Parity Logging with Reserved Space:
Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage

Jeremy C. W. Chan, Qian Ding*, Patrick P. C. Lee, Helen H. W. Chan*

The Chinese University of Hong Kong

FAST'14

The first two authors contributed equally to this work.

Motivation

- **Clustered storage systems** provide scalable storage by striping data across multiple nodes
  - e.g., GFS, HDFS, Azure, Ceph, Panasas, Lustre, etc.
- Maintain data availability with redundancy
  - Replication
  - Erasure coding

Motivation

- With explosive data growth, enterprises move to erasure-coded storage to save footprints and cost
  - e.g., 3-way replication has 200% overhead; erasure coding can reduce overhead to 33% [Huang, ATC'12]
- Erasure coding recap:
  - Encodes data chunks to create parity chunks
  - Any subset of data/parity chunks can recover original data chunks
- Erasure coding introduces two challenges: (1) updates and (2) recovery/degraded reads

Challenges

1. Updates are expensive

- When a data chunk is updated, its encoded parity chunks need to be updated
- Two update approaches:
  - In-place updates: overwrites existing chunks
  - Log-based updates: appends changes

Challenges

2. Recovery/degraded reads are expensive

- Failures are common
  - Data may be permanently loss due to crashes
  - 90% of failures are transient (e.g., reboots, power loss, network connectivity loss, stragglers) [Ford, OSDI'10]
- Recovery/degraded read approach:
  - Reads enough data and parity chunks
  - Reconstructs lost/unavailable chunks

Challenges

- How to achieve both efficient updates and fast recovery in clustered storage systems?
- Target scenario:
  - Server workloads with frequent updates
  - Commodity configurations with frequent failures
  - Disk-based storage
- Potential bottlenecks in clustered storage systems
  - Network I/O
  - Disk I/O

Our Contributions

- Propose **parity-logging with reserved space**
  - Uses hybrid in-place and log-based updates
  - Puts deltas in a reserved space next to parity chunks to mitigate disk seeks
  - Predicts and reclaims reserved space in workload-aware manner

  → **Achieves both efficient updates and fast recovery**

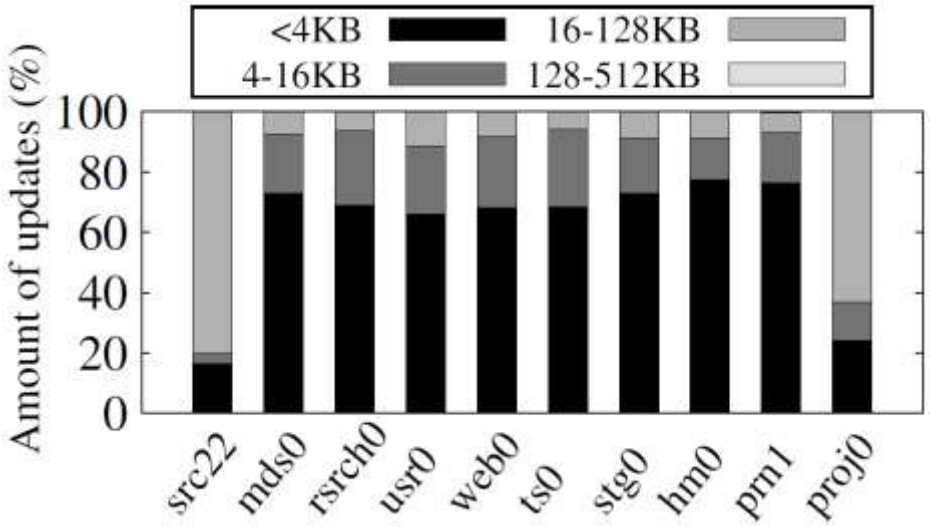
- Build a clustered storage system prototype **CodFS**
  - Incorporates different erasure coding and update schemes
  - Released as open-source software

- Conduct extensive trace-driven testbed experiments

Background: Trace Analysis

- **MSR Cambridge traces**
  - Block-level I/O traces captured by Microsoft Research Cambridge in 2007
  - 36 volumes (179 disks) on 13 servers
  - Workloads including home directories and project directories
- **Harvard NFS traces (DEAS03)**
  - NFS requests/responses of a NetApp file server in 2003
  - Mixed workloads including email, research and development

MSR Trace Analysis



Distribution of update size in 10 volumes of MSR Cambridge traces

- **Updates are small**
  - All updates are smaller than 512KB
  - 8 in 10 volumes show more than 60% of tiny updates (<4KB)

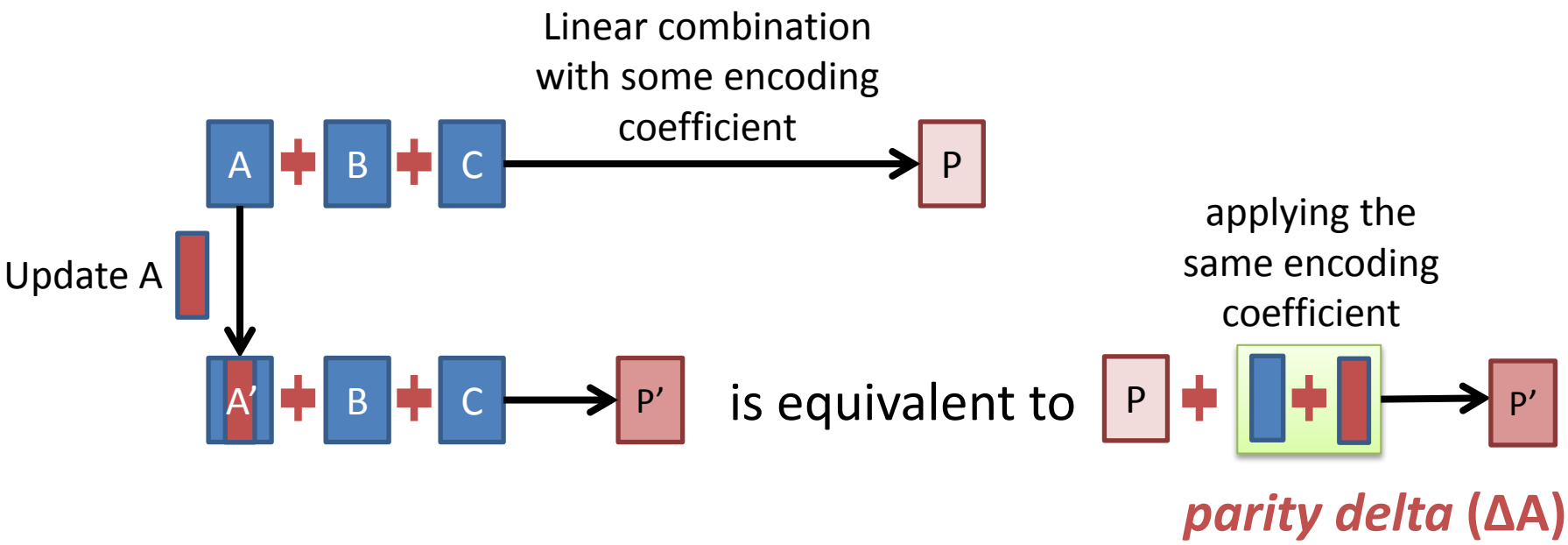
MSR Trace Analysis

- **Updates are intensive**
  - 9 in 10 volumes show more than 90% update writes over all writes
- **Update coverage varies**
  - Measured by the fraction of WSS that is updated at least once throughout the trace period
  - Large variation among different workloads
    - need a dynamic algorithm for handling updates

- Similar observations for Harvard traces

Objective #1: Efficient handling of small, intensive updates in an erasure-coded clustered storage

Saving Network Traffic in Parity Updates



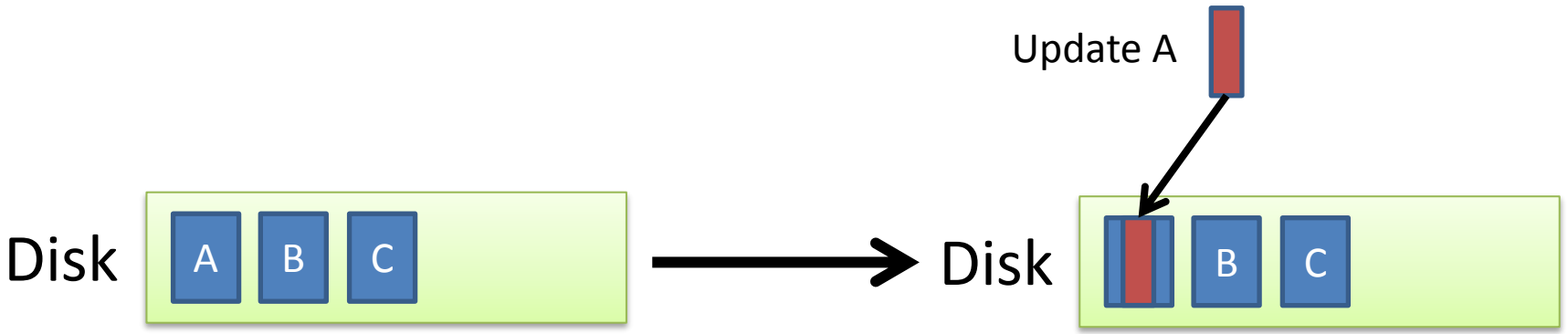
- Make use of **linearity** of erasure coding
  - CodFS reduces network traffic by only sending parity delta

Question:

How to save data update (A') and parity delta (ΔA) on disk?

Update Approach #1: in-place updates (overwrite)

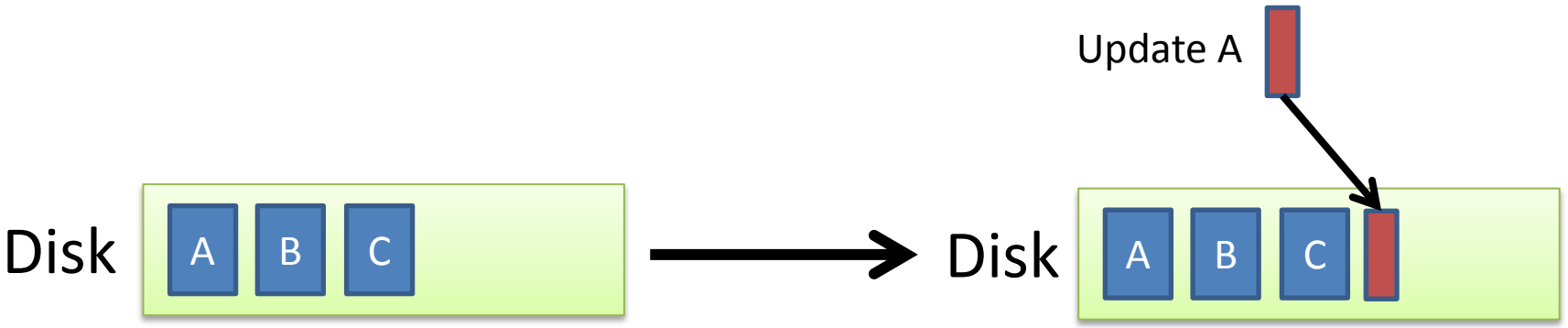
- Used in host-based file systems (e.g., NTFS and ext4)
- Also used for parity updates in RAID systems



Problem: significant I/O to read and update parities

Update Approach #2: log-based updates (logging)

- Used by most clustered storage systems (e.g. GFS, Azure)
  - Original concept from log-structured file system (LFS)
- Convert random writes to sequential writes



Problem: fragmentation to chunk A

Objective #2: Preserves sequentiality in large read (e.g. recovery) for both data and parity chunks

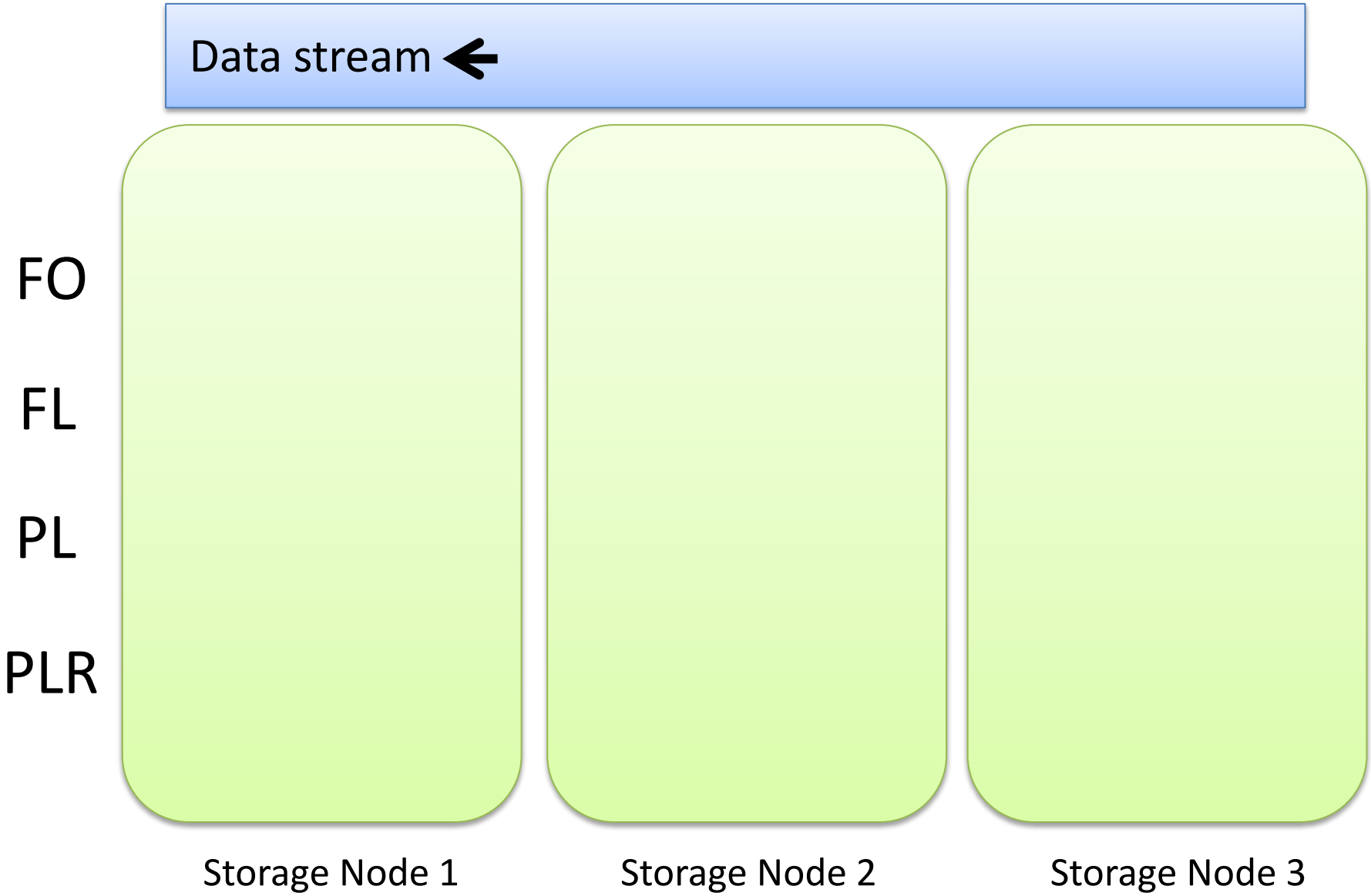
Parity Update Schemes

	Data Update	Parity Delta
Full-overwrite (FO)	O	O
Full-logging (FL)	L	L
Parity-logging (PL)	O	L
Parity-logging with reserved space (PLR)	O	L

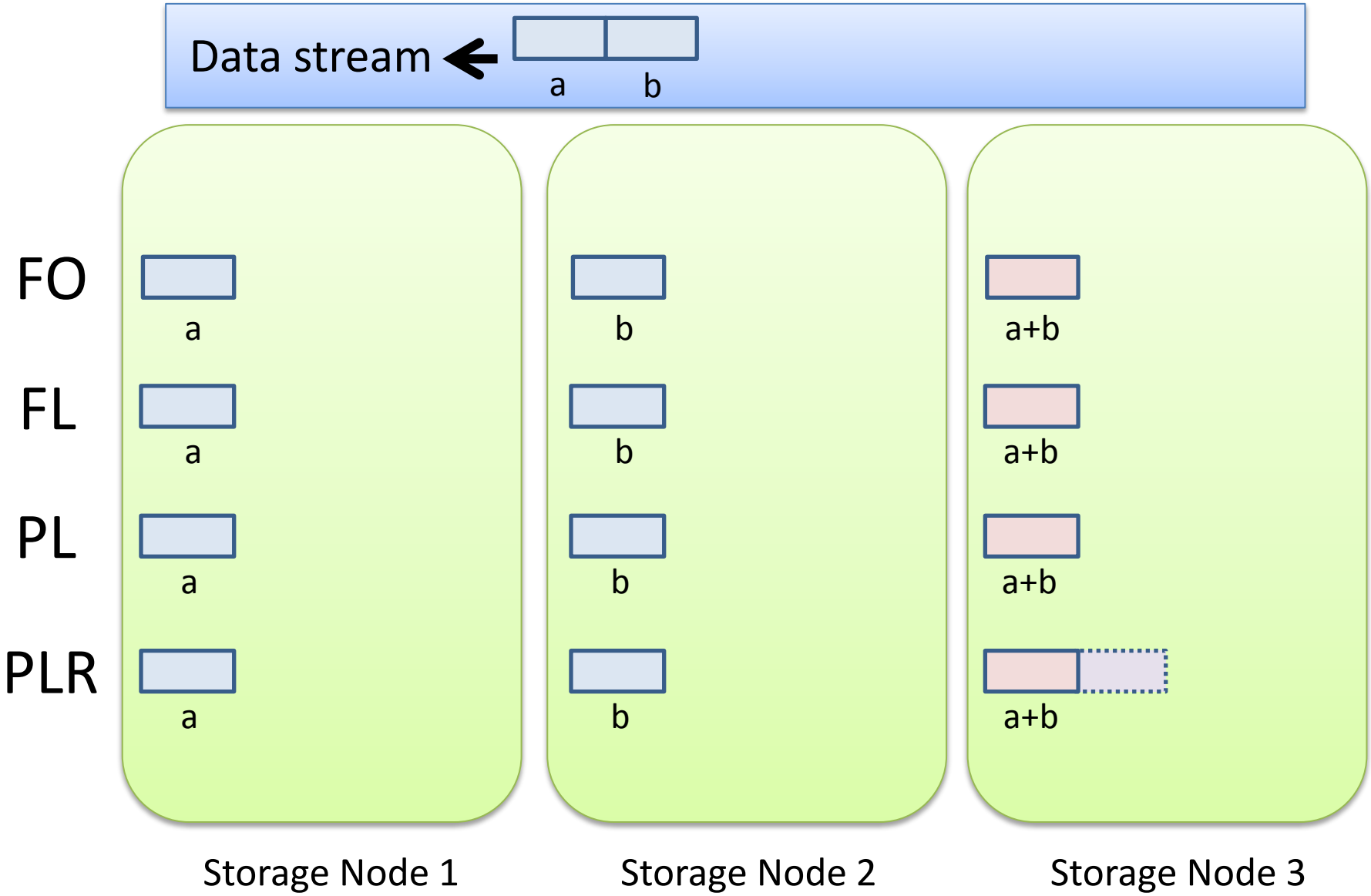
Our Proposal

O: Overwrite L: Logging

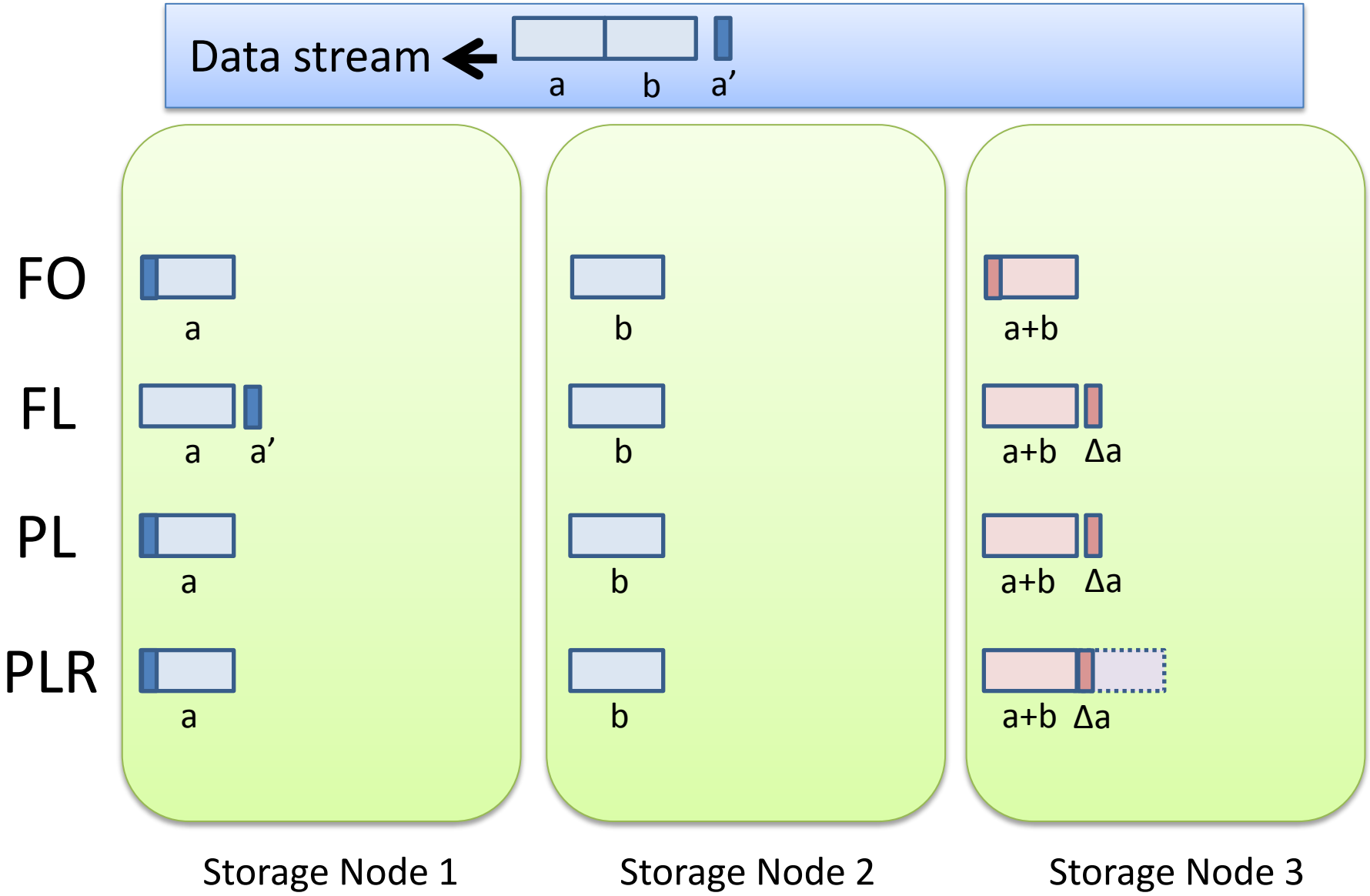
Parity Update Schemes



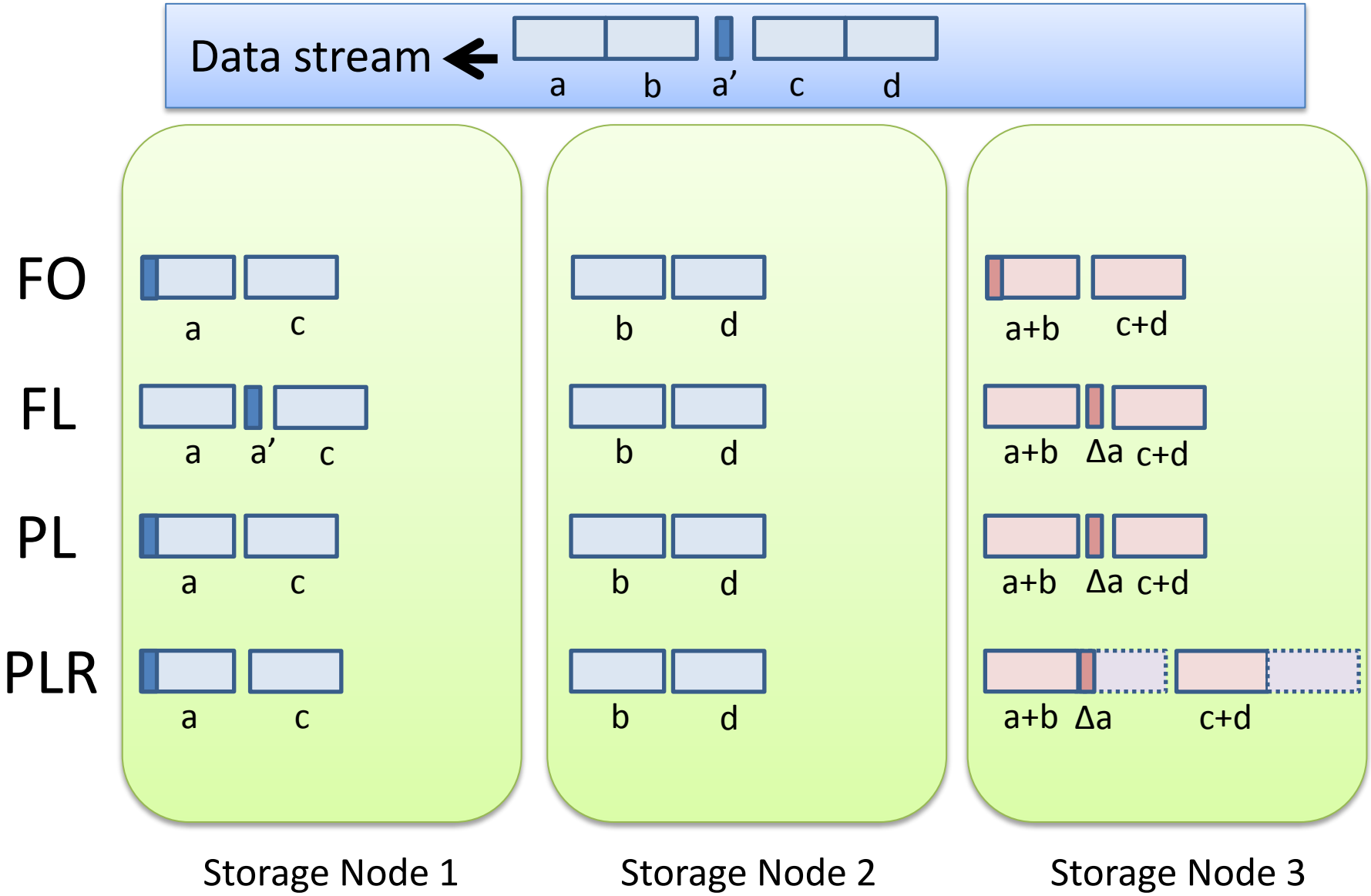
Parity Update Schemes



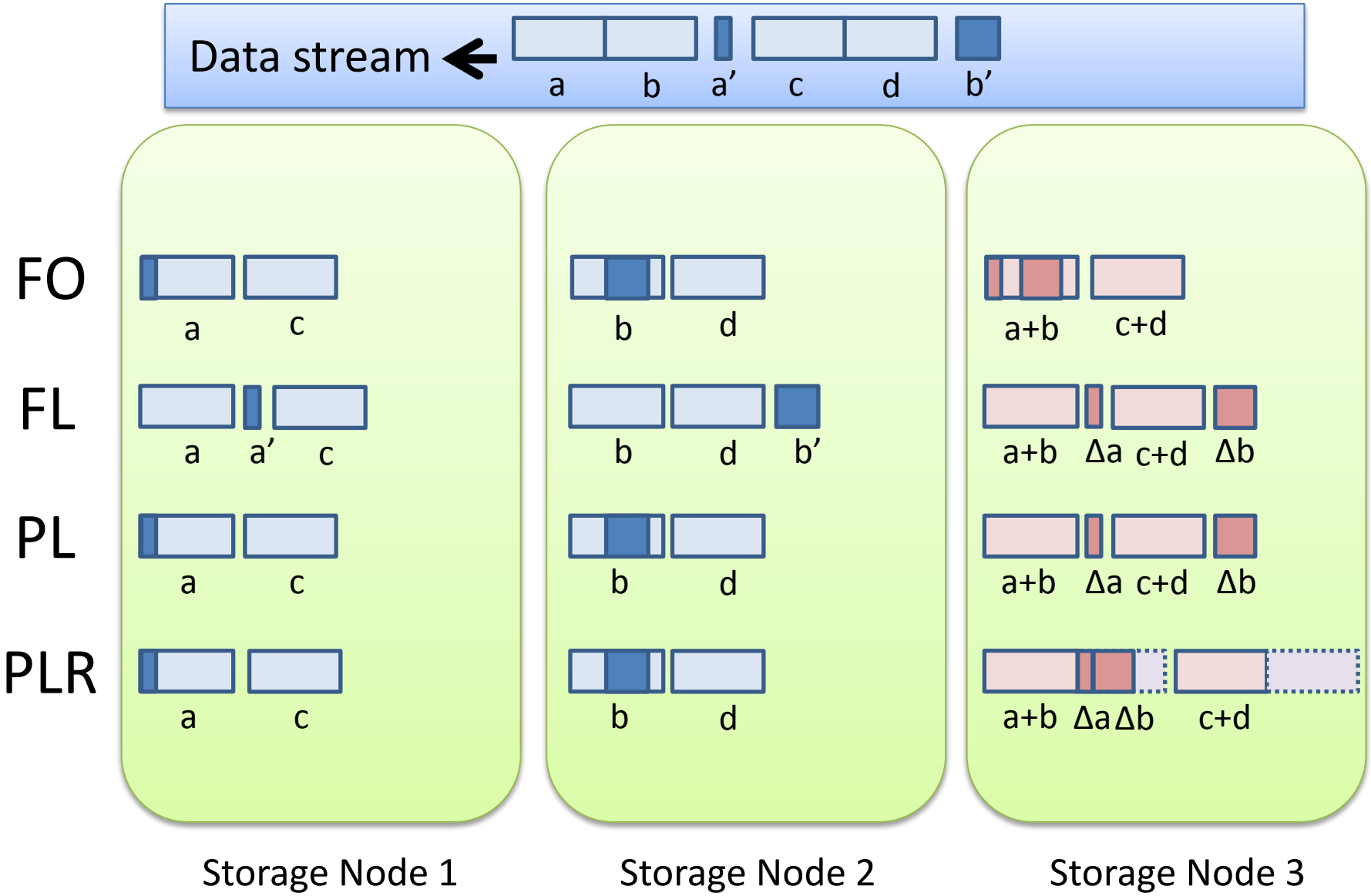
Parity Update Schemes



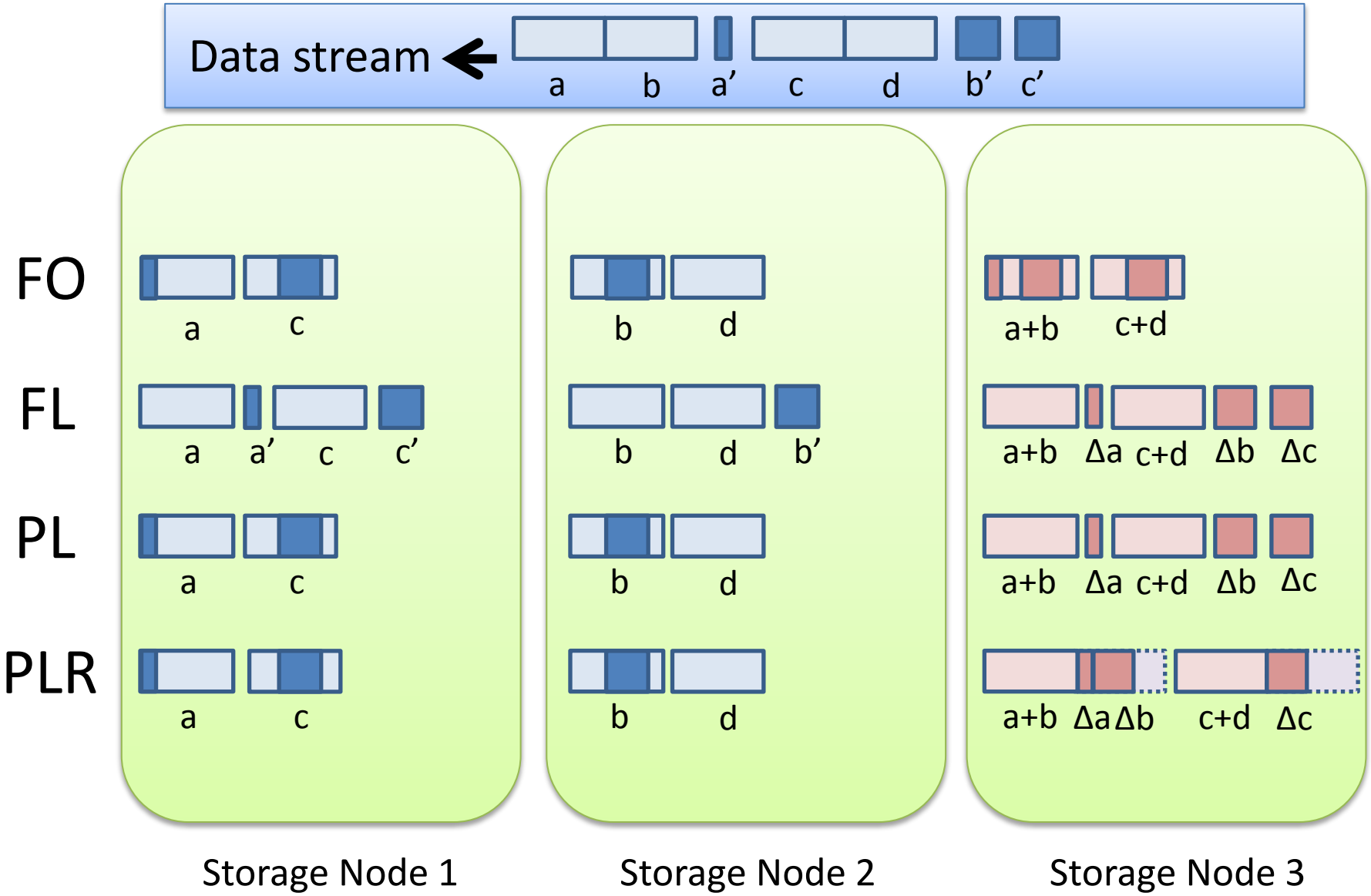
Parity Update Schemes



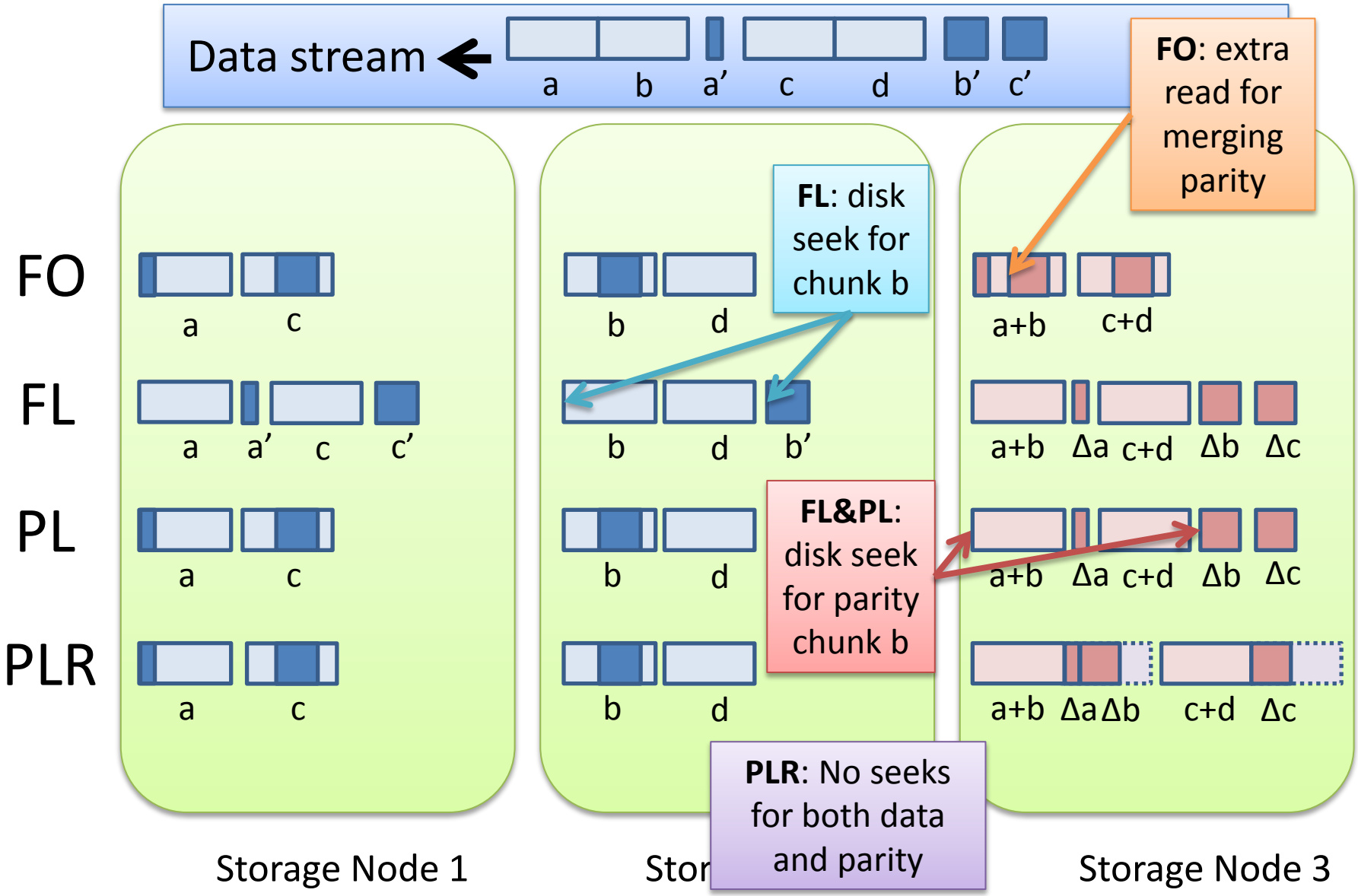
Parity Update Schemes



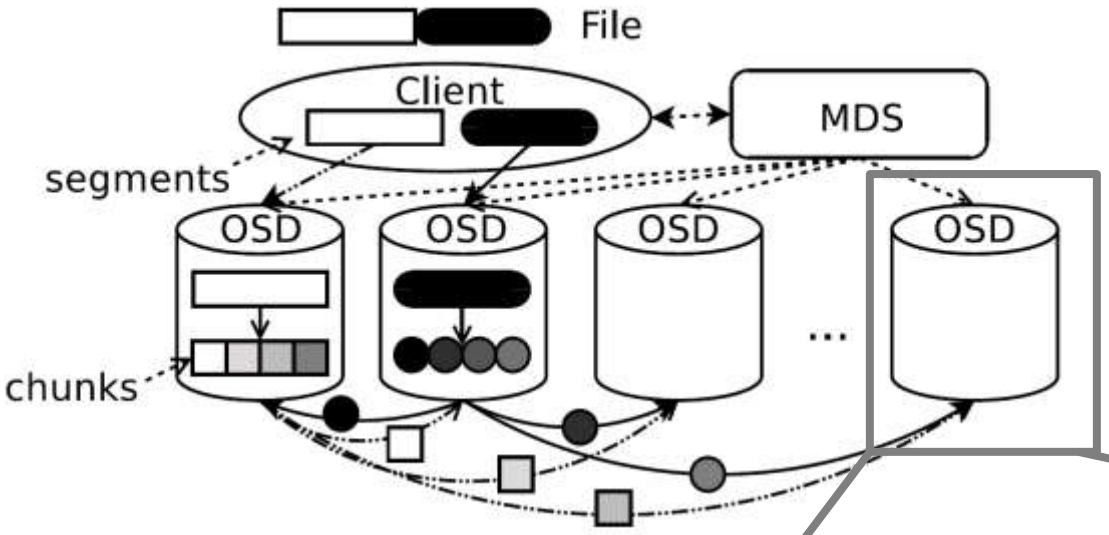
Parity Update Schemes



Parity Update Schemes

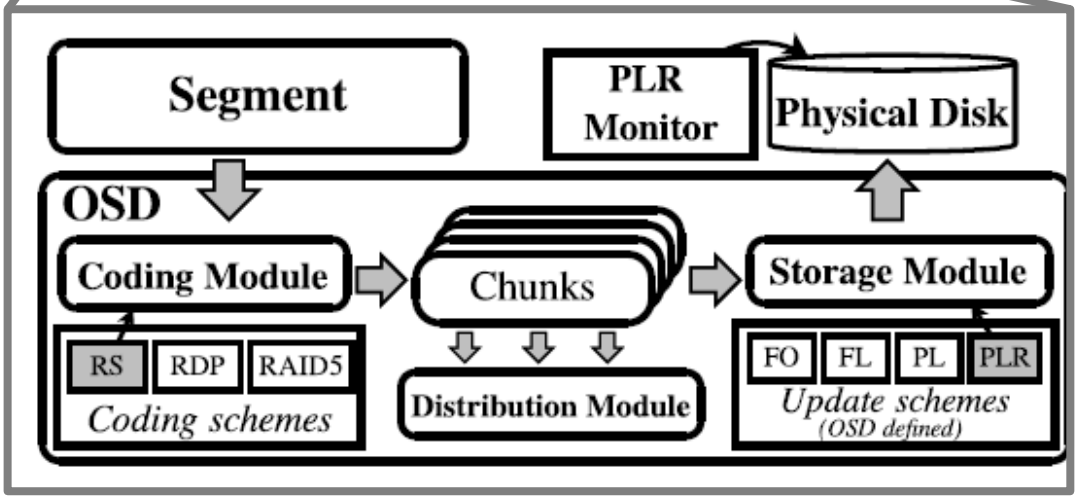


Implementation - CodFS



CodFS Architecture

- Exploits parallelization across nodes and within each node
- Provides a file system interface based on FUSE

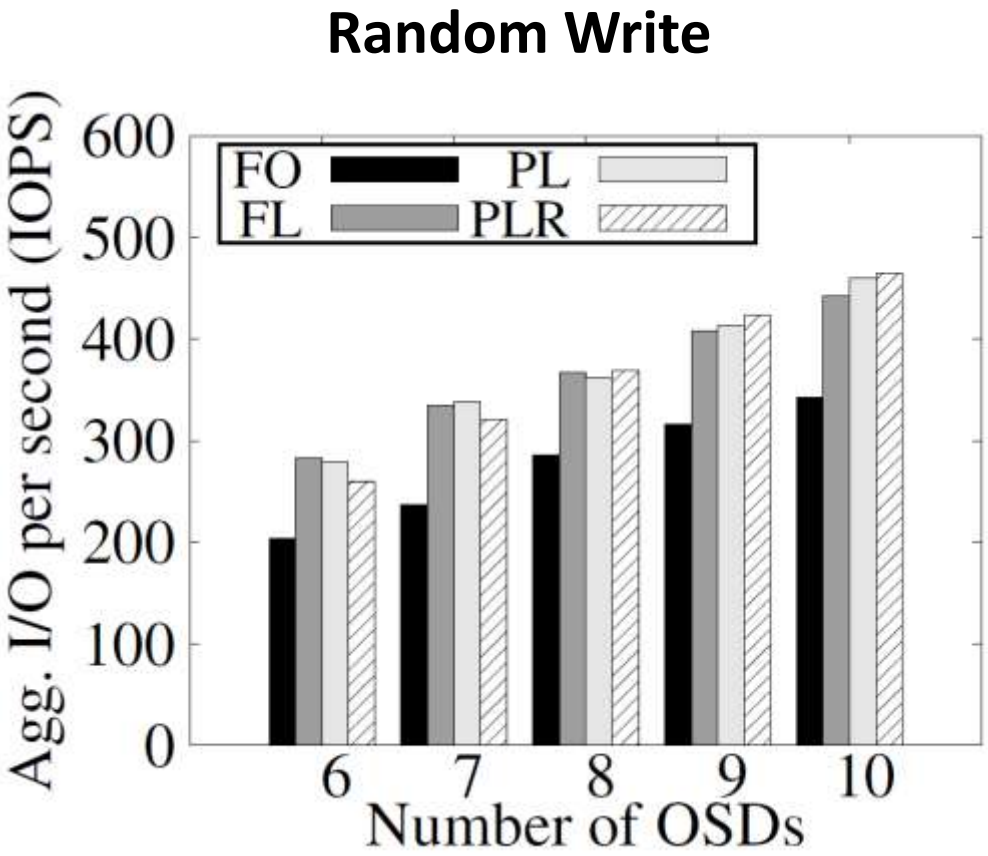


OSD: Modular Design

Experiments

- Testbed: 22 nodes with commodity hardware
  - 12-node storage cluster
  - 10 client nodes sending
  - Connected via a Gigabit switch
- Experiments
  - Baseline tests
    - Show CodFS can achieve theoretical throughput
  - Synthetic workload evaluation ← Focus of this talk
  - Real-world workload evaluation

Synthetic Workload Evaluation



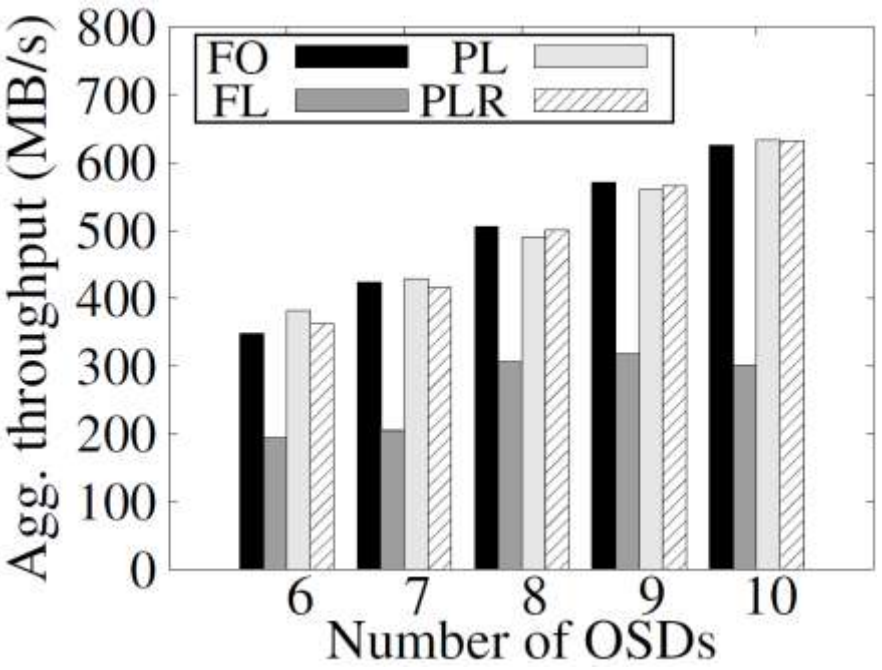
Logging parity (FL, PL, PLR) helps random writes by saving disk seeks and parity read overhead

FO has 20% less IOPS than others

IOZone record length: 128KB
RDP coding (6,4)

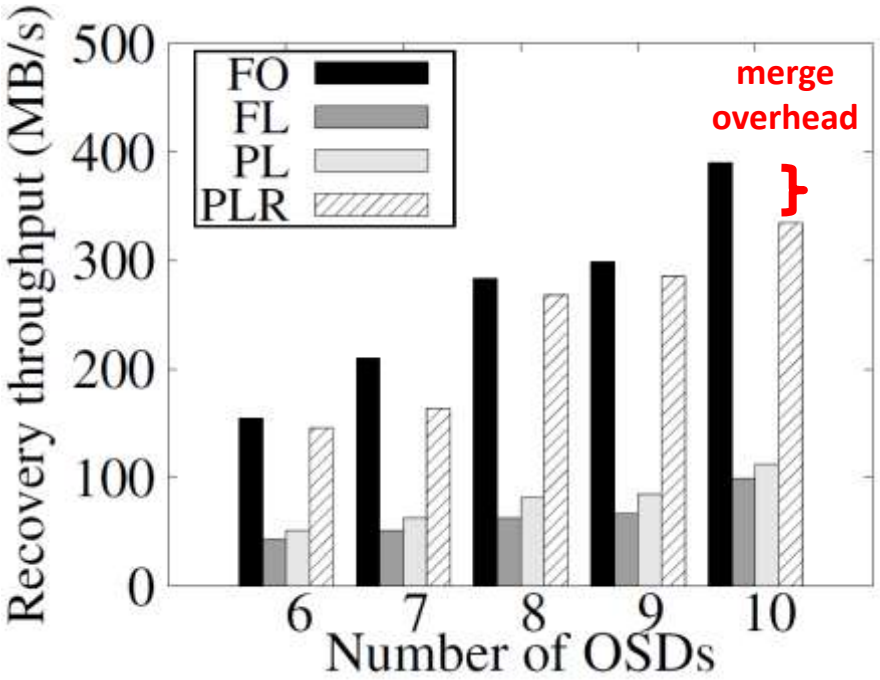
Synthetic Workload Evaluation

Sequential Read



Only FL needs disk seeks in reading data chunk

Recovery



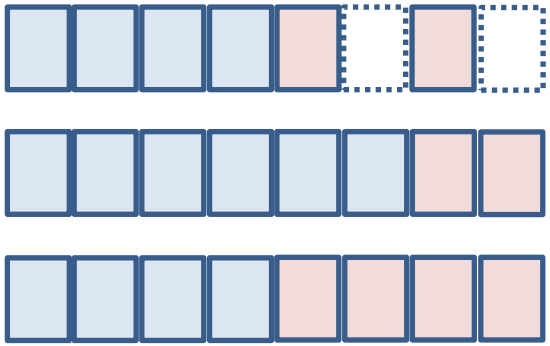
No seeks in recovery for FO and PLR

Fixing Storage Overhead

PLR (6,4)

FO/FL/PL (8,6)

FO/FL/PL (8,4)



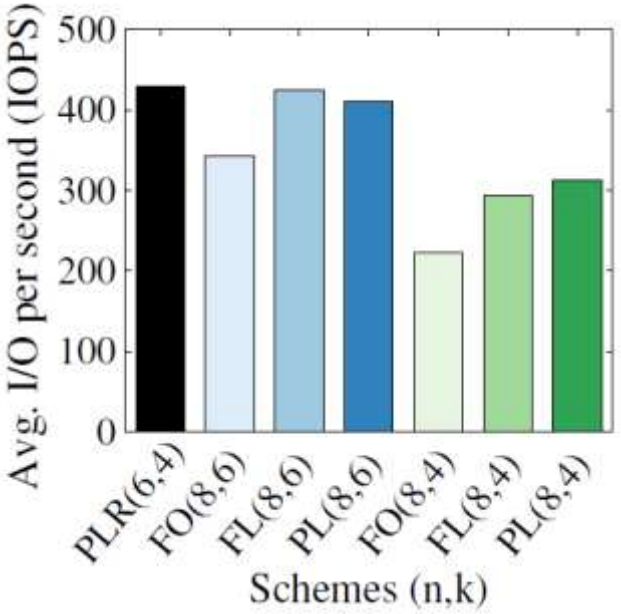
- FO (8,6) is still 20% slower than PLR (6,4) in random writes
- PLR and FO are still much faster than FL and PL

Data Chunk

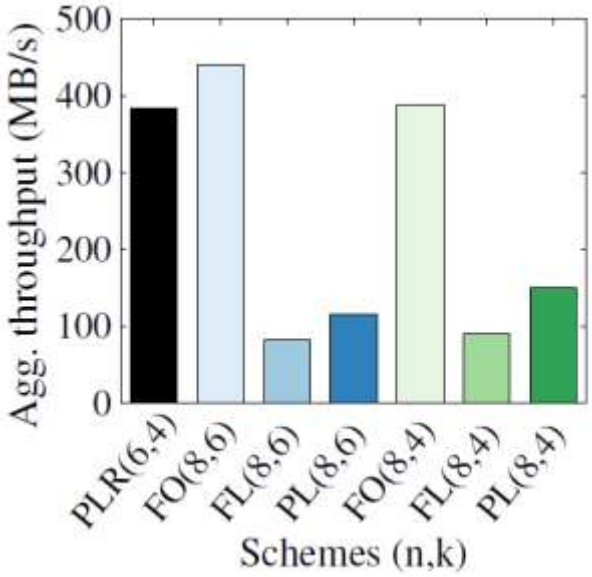
Parity Chunk

Reserved Space

Random Write



Recovery



Dynamic Resizing of Reserved Space

- **Remaining problem**
  - What is the appropriate reserved space size?
    - Too small – frequent merges
    - Too large – waste of space
  - Can we shrink the reserved space if it is not used?
- **Baseline approach**
  - Fixed reserved space size
- **Workload-aware management approach**
  - **Predict**: exponential moving average to guess reserved space size
  - **Shrink**: release unused space back to system
  - **Merge**: merge all parity deltas back to parity chunk

Dynamic Resizing of Reserved Space

Step 1: Compute utility using past workload pattern

$$u_t(p) = \alpha \frac{use(p)}{r_t(p)} + (1-\alpha)u_{t-1}(p)$$

smoothing factor

current usage

previous usage

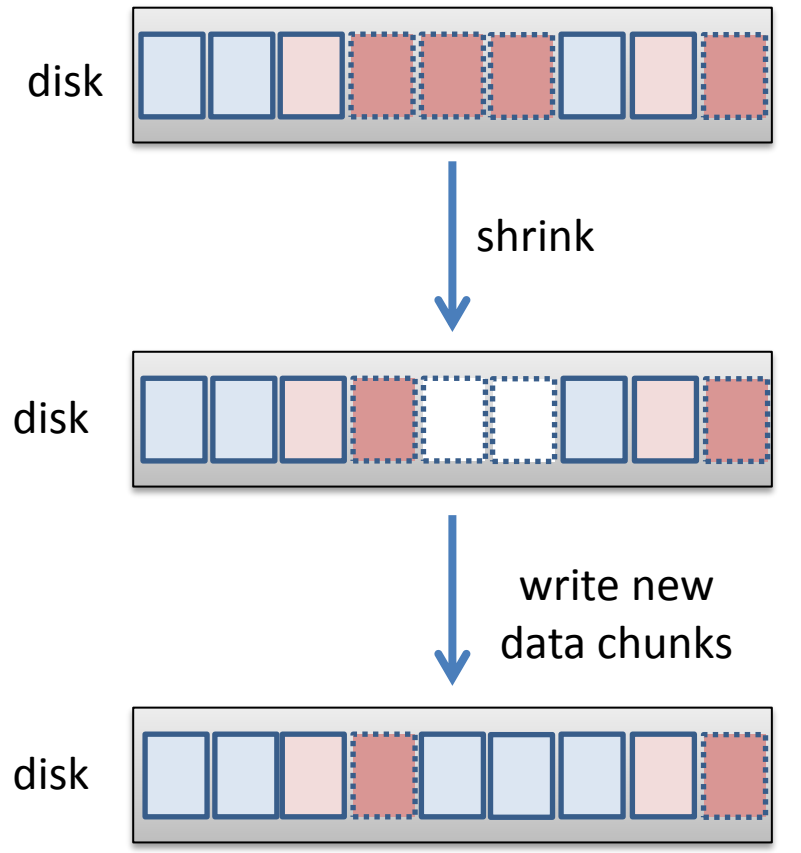
Step 2: Compute utility using past workload pattern

$$c(p) = \left\lfloor \frac{(1-u_t(p))r_t(p)}{ChunkSize} \right\rfloor \times ChunkSize$$

no. of chunk to shrink

Step 3: Perform shrink

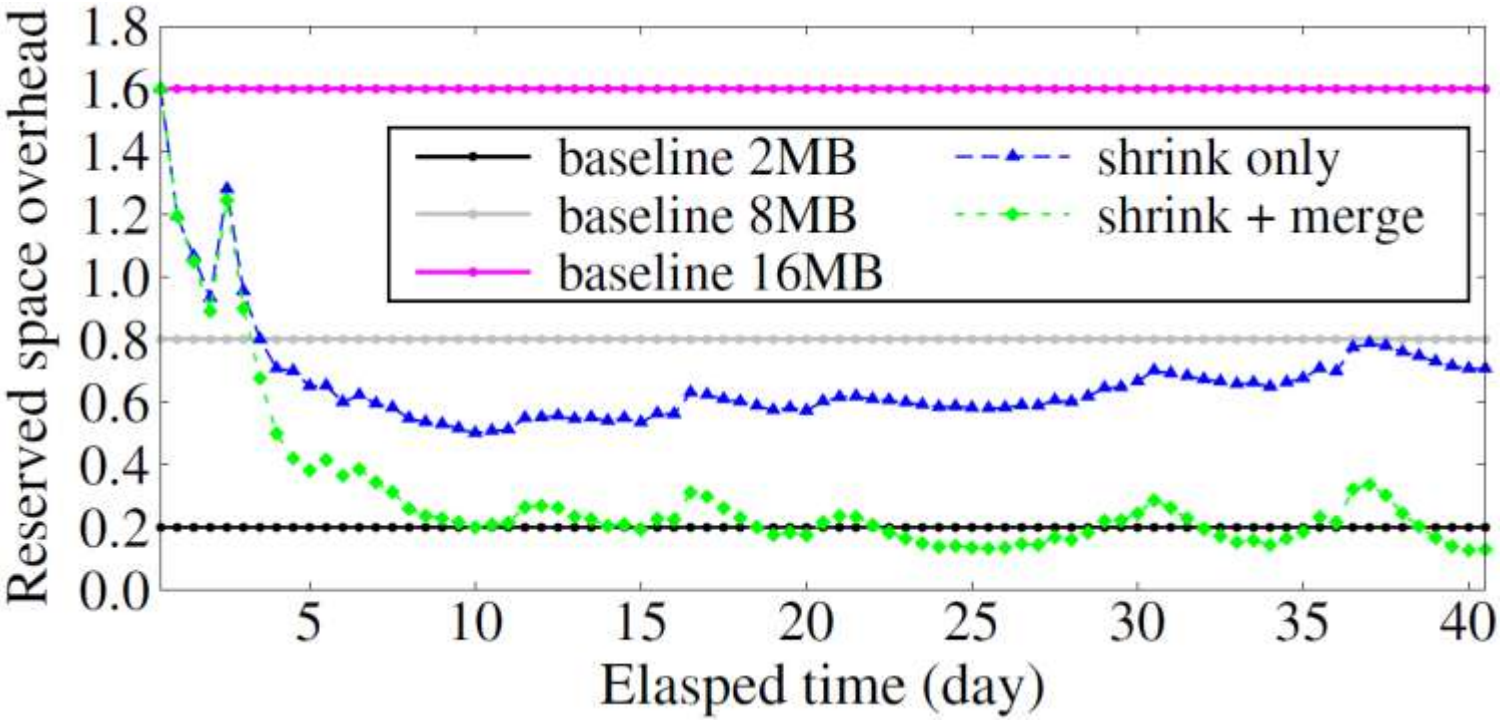
$$r_{t+1}(p) = r_t(p) - c(p)$$



shrinking reserved space as a multiple of chunk size avoids creating unusable “holes”

Dynamic Resizing of Reserved Space

Reserved space overhead under different shrink strategies in Harvard trace



16MB baseline

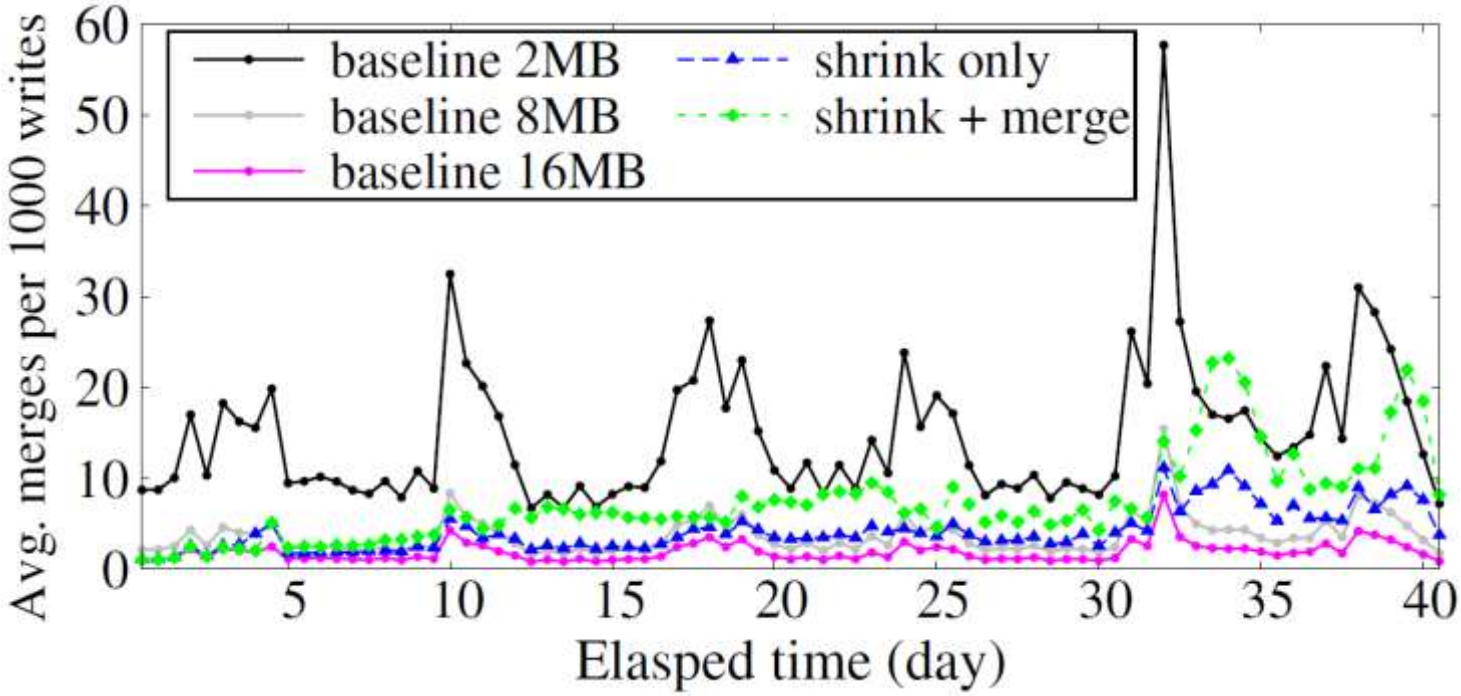
Shrink only performs shrinking at 00:00 and 12:00 each day

Shrink + merge performs a merge after the daily shrinking

*(10,8) Cauchy RS Coding with 16MB segments

Penalty of Over-shrinking

Average number of merges per 1000 writes under different shrink strategies in the Harvard trace



Penalty of inaccurate prediction

Less than 1% of writes are stalled by a merge operation

*(10,8) Cauchy RS Coding with 16MB segments

Open Issues

- Latency analysis
- Metadata management
- Consistency / locking
- Applicability to different workloads

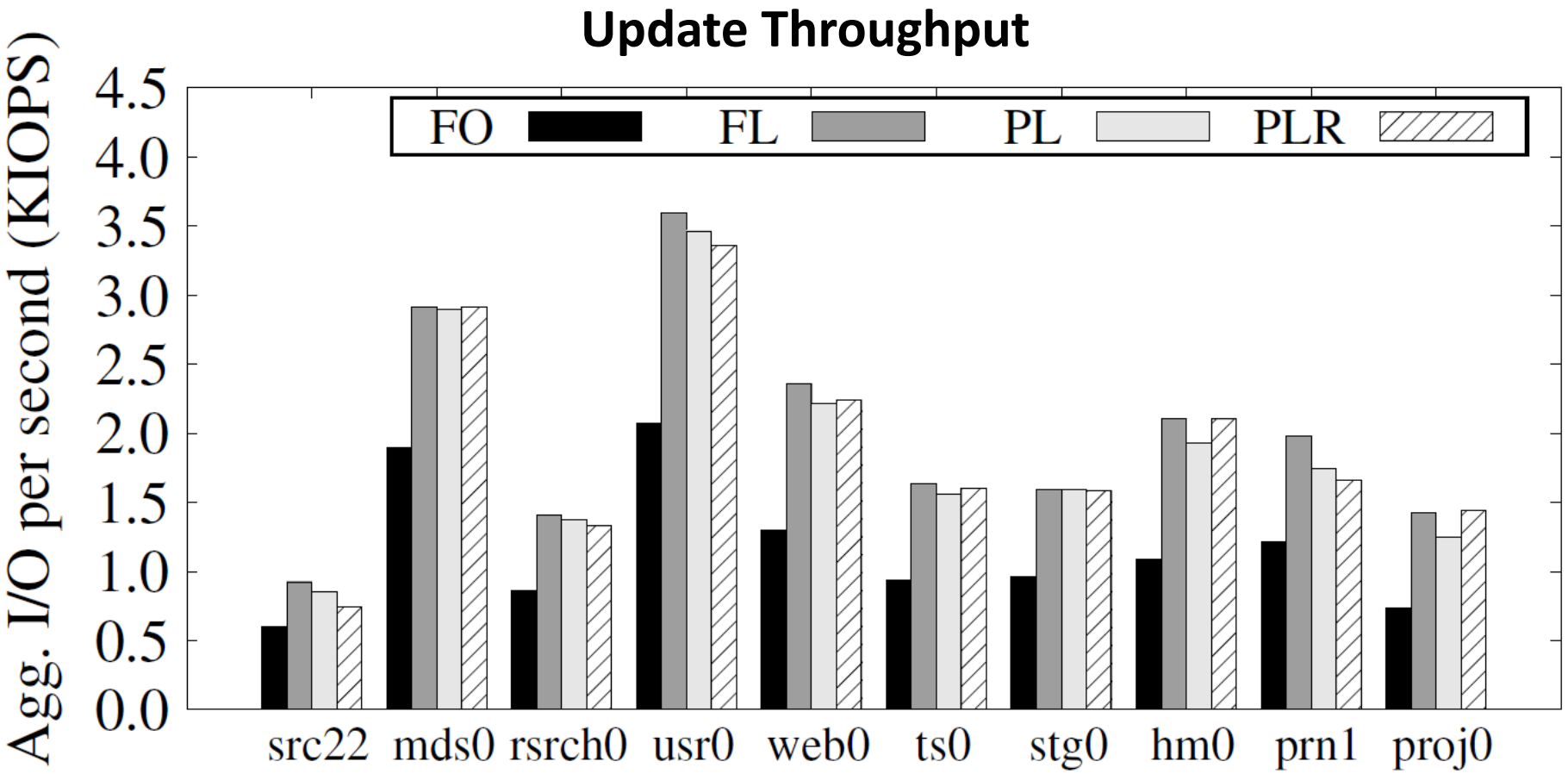
Conclusions

- Key idea: Parity logging with reserved space
  - Keep parity updates next to parity chunks to reduce disk seeks
- Workload aware scheme to predict and adjust the reserved space size
- Build CodFS prototype that achieves **efficient updates** and **fast recovery**
- Source code: **http://ansrlab.cse.cuhk.edu.hk/software/codfs**

Backup

MSR Cambridge Traces Replay

PLR ~ PL ~ FL >> FO



MSR Cambridge Traces Replay

PLR ~ FO >> FL ~ PL

