.

[34] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi Non-Volatile RAM for Low Overhead Persistency Enforcement in Smartphones. In *HotStorage'14* (Philadelphia, PA, June 2014).

[35] MICROSOFT HOLOLENS. http://www.microsoft.com/microsoft-hololens/en-us.

[36] MONSOON POWER MONITOR. http://www.msoon.com/LabEquipment/PowerMonitor/.

[37] NAM, Y., RHO, S., AND LEE, C. Physical Acitvity Recognition using Multiple Sensors Embedded In a Wearable Device. *ACM Transactions on Embedded Computing Systems 12*, 2 (2013).

[38] NARAYANAN, D., AND HODSON, O. Whole-system Persistence with Non-volatile Memories. In *Proc. ACM ASPLOS'12* (London, United Kingdom, Mar. 2012).

[39] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and Storage Flexibility in the Blue File System. In *Proc. OSDI'04* (San Francisco, CA, Dec. 2004).

[40] PEEK, D., AND FLINN, J. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *Proc. OSDI'06* (Seattle, WA, Nov. 2006).

[41] PERING, T., AGARWAL, Y., GUPTA, R., AND WANT, R. Coolspots: Reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *Proc. MobiSys'06* (Uppsala, Sweden, June 2006).

[42] SAHNI, H., BEDRI, A., REYES, G., THUKRAL, P., GUO, Z., STARNER, T., AND GHOVANLOO, M. The Tongue and Ear Interface: A Wearable System for Silent Speech Recognition. In *Proc. ISWC'14* (Seattle, WA, Sept. 2014).

[43] SAMSUNG GALAXY GEAR SPECS. http://www.samsung.com/us/mobile/wearable-tech/SM-V7000ZKAXAR.

[44] SAMSUNG PHONE TECHNICAL SPECIFICATIONS. http://www.samsung.com/us/mobile/cell-phones/all-products.

[45] SANI, A. A., BOOS, K., YUN, M. H., AND ZHONG, L. Rio: A System Solution for Sharing I/O Between Mobile Systems. In *Proc. 12th ACM MobiSys* (Bretton Woods, NH, June 2014).

[46] SOBTI, S., GARG, N., ZHANG, C., YU, X., KRISHNAMURTHY, A., AND WANG, R. Y. Personalraid: Mobile storage for distributed and disconnected computers. In *Proc. FAST'02* (Monterey, CA, Jan. 2002).

[47] SOBTI, S., GARG, N., ZHENG, F., LAI, J., SHAO, Y., ZHANG, C., ZISKIND, E., KRISHNAMURTHY, A., AND WANG, R. Y. Segank: A distributed mobile storage system. In *Proc. FAST'04* (San Francisco, CA, Mar. 2004).

[48] SORBER, J., BANERJEE, N., CORNER, M. D., AND ROLLINS, S. Turducken: Hierarchical power management for mobile devices. In *Proc. MobiSys'05* (Seattle, WA, June 2005).

[49] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. SOSP'95* (Copper Mountain Resort, Colorado, Dec. 1995).

[50] THE SCALABLE TIME SERIES DATABASE. http://opentsdb.net/.

[51] UVEBAND. http://www.uveband.com/.

[52] XU, F., LIU, Y., MOSCIBRODA, T., CHANDRA, R., JIN, L., ZHANG, Y., AND LI, Q. Optimizing Background Email Sync on Smartphones. In *Proc. 11th ACM MobiSys* (Taipei, Taiwan, June 2013).

[53] ZEO MOBILE SLEEP MANAGER. http://www.engadget.com/products/zeo/sleep-manager/mobile/.

[54] ZHANG, M., AND SAWCHUK, A. A. USC-HAD: A Daily Activity Dataset for Ubiquitous Activity Recognition Using Wearable Sensors. In *Proc. UbiComp'12* (Pittsburgh, USA, Sept. 2012).