

DynaMix: Dynamic Mobile Device Integration for Efficient Cross-device Resource Sharing

Dongju Chae, *POSTECH;* Joonsung Kim and Gwangmu Lee, *Seoul National University;* Hanjun Kim, *POSTECH;* Kyung-Ah Chang and Hyogun Lee, *Samsung Electronics;* Jangwoo Kim, *Seoul National University*

https://www.usenix.org/conference/atc18/presentation/chae

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

DynaMix: Dynamic Mobile Device Integration for Efficient Cross-device Resource Sharing

Dongju Chae[¶] Joonsung Kim[†] Gwangmu Lee[†]
Hanjun Kim[¶] Kyung-Ah Chang[*] Hyogun Lee[*] Jangwoo Kim[†]
[¶]*POSTECH* [†]*Seoul National University* [*]*Samsung Electronics*

Abstract

In the era of the Internet of Things, users desire more valuable services by simultaneously utilizing various resources available in remote devices. As a result, cross-device resource sharing, a capability to utilize the resources of a remote device, becomes a desirable feature to enable interesting multi-device services. However, the existing resource sharing mechanisms either have limited resource coverage, involve complex programming efforts for utilizing multiple devices, or more importantly, incur huge inter-device network traffic.

We propose DynaMix, a novel framework that realizes efficient cross-device resource sharing. First, DynaMix maximizes resource coverage by dynamically integrating computation and I/O resources of remote devices with distributed shared memory and I/O request forwarding. Second, DynaMix obviates the need for multi-device programming by providing the resource sharing capability at the low level. Third, DynaMix minimizes inter-device network traffic by adaptively redistributing tasks between devices based on their dynamic resource usage. By doing so, DynaMix achieves efficient resource sharing along with dynamic plug-and-play and reconfigurability. Our example implementation on top of Android and Tizen devices shows that DynaMix enables efficient cross-device resource sharing in multi-device services.

1 Introduction

In the era of the Internet of Things, a user can access an increasing number of heterogeneous devices (e.g., smartphones, wearable devices, smart TVs) equipped with diverse, and possibly different, hardware resources (e.g., CPU, memory, camera, screen). As a result, such an environment poses the need for multi-device services which simultaneously utilize the diverse resources of the heterogeneous devices. For instance, when watching movies or viewing PDF files, a user can use a large TV screen rather than a smaller smartphone screen. Also, a user can take pictures from various angles by using multiple remote cameras. In a similar sense, a number of recent studies [33,37,38] develop and demonstrate multi-device services utilizing resources of multiple devices.

However, the existing cross-device resource sharing schemes suffer from several challenges. First, using network libraries explicitly imposes significant programming burden on developers [2, 3, 6] as they should follow a server-client model that involves careful task distribution between server and client processes. Distributed programming platform [54] may reduce the programming burden; however, they still impose the burden of efficiently partitioning an application. Second, code offloading [19, 21, 28] and remote I/O [11] can enable cross-device resource sharing without the programming burden. Unfortunately, neither of them supports all computation (e.g., CPU, memory) and I/O sharing at the same time, which limits their applicability. More importantly, the existing schemes do not optimize the placement of tasks and hence suffer when running on slow wireless networks.

Motivated by the limitations of the existing mechanisms, we need a new cross-device resource sharing mechanism achieving all of the following design goals. First, it should fully integrate the diverse resources of different devices including CPU, memory, and I/O resources. Second, it should achieve good programmability by not exposing any cross-device resource sharing details to the application layer. Third, it should dynamically redistribute tasks between devices to minimize the negative performance impacts of slow wireless networks.

In this paper, we propose **DynaMix**, a novel framework to enable **Dyna**mic **M**obile device **i**ntegration for efficient **cross**-device resource sharing. First, DynaMix fully integrates diverse resources using Distributed Shared Memory (DSM) and I/O request forwarding; DSM integrates CPU and memory, and I/O request forwarding integrates I/O resources. Second, as DSM and I/O request forwarding enable low-level re-

source sharing below the application level, DynaMix does not demand applications to be aware of multiple devices, achieving good programmability. Third, DynaMix dynamically redistributes tasks between devices in a way that minimizes inter-device communication by monitoring per-device resource usage and inter-device network usage. In addition, DynaMix supports seamless plug-and-play of remote devices by monitoring their connectivity and by taking checkpoints of an application's states.

For evaluation, we implement DynaMix on various Android and Tizen devices (e.g., Google Nexus, Samsung Smart TV). We also introduce three multi-device services to demonstrate the effectiveness of DynaMix: home theater, smart surveillance, and photo classification. The experimental results clearly show that DynaMix enables efficient cross-device resource sharing by fully integrating diverse resources and by dynamically redistributing tasks between devices. For instance, DynaMix achieves the target performance goal of home theater (i.e., 24 FPS when playing HD movies), whereas the existing mechanisms suffer from severe performance degradation (e.g., only 8.2 FPS with request forwarding).

In summary, our contributions are as follows:

- **Novel Platform.** We propose DynaMix, a novel framework to fully integrate remote resources for efficient cross-device resource sharing.
- **High Applicability.** DynaMix can easily be deployed to existing devices, and its low-level resource sharing enables easy programmability.
- **High Performance.** DynaMix minimizes the inter-device communication overheads by dynamically redistributing tasks between devices.
- **High Reliability.** DynaMix supports seamless plug-and-play of remote devices, improving the reliability of multi-device services.

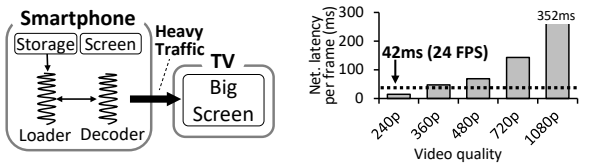
2 Background and Motivation

In the IoT environment, cross-device resource sharing is a promising solution to satisfy various service demands of users who can access an increasing number of heterogenous devices. The users can select favorable resources in different devices, so that they enjoy the same application in different ways depending on their resource configurations.

2.1 Limitations of Existing Schemes

To enable multi-device services, researchers have proposed various resource sharing schemes. We group them into three categories and compare their tradeoffs.

I/O Request Forwarding. The I/O request forwarding is a method to utilize remote I/O resources (e.g., camera, screen, audio, sensor) by forwarding I/O requests



(a) An example setup (b) Network latency

Figure 1: An example setup to play a video on a remote screen and network latency to send a single frame

to the target device, which then accesses the requested resources on behalf of the requesting device. The request forwarding schemes can forward the I/O requests in different layers (e.g., kernel, platform, user). For example, Rio [11] forwards I/O requests at the kernel level to a remote device which then performs the delivered I/O requests. M+ [43] provides cross-device functionality sharing at the platform level by forwarding IPC messages. Both schemes enable the transparent access to remote I/O resources. On the other hand, user-level [2,3,6] request forwarding schemes make programmers explicitly handle the remote I/O requests.

However, the applicability of the existing request forwarding schemes is limited as follows. First, they support only I/O resources for resource sharing[1]. Next, they require carefully-designed abstraction layers to support single-device applications. Furthermore, they can suffer from severe network overheads unless they access resources in an optimized task distribution. Figure 1a shows an example kernel-level request forwarding setup configured to use a remote screen to play a video. Since the local device forwards the decoded frame to the remote screen, it can suffer from the severe communication overhead as the video quality increases. Figure 1b shows that only the lowest resolution quality can barely meet the 24 frames per second (FPS) performance goal. Actually, moving `Decoder` task from the smartphone to the TV would greatly reduce the network overheads as only the small traffic between `Loader` and `Decoder` is exposed. From this example, we can see why the resource-aware task redistribution is important.

Code Offloading and Distributed Computation. The code offloading [19, 21] and distributed computation [28] schemes utilize remote computation resources (e.g., CPU, memory) by offloading performance-critical code regions to more powerful devices. They can not only improve the performance but also save the power consumption of the requesting device by using a faster CPU or exploiting the increased parallelism with more cores. In addition, COMET [28] implements a software-based distributed shared-memory (DSM) framework to support efficient thread offloading among devices.

However, the applicability of the existing code of-

[1]Note that M+ [43] restrictively uses CPU and memory resources for specific platform services.

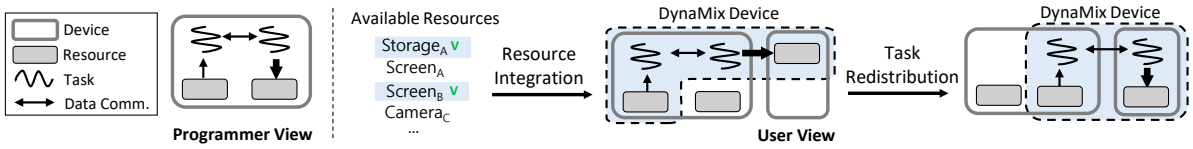


Figure 2: An example workflow of the DynaMix framework. Though programmers develop single-device applications, users can configure their services (i.e., DynaMix devices) by selecting desired resources to access remote resources.

floading schemes is limited as follows. First, they support only computation resources for cross-device sharing, which leaves I/O resources to be wasted. Next, their migration points of non-DSM schemes are restricted to specific function entries, similar to Remote Procedure Calls (RPC). Also, the migrated tasks should eventually go back to the requesting device which restricts the scope of performance-critical task redistributions.

Distributed Programming Platform. A distributed programming platform such as Sapphire [54] is similar to the distributed computation scheme but provides an interface to enable more flexible task migrations. Once the tasks are deployed by the platform-defined unit objects, the platform supports a limited form of task redistributions to reduce the performance overhead.

However, such distributed programming platform suffers from the following limitations. First of all, the resource coverage is still limited to computation resources for cross-device sharing. Next, the scheme leaves the burden of difficult multi-device programming (e.g., device-aware task partitioning, dynamic exception handling) to application developers.

2.2 Design Goals

Motivated by the limitations, we claim that an ideal resource sharing framework must satisfy the following.

High Resource Coverage. The framework should cover both I/O and computation resources for cross-device sharing. Various types of I/O resources enable the framework to provide interesting multi-device services which are infeasible in a single device alone due to its limited capabilities (e.g., device's unsupported resource types and physical location). In addition, sharing computation resources allows an application to run in a more efficient way by distributing its tasks across other devices.

Single-device Application Support. The framework should transparently support single-device applications for multi-device services. Developing multi-device applications [2, 3, 6] using a server-client model often imposes an excessive burden on developers (e.g., statically separated multiple programs). Also, this approach is practically limited toward satisfying users' various demands and developers have to manually handle dynamic behaviors. On the other hand, if the framework transparently supports a single-device application to access remote resources, developers no longer consider how remote resources are accessed. Users create their own service by selecting favorable resources and the framework provides seamless mechanisms to access them, significantly reducing the programming burden.

Resource-aware Task Redistribution. The framework should minimize the inter-device communication overhead with dynamic inter-device task redistributions. The communication overhead incurred by the remote access highly depends on dynamic factors, such as the reconfiguration of the resource sharing, the available network bandwidth, and runtime behaviors of tasks in an application. Therefore, it is important to adaptively redistribute tasks to the optimal devices to minimize the overhead.

3 DynaMix Framework

3.1 Overview

Figure 2 shows an example workflow of DynaMix framework. First, programmers develop DynaMix applications. To reduce the burden of the programmers, DynaMix requires neither any special programming concepts nor special APIs except the underlying memory consistency model described in §4.1. Therefore, programmers can write ordinary multi-threaded programs on a single device with multi-thread libraries without concerns about remote resources. This single-device programming model of DynaMix makes developing new applications and porting existing applications easy. Second, users can select desired resources (e.g., Storage_A and Screen_B in Figure 2) to execute DynaMix applications at runtime. DynaMix framework dynamically integrates the selected resources and constructs a single virtual device called a *DynaMix device*. Third, DynaMix detects the network traffic and automatically redistributes tasks across the devices to minimize the network overhead. Within the DynaMix device, tasks (i.e., threads) of the DynaMix applications can freely access remote resources or be migrated for the optimal task redistribution.

3.2 DynaMix Operations

DynaMix framework has two basic operation models: *remote resource integration* and *resource-aware task redistribution*. To support the operations, users should first make their devices DynaMix-enabled by installing two software components on each device: *resource integrator* and *thread migrator*. The resource integrator integrates both computation and I/O resources (or constructs a DynaMix device) by applying a distributed shared

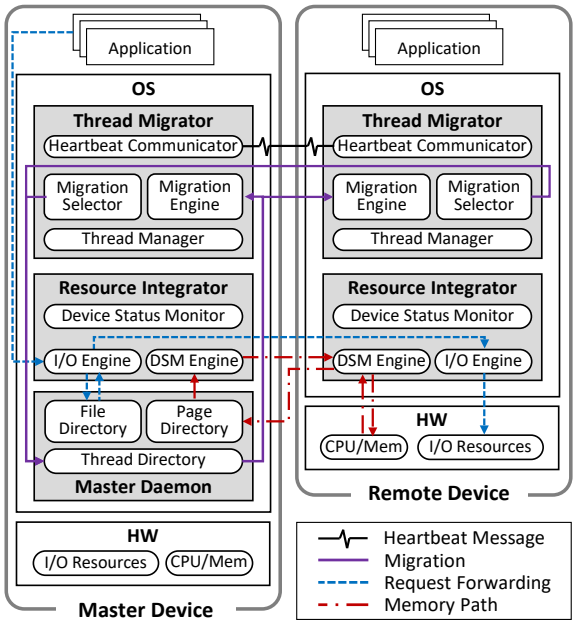


Figure 3: DynaMix architectural overview

memory (DSM) model and I/O request forwarding together (§3.2.1). The thread migrator monitors both inter-thread communication and device connectivity, and dynamically redistributes threads to their optimal locations to minimize the communication overhead (§3.2.2).

3.2.1 Remote Resource Integration

The resource integrators installed on each device collaboratively apply a DSM model and a kernel-level I/O request forwarding to integrate both computation (e.g., CPU, memory) and I/O (e.g., display, storage) resources. This mechanism enables DynaMix to satisfy two design goals of ideal resource sharing: **single-device programming model** and **high resource coverage**.

The resource integrator performs the integration in three steps. First, the resource integrator collects the information of remote resources (e.g., CPU frequency, memory size, I/O type), broadcasted by remote resource integrators, and makes the resources available for user applications. Second, if an application tries to use a remote resource, the resource integrator forwards the request to the target resource integrator. Third, the target resource integrator delivers the outcome to the application through the shared memory for computation results or through forwarding for I/O results.

3.2.2 Resource-aware Task Redistribution

With only the I/O request forwarding, DynaMix can incur severe inter-device communications. Therefore, DynaMix applies a resource-aware task redistribution mechanism by adaptively migrating threads to the optimal devices in a way to minimize the overall inter-device traffic. This mechanism satisfies the design goal of **resource-aware task redistributions**.

The resource integrator and the thread migrator work together to enable task redistributions as follows. First, the resource integrator monitors per-thread resource usage (e.g., CPU, network) to detect possible resource contentions. Second, on detecting a contention, the thread migrator compares tradeoffs of various thread allocation scenarios, and finds the best one. Third, the thread migrator migrates threads based on the scenario by delivering their execution contexts to the target devices.

4 Implementation

This section describes how we implement the aforementioned core components (*resource integrator* and *thread migrator*) and a newly introduced *master demon* component to correctly orchestrate operations. The master daemon runs on a failure-free master device on which a user launches applications. Note that we regard failures in the master device as user-intended ones such as device shutdown. Figure 3 illustrates the overall architecture.

4.1 Resource Integrator

The resource integrator consists of three components: a DSM engine, an I/O engine, and a device status monitor. The DSM and I/O engines integrate computation and I/O resources, respectively, and the device status monitor detects intra-device resource contentions.

4.1.1 DSM Engine

The DSM engine integrates the memory regions of multiple devices into a single memory space in a DSM manner. On receiving a memory access request, the DSM engine either delivers its local memory data or forwards a request to the destination DSM engine owning the data. It also works with the master daemon to orchestrate these communications for globally consistent memory management (§4.3.2). The DSM engine applies three performance optimizations as follows. First, it adopts a lazy release consistency (LRC) model [32] to safely delay memory synchronization within acquire-release block, similar to previous work [27,28]. Second, it actively performs memory prefetches on detecting sequential memory access patterns. Third, it uses a page-level coherence block to reduce the coherence overheads.

To support the page-level DSM, the DSM engine leverages a page fault handler in Linux kernel which manages page permissions. When an application enters a critical section (i.e., lock acquire), the DSM engine disables write permissions of all shareable pages in the target application. In this way, the DSM engine can detect the page modifications during a critical section. On the exit of the critical section (i.e., lock release), it recovers the write permissions. Due to the LRC model, the memory transfer of the modified pages occurs only when

another device newly acquires the same lock, which minimizes unnecessary network communications.

To further reduce the network traffic, the DSM engine transfers only the updated contents called *diffs*. On a lock release, the DSM engine generates diffs by comparing the contents of original and modified pages. When another device acquires the same lock, its DSM engine receives the corresponding diffs from the previous lock holder and applies them into the original pages.

4.1.2 I/O Engine

The I/O engine manages the access to local and remote I/O resources through kernel-level request forwarding. For the purpose, the I/O engine provides a *device file* boundary for cross-device I/O sharing similar to previous approaches [10, 11]. Mobile platforms with the Linux kernel base (e.g., Android, Tizen) use device files as their I/O abstraction layer because such files are device-agnostic. In this way, DynaMix can support a wide spectrum of I/O resources. To forward incoming requests from the host to a remote target device, the I/O engine intercepts I/O-related system calls (e.g., *open*, *read*, *write*, *ioctl*) and delivers them to the remote device with their input parameters. The remote device then performs the forwarded requests and returns the results to the host. The I/O engine also cooperates with a platform to allow users to access remote I/O resources transparently.

For example, to access audio peripherals (e.g., speaker, microphone) on a remote device, the I/O engine creates virtual device files corresponding to device files for audio peripherals (e.g., `/dev/snd/pcmCxDxx`, `/dev/snd/control`). The host I/O engine transfers requests coming through a virtual device file to the remote I/O engine which executes the requests with the corresponding original device file. Note that an audio Hardware Abstraction Layer (HAL) library (e.g., tinyalsa) is modified to access virtual device files instead of original device files. In this way, DynaMix applications can transparently access the remote audio peripherals.

Unfortunately, such kernel-level request forwarding does not directly support some I/O resources (e.g., a screen, file system) that require special management. For example, to display frame data from a frame buffer (`/dev/graphics/fb0`) in the host, the remote I/O engine should cooperate with graphics APIs in a platform to follow the existing graphics stack (i.e., SurfaceFlinger). In particular, to access a file on a remote storage, the I/O engine works with the master daemon which keeps a file directory containing the file metadata. Therefore, devices joining the DynaMix device should upload their file metadata information to the shared file directory. On receiving a file access request, the I/O engine first checks the local file directory. If the file does not exist, the I/O engine asks the master daemon to find the location in the shared file directory and forwards the request to the owner device.

4.1.3 Device Status Monitor

The device status monitor periodically collects various system information (e.g., per-thread CPU utilization, network stall time) to detect CPU and network contentions. The device status monitor is implemented as a kernel thread, which enables more accurate resource monitoring. It detects CPU contentions when CPUs are fully utilized but each thread has low CPU utilization without the existence of other bottlenecks (e.g., no I/O wait). On the other hands, it detects network contentions when the stalled time due to remote I/O accesses or memory synchronization exceeds a pre-defined threshold [2] (e.g., 30% in our environment). On detecting such contentions, the device status monitor immediately notifies the master daemon to initiate thread redistributions.

4.2 Thread Migrator

The thread migrator consists of four components: a thread manager, a migration selector, a migration engine, and a heartbeat communicator.

4.2.1 Thread Manager

The thread manager preserves various information of running threads such as execution states, resource usage, and locks. On resource contentions, the thread manager calculates threads' data communications[3] (i.e., thread-to-thread and thread-to-resource) and sends the results to the migration selector which determines the best victim for migration and its destination device. The thread manager also implements kernel-level locks to synchronize threads across different devices. Note that we modify a user-level multi-thread library (e.g., POSIX) to access these locks internally. The thread manager checks with the master daemon before allowing a thread to acquire a lock. The master daemon then forces the prior lock holder to transfer the updated memory within the acquire-release block, following the LRC model.

For reliable execution, the master thread manager keeps execution contexts of the migrated threads as a *checkpoint*, so it can consistently recover missing threads for an unintended device disconnection. After the checkpoint is created, non-migrated threads in the same application update memory pages in a copy-on-write manner to maintain original contents of shared pages. The checkpoint is updated only when the size of copied data exceeds a threshold (e.g., 20% of total memory size). As

[2] This conservative detection using the static threshold works well in DynaMix because the migration selector (§4.2.2) considers the trade-offs of all candidates and eventually decides the best migration target.

[3] DSM and I/O engine provide the information of data communications. The profiling overhead of each engine is typically insignificant because DSM engine measures the communication only in critical sections and I/O engine merely records the size of transferred data.

Algorithm 1: Migration Selector

```
input  : the analyzed data communication result, C.
input  : a set of local threads, threads_local.
input  : a set of remote devices, devices.
output : a tuple of a migration victim thread group to recommend,
         its destination device, and the network gain.
/* Construct thread groups                                    */
tgroups = [[T] for thread T in threads_local]
do
    /* Compare the amount of inter-thread comm.               */
    foreach (tg1, tg2) where tg1, tg2 ∈ tgroups do
        if communication(tg1, tg2, C) > D_thre then
            merge_groups(tg1, tg2, tgroups)
        end
    end
while tgroups changed;
/* Find a victim thread group and a destination device
   that yields the largest network gain                       */
(victim_tg, dest_dev, max_gain) = (null, null, 0)
foreach tg ∈ tgroups do
    /* Consider devices with enough idle CPU BW               */
    foreach dev ∈ possible_devices(tg, devices) do
        net_gain = estimate_net_gain(tg, dev, C)
        if net_gain > max_gain then
            (victim_tg, dest_dev, max_gain) = (tg, dev, net_gain)
        end
    end
end
return (victim_tg, dest_dev, max_gain)
```

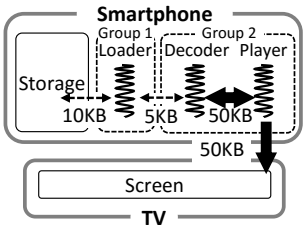
the threshold can affect memory pressure on a device, it is experimentally decided by considering an available memory size not to hurt other applications' performance.

4.2.2 Migration Selector

With the information delivered by a thread manager, the migration selector determines the best victim thread for migration and its destination device. The estimation relies on recent access patterns of an application with the assumption that similar behaviors appear in the near future. This assumption is reasonable in DynaMix's target applications which mainly access remote resources (e.g., repeatedly accessing a remote screen or camera) unless a user changes the resource configuration. The migration selector determines the best victim thread for migration and its destination device, and notifies the information to the master daemon as *migration recommendation*. Algorithm 1 describes how the migration selector finds the migration recommendation.

The migration selector first groups tightly coupled threads as a *thread group* which is a minimal migration unit. Such grouping simplifies the selection process and prevents unnecessary migration initiations. The algorithm sets threads as a thread group if their inter-thread communication amount is larger than a predefined threshold (D_{thre}). Next, it finds the best victim group and its destination device in a way to maximize the network overhead reduction, *network gain*. Note that the selected victim group is temporarily excluded in the next target selection during a specific time period to avoid frequent migration invocations on the same group. The time period is extended using exponential backoff.

The destination device should have idle CPU bandwidth enough to accommodate the migrated threads. To consider the different CPU performance of devices, the algorithm uses a scaling factor, $Perf_{dest,source}$. For example, if $Perf_{dest,source}$ is 0.9, the destination device's CPU is slower than the source's CPU by 10%.



(a) Thread grouping

Status	Value
$CPU_{\texttt{thread}}$	30%
$idle_{tv}$	70%
$Perf_{tv,phone}$	0.9
$NAcq_{\texttt{load,dec}}$	10
$lat_{tv,phone}$	0.01s
$BW_{tv,phone}$	100KB/s

(b) Device status

Thread Placement	T_{ct}	T_{dt}	T_{gain}
Group 1 on TV	-0.1 s	-0.15 s	-0.25 s
Group 2 on TV	-0.1 s	0.45 s	0.35 s

(c) Network gain estimation

Figure 4: Migration victim and destination selection

Calculating the network gain. The network gain T_{gain} quantifies how much the thread migration will improve the network performance in terms of the latency to transfer control messages (T_{ct}) and data (T_{dt}): $T_{gain} = T_{ct} + T_{dt}$.

A lock-acquire operation is the most critical source of the control messages, and each one incurs a three-hop latency (§4.2.1). The latency between thread i and j is the number of acquire operations ($NAcq_{i,j}$) times the three-hop latency between them ($lat_{D(i),D(j)}$), where $D(i)$ and $D(j)$ indicate the devices running thread i and j. Therefore, the total transfer latency is $\Sigma_{i \in tg}\Sigma_{j \in tcom} NAcq_{i,j} \times lat_{D(i),D(j)}$, where tg is the thread group and $tcom$ is a set of communicating threads. Then, the network gain of control message transfer, T_{ct}, is the latency difference due to the migration to the destination, dst:

$$T_{ct} = \sum_{i \in tg} \sum_{j \in tcom} NAcq_{i,j} \times \left(lat_{D(i),D(j)} - lat_{dst,D(j)}\right)$$

The data transfer latency gain can also be calculated in a similar manner. If $D_{i,j}$ is the size of transferred data between thread i and j, and $BW_{Di,Dj}$ is the network bandwidth between them, the data transfer latency is $D_{i,j}/BW_{Di,Dj}$. Then, the network latency gain of data transfer, T_{dt}, is the latency difference due to the migration to the destination, dst:

$$T_{dt} = \sum_{i \in tg} \sum_{j \in tcom} \left(D_{i,j}/BW_{D(i),D(j)} - D_{i,j}/BW_{dst,D(j)}\right)$$

Example Victim/Destination Selection Scenario. We illustrate example operations of the migration selector. Figure 4a shows the data communication status collected by the thread manager, and Figure 4b shows the status of devices collected by the device status monitor and the heartbeat communicator. Figure 4a also shows two thread groups, where the `Loader` and the `Decoder`

thread are allocated in the same group because they heavily communicate each other. As the TV has enough idle CPU bandwidth ($70\% \times 0.9 = 63\%$) to accommodate either thread group (30% for Group 1, $30\% \times 2 = 60\%$ for Group 2), both groups can be migrated to the TV. Next, the migration selector compares the network gains of migrating either thread group (Figure 4c), and reports the best group and destination (i.e., Group 2 and TV) to the master daemon.

4.2.3 Migration Engine

After the migration selector decides the victim threads (i.e., thread group) and its destination device, the migration engine eventually performs a thread migration. DynaMix supports a low-overhead migration by adopting thread cloning and live migration. It minimizes the downtime, a suspended time period during migration, by transferring essential pages in a short time.

First, the source device sends only the memory layout (e.g., heap, stack) of the victim threads to the destination device which then creates their clone threads suspended during the migration. Next, for a short period (e.g., 2 secs), the migration engine transfers the most recently accessed pages (e.g., using the LRU-based page cache in Linux kernel) to the destination device, while the victim threads run on the original device. Using write permission faults (similar to §4.1.1), the migration engine detects and records the updated pages during the memory transfer. After finishing (i.e., timeout) the memory transfer, it sends the victim thread's execution context (e.g., process control blocks) with the updated pages in the meanwhile. This transparent live migration (similar to [20]) effectively hides the migration latency and minimizes the service downtime. Finally, the clone threads continue their execution on the destination device after the victim threads are suspended on the original device.

4.2.4 Heartbeat Communicator

For dynamic resource integration, DynaMix supports seamless operations whiles devices are plugged in and out. The heartbeat communicators periodically exchange heartbeat messages to check the device connectivity and share their resource status (e.g., CPU idleness, network latency, bandwidth). The resource status information is then delivered to the migration selector. Note that the inter-device network latency can be estimated from the round-trip latency of heartbeat messages.

The heartbeat communicator can detect which remote device joins or leaves a DynaMix device. For a newly joined device, its heartbeat communicator broadcasts heartbeat messages. On receiving the message, the master daemon enlists the new device in the DynaMix device. The heartbeat communicator also detects unstable devices by monitoring the connectivity (e.g., the number of packet drops). If a device becomes unstable, the heartbeat communicator notifies the master daemon to initiate migrating the threads in the device to more stable devices to avoid thread recovery that may cause the loss of the overall progress. For an unexpected disconnection. the master heartbeat communicator notifies the thread manager to recover from the latest checkpoint (§4.2.1).

4.3 Master Daemon

A DynaMix device has a single master daemon[4] that manages various system states (e.g., threads, locks, memory pages, files) to orchestrate DynaMix operations and components. The master daemon runs on the failure-free master device, and consists of three components: a thread directory, a page directory, and a file directory.

4.3.1 Thread Directory

The thread directory manages the global states of threads such as thread locations, and arbitrates the thread migration process. It collects resource contention signals from the device status monitors, and migration recommendations from the migration selectors. On receiving recommendations, the thread directory selects the best migration victim and its destination to achieve the highest network gain, and then manages the migration engines to perform the designed migrations.

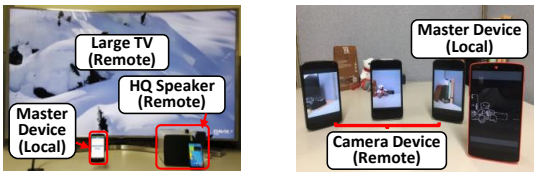
The thread directory also keeps the lock information (e.g., current owner, status). To acquire a lock, each device should consult the master device's thread directory. To reduce the lock acquisition overhead, the thread directory can speculatively grant the lock to frequent lock holding devices. When another device attempts to acquire the lock, the thread directory reclaims the speculatively given lock. Note that when a device is disconnected, the thread directory immediately reclaims all locks held by the device to avoid a deadlock.

4.3.2 Page Directory

The page directory manages the sharing state of memory pages to orchestrate memory synchronization operations. When a device sends a remote read request due to a page fault, the page directory consults a sharer table which keeps the sharer device lists of each page. It then relays the request to one of the sharer devices which will deliver the page to the requesting device.

On a lock release, the lock owner device reports the address list of updated pages to the page directory. In this way, the page directory identifies which pages should be sent to the next owner. When another device acquires the lock, the page directory manages its prior owner to forward the updated pages or their diffs if the new owner has old copies. Note that the transfers of shared pages

[4] Such a centralized approach enables easy management but might limit scalability. We believe that composing multiple DynaMix devices rather than a single large one is much preferable in our scenarios.



(a) Home Theater (b) Smart Surveillance

Figure 5: Two example DynaMix applications

mostly occur when a device newly acquires a lock due to LRC memory model.

4.3.3 File Directory

The file directory manages the file metadata and the physical locations of shared files for the globally consistent file view. Whenever a new device joins the DynaMix device or a device updates the metadata, the master daemon updates its file directory and then notifies the updated information to other devices. Note that except the device which owns a file, each device holds the file's read-only copy in memory.

5 Evaluation

5.1 Experimental Setup

We implemented our example DynaMix prototype which can be easily installed on top of existing Android and Tizen devices. For our evaluation, we installed DynaMix on Google Nexus smartphones (i.e., Nexus 4 and 5) and an in-house Samsung Smart TV. The smartphones run Android 5.1.1 (CyanogenMod 12.1) with Linux kernel version 3.4 patch, and the Samsung Smart TV runs Tizen 2.3 with Linux kernel version 3.0 patch. All devices are connected to the same Wi-Fi network (IEEE 802.11ac) with the maximum bandwidth of 100Mbps.

To evaluate the DynaMix prototype, we introduce three example multi-device use cases (i.e., home theater, smart surveillance, and photo classification) designed to utilize both computation and I/O resources simultaneously. We believe users can easily make other interesting services using our framework.

Home Theater. The home theater is a typical multi-threaded movie player application which loads and decodes a movie file from a storage, shows the video on a screen, and plays the audio through a speaker. DynaMix allows users to configure resources (e.g., a large TV, an HQ speaker) used to run the home theater. Figure 5a shows the example home theater setup with three devices. We used FFmpeg [4] to decode video and audio frames. Here, the home theater plays a movie with both video and audio frames synchronized.

Smart Surveillance. The smart surveillance is another possible service that performs image processing (e.g., edge detection) with preview images from a remote camera. Figure 5b shows the example smart surveillance

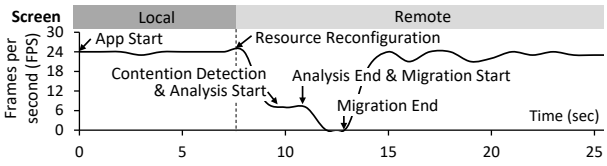


Figure 6: Perf. timeline of the home theater application

setup using four devices. A processing thread performs edge detection on preview images from a selected camera device, and a UI thread displays the processed image on the screen. We used the Canny edge detection algorithm [16] to detect moving objects in the service.

Photo Classification. Lastly, integrating storage resources enables a shared storage system across devices where users can observe scattered remote files (e.g., photos, videos) in the same hierarchy and easily access them at any device. To evaluate the storage system, we perform object classifications for photos scattered in the connected devices. For the purpose, we used an object classifier with the pre-trained CNN model (SSD_MobileNet [31]) using a TensorFlow [9] library. Each thread classifies its assigned photos with the classifier and reports the results to the collector thread.

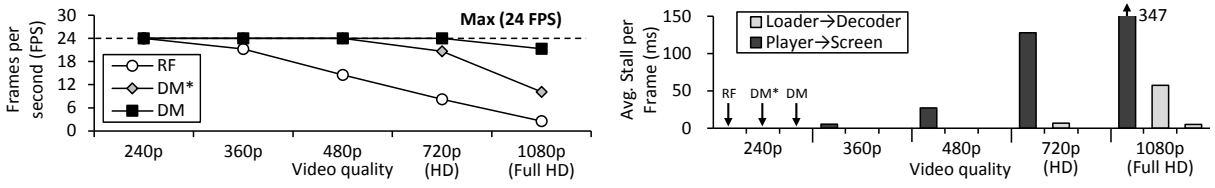
5.2 Operation Models

This evaluation revisits the basic operation models of DynaMix in §3.2. We use the multi-threaded home theater with loader, decoder, and player threads. It runs on the DynaMix device (Figure 5a) configured with a Samsung Smart TV as the remote screen, an HQ speaker attached to Nexus 4 as the remote audio, and a Nexus 5 smartphone as the master device. We play an HD (720p) movie and measure its frames per second (FPS).

Figure 6 shows its performance timeline. When the user initially plays a movie on the master device, the home theater displays video frames on the local smartphone screen with the target performance of 24 FPS. However, after the user suddenly switches the screen device to the remote TV (8 sec), it suffers from significant FPS drops due to the huge network traffic caused by forwarding HD video frames to the TV. Therefore, DynaMix immediately detects a network contention (10 sec), decides the best task redistribution plan (11 sec), and migrates the video decoder and player threads to the TV (13 sec). Although the performance temporarily degrades due to the increased network consumption caused by the migration, DynaMix quickly restores the target performance (14 sec) with a negligible service downtime. This experiment verifies that DynaMix enhances the service quality with resource-aware task redistribution even in the sudden resource reconfiguration.

5.3 Service Quality

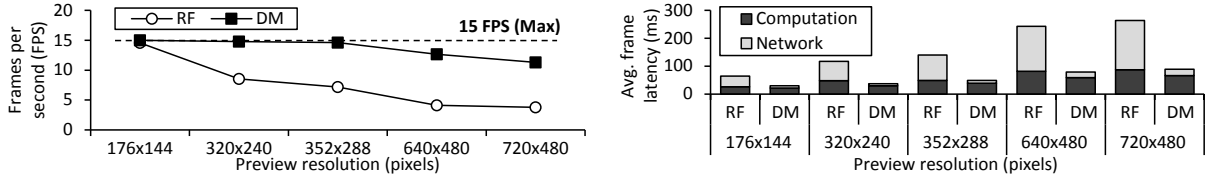
We now evaluate three use cases (i.e., home theater, smart surveillance, and photo classification) to verify that



(a) Throughput (FPS)

(b) Frame-average network stall time

Figure 7: The home theater performance for Request Forwarding (RF) and DynaMix (DM) (*: no prefetching)



(a) Throughput (FPS)

(b) Frame-average detection latency breakdown

Figure 8: The smart surveillance performance for Request Forwarding (RF) and DynaMix (DM)

DynaMix significantly enhances the service quality. To show the benefit of resource-aware task redistribution, we compare DynaMix with Request Forwarding (RF) as a representative baseline because its resource sharing mechanism conceptually includes the state-of-the-art work (e.g., Rio [11]), which accesses a remote resource and receives the result via a wireless network. We also evaluate DynaMix without memory prefetching.

Home Theater. We configure a DynaMix device as explained in §5.2, and measure the throughput (i.e., frames per second) of the home theater on the DynaMix device. RF forwards decoded frames to the remote device because all threads run on the master device.

Figure 7a compares the throughput (FPS) of the home theater for RF and DynaMix. While RF suffers from increasing throughput degradations with the target video quality improved, DynaMix successfully achieves the target throughput up to the decent quality (480p) even without memory prefetching. Enabling the prefetching further enhances the throughput, which makes DynaMix achieve the throughput close to the maximum for the full HD quality (1080p). DynaMix achieves 8.3x higher throughput than RF, while paying only 11% performance drop from the maximum throughput for 1080p.

Figure 7b compares the network stall time of three design points to process a video frame for the various video qualities. The network stall time means how much network traffic affects the per-frame latency, and helps to clearly investigate why RF suffers from the low throughput. First, RF incurs severe network traffic to transfer a decoded frame between the player thread and the remote TV screen even for a relatively inferior quality (360p). Moreover, RF suffers from a huge amount of network stall as the video quality increases. On the other hand, as DynaMix can migrate the video decoder and player threads to the TV, the loader thread on the master device can timely transfer small-sized encoded frames to the decoder threads on the TV. As a result, DynaMix effectively hides the network stall up to 480p, and applying the memory prefetching further amortizes the network overheads (i.e., near-zero network stall for 1080p).

Smart Surveillance. We configure a DynaMix device to use a Nexus 5 device as a master device with a screen, and three Nexus 4 devices as remote cameras, as shown in Figure 5b. As this application allows a user to select a target remote camera, we randomly choose one camera as the current input feeder. RF receives preview images from the remote camera because a processing thread runs on the master device. We now assume that DynaMix is equipped with memory prefetching by default.

Figure 8a compares the throughput (FPS) of the smart surveillance for RF and DynaMix. While RF suffers from significant throughput degradation with the preview resolution increased, DynaMix retains moderate performance drop as only 24.7% compared with the target throughput for the highest preview resolution (720x480). Figure 8b shows the breakdown of the average latency in detecting edges of a preview image. After analyzing the tradeoff, DynaMix migrates the edge-detector threads to the camera devices to avoid the network contention. As a result, it achieves far less network latency than forwarding raw preview images from the camera to the master device. Note that the computation still occupies a significant portion of the total latency due to the lack of sufficient computation resources. This result suggests deploying faster CPUs on remote cameras so that DynaMix can completely remove the computation overhead.

Photo Classification. We configure a DynaMix device to use up to four Nexus 5 smartphones to construct the shared storage system. Each device has 100MB of photos with different sizes ranging from 4KB to 10MB. A user can choose the number of classifier threads and

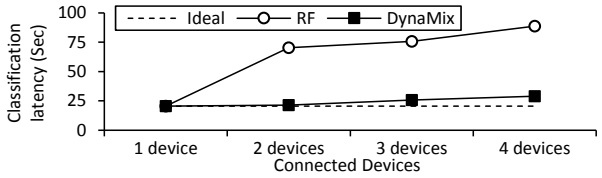


Figure 9: The total latency of the photo classification for all photo files on the shared storage system

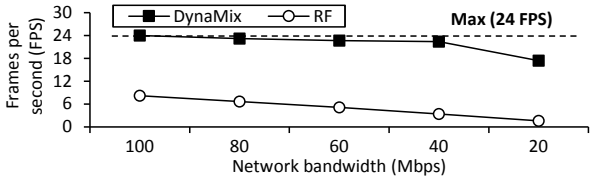


Figure 10: The home theater performance on various network bandwidth to play an HD video

	Remote File Size (B)				
	<10K	<100K	<1M	<5M	≤10M
RF (%)	87.8	64.4	33.3	13.9	0
Mig. (%)	12.2	35.6	66.7	86.1	100

Table 1: DynaMix's request forwarding (RF) vs. migration (Mig.) ratio on the photo classification

	Average Power (mW)		
	Master Device	Screen Device	Total
RF	4985.90	5151.37	10137.27
DynaMix	2956.55	6480.51	9427.06

Table 2: The power consumption of the home theater for Request Forwarding (RF) and DynaMix

would launch threads in proportion to the total size of photos. In our four-device configuration, we assume that four threads classify total 400MB of photos.

We then measure the latency to perform the classification for all photos, and compare the performance of RF and DynaMix. We also mark the ideal performance to identify the bottleneck. We assume that the ideal one classifies all files on the remote devices without any network overheads. Note that RF forwards remote files to the threads running on the master device.

Figure 9 compares the performance of the total classification latency. As the number of connected devices increases, RF suffers from high latency due to the increasing network overheads incurred by forwarding remote files to the classifiers. On the other hand, DynaMix is barely affected by the network overheads and thus achieves the latency close to the ideal one, even for the four-device configuration. It is because DynaMix dynamically redistributes the threads across devices to minimizes the network overheads.

Furthermore, DynaMix can dynamically use either of request forwarding and the adaptive task migration, based on their tradeoffs. Note that the migrated threads should use request forwarding during a certain time period to prevent frequent migrations (§4.2.2). To emphasize the point, Table 1 shows the percentage of the two cases in the four-device configuration. DynaMix is likely to use the request forwarding more for small files (i.e., <100KB) to avoid the migration overhead, whereas it is likely to migrate threads for large-sized files (i.e., >1MB) to avoid the transfer overhead.

5.4 Network Sensitivity

To identify the performance impact for a given network bandwidth, we measure the performance of the home theater by playing an HD movie on RF and DynaMix. For this experiment, we vary the available bandwidth with Linux `tc` utility, and measure the average FPS as the performance metric.

Figure 10 shows that RF severely suffers from its performance drops even with the maximum network bandwidth available (100Mbps) and further as the network bandwidth decreases. On the other hand, DynaMix maintains the target 24 FPS with only 40Mbps of bandwidth available. This result indicates that DynaMix effectively minimizes the network overhead by adaptively redistributing tasks among devices.

5.5 Power Consumption

In this experiment, we measure the power consumption of DynaMix while playing an HD movie clip, and compare it against RF. To measure the impact of inter-device traffic reduction, we use two Nexus 5 smartphones as a master device and a screen device. We use Monsoon power monitor [5] to measure the power consumption.

Table 2 measures the power consumption of the devices. First, the master device consumes much less power with DynaMix than RF by migrating a rendering task to the remote device and thus reducing the network traffic. On the other hand, the screen device consumes little more power with DynaMix than RF by running a relocated rendering task. As a result, DynaMix reduces the total power by 7% mainly due to the reduced network overhead. More importantly, as DynaMix's service quality (or performance) is 3-4 times higher than RF (Figure 10) and their power consumptions are similar (Table 2), DynaMix's overall energy efficiency can be considered 3-4 times higher for the target service quality. For further energy reduction, DynaMix may redistribute tasks in a way to maximize the energy efficiency.

6 Discussion

Heterogeneous ISA/OS. One interesting issue related to our work is to extend the coverage of architecture and operating system used by DynaMix devices. However, it is a well-known challenge to seamlessly share resources in heterogeneous devices using different ISAs and OSes. Therefore, existing work often assume either homogeneous ISA/OS [11, 43] or expensive VM supports to emulate the homogeneous platform [19, 21, 28].

In this work, DynaMix also assumes homogeneous ISA/OS environments for the most popular mobile platform (i.e., Andriod/ARM). However, we showed that its OS coverage can be easily extended to Tizen which shares the Linux kernel base. To improve the platform coverage further, we believe that the following directions seem to be promising. For non-Linux based OSes (e.g., iOS), DynaMix may implement a compatibility layer between kernel and application by taking approaches similar to existing OS-compatibility schemes [8, 12]. To support cross-ISA (e.g., ARM to x86) migrations, DynaMix may implement a native offloading using the compiler-assisted method [39] or dynamic binary translation [51].

Developing DynaMix Applications. To fully utilize DynaMix's resource-aware task redistribution, programmers are recommended to compose applications with multiple threads, specialized in certain computing jobs or I/O resource accesses. We believe this guideline is not burdensome to programmers, as many recent programming conventions also recommend similar guidelines for optimal performance [1, 7]. To further accommodate easy application development, DynaMix may adopt existing automatic code parallelization techniques [34, 40, 49] to maximize the effectiveness of resource-aware task redistribution without additional programming effort.

Security Concerns. Another assumption of this work is that a user shares resources in only user-owned trusted devices, as existing schemes such as task offloading [19, 21, 28] and remote IO forwarding [11]. In fact, resource sharing with untrusted devices is not common scenarios that DynaMix considers. Therefore, the security issues related to untrusted devices are beyond the scope of our work. However, we believe that DynaMix can resolve such security issues by adopting existing secure task-offloading schemes [26, 42, 45, 47], without a noticeable increase in complexity.

7 Related Work

Cross-device Resource Sharing. Single system image (SSI) [17, 18, 25] is traditional work to integrate resources by creating one single system with a cluster of machines connected to a fast and stable wired network. However, its complex operations and huge synchronization overheads are not suitable to the mobile environment with limited communication capabilities. Thus, similar studies in mobile computing have focused on how to selectively integrate remote resources. For example, offloading schemes [19, 21, 28] offload compute-intensive tasks to powerful servers, even for heterogeneous ISAs [39, 51]. Solutions to utilize other resources such as GPU [22], screen [14], storage [23, 44, 46], generic I/O resources [11] and platform-level services [43] have also been proposed. While they only support specific types of resources, DynaMix integrates a wide spectrum of computation and I/O resources. On the other hand, some studies [50, 53] have optimized spectrum utilization sharing in cellular networks. Such techniques are orthogonal to our work but we can adopt them to more efficiently communicate between devices.

Multi-device Programming Platform. To facilitate easy application development in the multi-device environment, [54] allows programmers to develop unit objects and automatically deploys them across devices. [27] also provides a DSM platform and APIs for multi-device applications. Such platforms, however, still force programmers to explicitly partition applications with special APIs. [24] provides a control interface to access various home appliances with unified APIs, but it does not distribute tasks for efficient resource utilization. DynaMix, on the other hand, enables task redistributions of single-device applications for multi-device services, without explicit application partitioning.

Thread Migration. The thread migration is a widely supported feature in distributed computing platforms [13, 41, 52, 55]. Especially, to reduce a service downtime during migration, various VM platforms [15, 29, 36] have implemented the pre-copy [20] or the post-copy [30, 35] live migrations, depending on the timing to send execution contexts. DynaMix also applies such live migration schemes to our environment. Researchers have proposed an online thread distribution algorithm [48] to minimize inter-thread network overheads. However, DynaMix resolves CPU contention as well as network contention, optimized to the mobile environment.

8 Conclusion

In the era of the Internet of Things, a user can access an increasing number of heterogeneous devices. We proposed DynaMix, a novel framework to enable efficient cross-device resource sharing by integrating diverse resources and dynamically redistribute tasks. Our example implementation on the top of Android and Tizen devices showed that DynaMix can efficiently support multi-device services using single-device applications.

Acknowledgment

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2015M3C4A7065647, NRF-2017R1A2B3011038), and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R0190-15-2012). We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-University Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] Android developer training - sending operations to multiple threads. https://developer.android.com/training/multiple-threads/index.html.

[2] Android IP Webcam. https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en.

[3] Android Wi-Fi Speaker. https://play.google.com/store/apps/details?id=pixelface.android.audio&hl=en.

[4] FFmpeg. https://ffmpeg.org/.

[5] Monsoon Solutions Inc. Monsoon Power Monitor. http://www.msoon.com/.

[6] Nest cam. https://www.nest.com/camera/meet-nest-cam/.

[7] Tizen development guide - using threads. https://developer.tizen.org/development/guides/native-application/user-interface/efl/core-loop-and-os-interfacing/using-threads?langredirect=1.

[8] Wine. https://www.winehq.org/.

[9] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] AMIRI SANI, A., BOOS, K., QIN, S., AND ZHONG, L. I/o paravirtualization at the device file boundary. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[11] AMIRI SANI, A., BOOS, K., YUN, M. H., AND ZHONG, L. Rio: A system solution for sharing I/O between mobile systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (2014).

[12] ANDRUS, J., HOF, A. V., ALDUAIJ, N., DALL, C., VIENNOT, N., AND NIEH, J. Cider: Native execution of iOS apps on Android. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014).

[13] ARIDOR, Y., FACTOR, M., AND TEPERMAN, A. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing* (1999).

[14] BARATTO, R. A., KIM, L. N., AND NIEH, J. THINC: A virtual display architecture for thin-client computing. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (2005).

[15] 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. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003).

[16] CANNY, J. A computational approach to edge detection. *IEEE Transactions On Pattern Analysis and Machine intelligence 8*, 6 (1986).

[17] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995).

[18] CHERITON, D. The v distributed system. *Commun. ACM 31*, 3 (1988).

[19] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th Conference on Computer Systems* (2011).

[20] CLARK, C., FRASER, K., HAND, S., AND HANSEN, J. G. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation* (2005).

[21] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010).

[22] CUERVO, E., WOLMAN, A., COX, L. P., LEBECK, K., RAZEEN, A., SAROIU, S., AND MUSUVATHI, M. Kahawai: High-quality mobile gaming using GPU offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015).

[23] DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANA, A., YALAGANDULA, P., AND ZHENG, J. PRACTI replication. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation* (2006).

[24] DIXON, C., MAHAJAN, R., AGARWAL, S., BRUSH, A., LEE, B., SAROIU, S., AND BAHL, P. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012).

[25] FEELEY, M. J., MORGAN, W. E., PIGHIN, E. P., KARLIN, A. R., LEVY, H. M., AND THEKKATH, C. A. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995).

[26] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010).

[27] GAO, J., SIVARAMAN, A., AGARWAL, N., LI, H., AND PEH, L.-S. DIPLOMA: Consistent and coherent shared memory over mobile phones. In *Proceedings of the 30th International Conference on Computer Design* (2012).

[28] GORDON, M. S., JAMSHIDI, D. A., MAHLKE, S., MAO, Z. M., AND CHEN, X. COMET: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012).

[29] GULATI, A., SHANMUGANATHAN, G., HOLLER, A., WALDSPURGER, C., JI, M., AND ZHU, X. VMware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal* (2012).

[30] HINES, M. R., AND GOPALAN, K. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2009).

[31] HUANG, J., RATHOD, V., SUN, C., ZHU, M., KORATTIKARA, A., FATHI, A., FISCHER, I., WOJNA, Z., SONG, Y., GUADARRAMA, S., AND MURPHY, K. Speed/accuracy trade-offs for modern convolutional object detectors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

[32] KELEHER, P., COX, A. L., AND ZWAENEPOEL, W. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (1992).

[33] KIM, B., HEO, S., LEE, G., PARK, S., KIM, H., AND KIM, J. Heterogeneous Distributed Shared Memory for Lightweight Internet of Things Devices. *IEEE Micro 36*, 6 (2016).

[34] KIM, H., JOHNSON, N. P., LEE, J. W., MAHLKE, S. A., AND AUGUST, D. I. Automatic speculative doall for clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (2012).

[35] KIM, J., CHAE, D., KIM, J., AND KIM, J. Guide-copy: Fast and silent migration of virtual machine for datacenters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013).

[36] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: The Linux virtual machine monitor. In *Proceedings of The Ottawa Linux Symposium* (2007).

[37] LEE, G., HEO, S., KIM, B., KIM, J., AND KIM, H. Integrated IoT Programming with Selective Abstraction. In *Proc. 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)* (2017).

[38] LEE, G., HEO, S., KIM, B., KIM, J., AND KIM, H. Rapid prototyping of IoT applications with Esperanto compiler. In *Proc. 28th International Symposium on Rapid System Prototyping (RSP)* (2017).

[39] LEE, G., PARK, H., HEO, S., CHANG, K.-A., LEE, H., AND KIM, H. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015).

[40] LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2006).

[41] MA, M. J., WANG, C.-L., AND LAU, F. C. Delta Execution: A preemptive Java thread migration mechanism. *Cluster Computing 3*, 2 (2000), 83–94.

[42] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010).

[43] OH, S., YOO, H., JEONG, D. R., BUI, D. H., AND SHIN, I. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (2017).

[44] PERKINS, D., AGRAWAL, N., ARANYA, A., YU, C., GO, Y., MADHYASTHA, H. V., AND UNGUREANU, C. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the 10th European Conference on Computer Systems* (2015).

[45] PORTOKALIDIS, G., HOMBURG, P., ANAGNOSTAKIS, K., AND BOS, H. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010).

[46] RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009).

[47] SHRAER, A., CACHIN, C., CIDON, A., KEIDAR, I., MICHALEVSKY, Y., AND SHAKET, D. Venus: Verification for untrusted cloud storage. In *Proceedings of the ACM Workshop on Cloud Computing Security Workshop* (2010).

[48] THITIKAMOL, K., AND KELENHER, P. Thread migration and communication minimization in DSM systems. *Proceedings of The IEEE 87*, 3 (1999), 487–497.

[49] VACHHARAJANI, N., RANGAN, R., RAMAN, E., BRIDGES, M. J., OTTONI, G., AND AUGUST, D. I. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (2007).

[50] WANG, J., ZHU, D., ZHAO, C., LI, J. C., AND LEI, M. Resource sharing of underlaying device-to-device and uplink cellular communications. *IEEE Communications Letters 17*, 6 (2013), 1148–1151.

[51] WANG, W., YEW, P.-C., ZHAI, A., MCCAMANT, S., WU, Y., AND BOBBA, J. Enabling cross-isa offloading for cots binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (2017).

[52] WOOD, T., SHENOY, P., VENKATARAMANI, A., AND YOUSIF, M. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (2007).

[53] YU, C.-H., DOPPLER, K., RIBEIRO, C. B., AND TIRKKONEN, O. Resource sharing optimization for device-to-device communication underlaying cellular networks. *IEEE Transactions on Wireless communications 10*, 8 (2011), 2752–2763.

[54] ZHANG, I., SZEKERES, A., VAN AKEN, D., ACKERMAN, I., GRIBBLE, S. D., KRISHNAMURTHY, A., AND LEVY, H. M. Customizable and extensible deployment for mobile/cloud applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014).

[55] ZHU, W., WANG, C.-L., AND LAU, F. C. M. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *Proceedings of the IEEE International Conference on Cluster Computing* (2002).