creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Doctoral Dissertation

Dynamic Remote Resource Utilization Mechanisms on Mobile Computing

Dongju Chae (채 동 주)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2019

Collection @ postech

모바일 컴퓨팅을 위한 동적 원격 자원 활용 기술

Dynamic Remote Resource Utilization Mechanisms on Mobile Computing

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Dynamic Remote Resource Utilization Mechanisms on Mobile Computing

by

Dongju Chae
Department of Computer Science and Engineering
Pohang University of Science and Technology

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Computer Science and Engineering

Pohang, Korea
12. 14. 2018
Approved by
Jong Kim (Signature)
Academic advisor

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Dynamic Remote Resource Utilization Mechanisms on Mobile Computing

Dongju Chae

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

12. 14. 2018

Committee Chair	Jong Kim	(Seal)
Member	Jangwoo Kim	(Seal)
Member	Jae-Joon Kim	(Seal)
Member	Gwangsun Kim	(Seal)
Member	Jae Wook Lee	(Seal)

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

DCSE 채 동 주. Dongju Chae
20120766
Dynamic Remote Resource Utilization Mechanisms on Mobile Computing,
모바일 컴퓨팅을 위한 동적 원격 자원 활용 기술
Department of Computer Science and Engineering , 2019, 113p, Advisor : Jong Kim. Text in English.

ABSTRACT

In the era of the Internet of Things (IoT), mobile computing is no longer limited to local resources. The rapid progress of mobile wireless networks such as WiFi and LTE enables mobile users to easily access various remote resources (e.g., CPU, DRAM, peripheral devices) in different devices (e.g., desktop, cloud server, smart TV). As a result, such an environment has enabled a lot of interesting services utilizing remote resources dynamically, which is infeasible in a single device alone due to its resource constraints. For example, one mobile user can offload compute-intensive tasks (e.g., face recognition) to cloud servers to improve performance and battery life. Also, another user can simultaneously utilize remote I/O resources (e.g., screen, camera) located in different mobile devices to construct a user-customized service.

However, existing system software on mobile devices has a limited capability to support such services. Although some solutions can provide the functionality to

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

access remote resources, they have suffered from severe performance degradation and inefficient resource utilization. Moreover, as mobile devices access remote resources using a slow and unreliable wireless network, naïve system software design often exposes the problems more seriously.

In this thesis, we propose the mobile computing framework to efficiently and dynamically utilize remote resources from possibly different devices and servers. The key idea is to identify performance-critical resources and possible performance bottlenecks in advance, and to timely redistribute tasks (or resources) across devices and servers. To achieve this, we leverage three different techniques to utilize cloud resources and mobile resources. First, we maximize the throughput of computation offloading in cloud servers. We estimate post-offload performance and perform load balancing for offloaded tasks while minimizing resource contentions. Second, we increase the duration of application caching in a mobile device. We extend a device's memory capacity and timely prepare essential memory pages using the cloud-assisted memory swap mechanism. Lastly, we improve the performance of I/O sharing between devices for multi-device services. We seamlessly integrate multiple resources in different devices and adaptively redistribute tasks between devices. The evaluation results show that our framework outperforms conventional software solutions regarding resource efficiency and performance.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Contents

I. Introduction **1**

II. Remote Resource Utilization on Mobile Computing **5**

2.1 Utilizing Cloud Resources 5

2.2 Cross-device Resource Sharing 6

2.3 Three Mechanisms to Efficiently Utilize Remote Resources 7

III. Cloud Platform for Cost-Effective Offloading of Mobile Applications **9**

3.1 Existing Schemes and Limitations 9

3.1.1 Basic Offload Mechanisms 9

3.1.2 Limitations of Existing schemes 10

3.2 CMcloud offloading 12

3.2.1 Operation Model 13

3.2.2 Architecture Model 15

3.3 CMcloud mechanism 17

3.3.1 Reference-model Server Profiling 17

3.3.2 Performance Estimation 19

3.3.3 Performance Monitoring 24

3.3.4 Migration 26

3.3.5 Cost-aware Application Scheduling in Cloud 27

3.4 Evaluation 27

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

3.4.1 Estimating Target CPU Performance 29
3.4.2 Cost Effectiveness with QoS requirements 31
3.4.3 Offloading/migration Overhead 34

IV. A Cloud-Assisted Swap Mechanism for Mobile Devices 36

4.1 Existing Schemes and Limitations 36
4.1.1 Application Caching 36
4.1.2 Application Termination 38
4.1.3 Possible Swap Mechanisms on Mobile Devices 39
4.2 CloudSwap 40
4.2.1 Baseline Remote Swap-In/Out Mechanism 41
4.2.2 Swap-In using App-aware Read-Ahead (ARA) 43
4.2.3 Utilizing Local Storage with Hybrid Swap 44
4.2.4 Remote Swap-In using Pattern-Aware Prefetch (PAP) . . . 47
4.2.5 Assumptions for Design and Operation 49
4.3 Implementation 50
4.3.1 Mobile Devices 50
4.3.2 Cloud Servers 53
4.4 Evaluation 55
4.4.1 Experimental Setup 55
4.4.2 Application Response Latency 56
4.4.3 Flash Storage Lifespan 59
4.4.4 Prefetch Accuracy and Overhead of PAP 60
4.4.5 Network Bandwidth Consumption 61

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

4.4.6 Power Consumption Overhead 62

V. Dynamic and Efficient Cross-device Resource Sharing 64

5.1 Existing Schemes and Limitations 64

5.1.1 Limitations of Existing Schemes 64

5.1.2 Design Goals 67

5.2 DynaMix Framework 69

5.2.1 Overview 69

5.2.2 DynaMix Operations 69

5.3 Implementation 71

5.3.1 Resource Integrator 72

5.3.2 Thread Migrator 76

5.3.3 Master Daemon 82

5.4 Evaluation 84

5.4.1 Experimental Setup 84

5.4.2 Operation Models 86

5.4.3 Service Quality 87

5.4.4 Network Sensitivity 92

5.4.5 Power Consumption 92

VI. Related Work 94

VII. Conclusion 98

Summary (in Korean) 100

References 102

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

List of Tables

3.1 CPUs used for tests 28

3.2 Workloads (in i7-2600) 29

3.3 CMcloud/Offloading overheads 34

4.1 Parameters of mobile devices and the cloud 55

4.2 Target workloads and their user behaviors 56

4.3 Prefetch accuracy and overhead of PAP 60

5.1 DynaMix's request forwarding (RF) vs. migration (Mig.) ratio on the photo classification 92

5.2 The power consumption of the home theater for Request Forwarding (RF) and DynaMix 93

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

List of Figures

2.1 Example code offloading 5

2.2 Example cross-device resource sharing (movie player) 6

3.1 Limitation of the existing mobile-cloud offloading schemes 10

3.2 CMcloud's example operation model 13

3.3 CMcloud's basic architecture model 16

3.4 CMcloud's performance estimation process using architecture performance modeling 19

3.5 An example reuse distance of four for A 22

3.6 Performance monitoring 25

3.7 Accuracy of the performance prediction 30

3.8 Datacenter throughput (out of 500 requests) 31

3.9 Datacenter utilization (out of 16 sockets) 33

3.10 Per-socket cost effectiveness 33

4.1 Applications killed by Low Memory Killer (LMK) [26] in an example multitasking scenario on an Android device 37

4.2 Possible swap mechanisms for mobile devices 39

4.3 Limitations of the conventional swap mechanisms 41

4.4 CloudSwap system architecture 42

4.5 Exploitation of locality with read-ahead 43

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

4.6 Swap-in frequencies of the swap-out pages 44
4.7 Delta sizes of frequently swap-in pages 46
4.8 Sources of swap-out pages with Hybrid Swap 46
4.9 Similar swap-in patterns of two different devices 47
4.10 Null-frame encoding of two given pages 48
4.11 Modifications to swap information 51
4.12 Application response latencies of Local Swap (Local), Remote Swap (Remote), and CloudSwap 57
4.13 Breakdown of the sources of swap-in pages 58
4.14 The amount of write I/O traffic and the expected storage lifespan of LMK, CloudSwap, and Local Swap 59
4.15 App response latencies of PAP and MRS with various user-behavior prediction accuracies 61
4.16 Reductions in the swap-out page sizes 61
4.17 Power consumption overheads of Local Swap, Remote Swap, and CloudSwap 63

5.1 An example setup to play a video on a remote screen and network latency to send a single frame 65
5.2 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. 66
5.3 DynaMix architectural overview 72

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

5.4 Migration victim and destination selection 80
5.5 Two example DynaMix applications 84
5.6 Perf. timeline of the home theater application 86
5.7 The home theater performance for Request Forwarding (RF) and DynaMix (DM) (*: no prefetching) 88
5.8 The smart surveillance performance for Request Forwarding (RF) and DynaMix (DM) 89
5.9 The total latency of the photo classification for all photo files on the shared storage system 91
5.10 The home theater performance on various network bandwidth to play an HD video 93

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

I. Introduction

In the IoT environment, mobile users now expect more interesting and better experiences in mobile computing beyond traditional use-cases which consider only a user-carrying mobile device. Cloud servers have been widely employed to overcome the limited resource capability of a mobile device. Also, users 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). The technological advances in mobile networks such as WiFi and LTE have accelerated such demands in mobile computing.

For example, recent advances towards mobile cloud computing propose a selective mobile-to-cloud offloading service by moving a mobile application from a slow mobile device to a fast server in the cloud during runtime [1, 2, 3]. Such offloading techniques can overcome the low performance and quick battery consumption of mobile devices, fundamental challenges in the mobile computing business. In addition, cross-device resource sharing schemes [4, 5, 6] enable several interesting multi-device scenarios. When watching movies or viewing PDF files, a user can use a large TV screen rather than a smaller smartphone screen. Similarly, a user can take pictures from various angles by using multiple remote cameras.

Massive cloud resources (e.g., CPU, DRAM) have been widely utilized in

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

various ways for mobile computing, and cloud code offloading is one of promising solutions. However, the existing offload schemes depend on many unrealistic assumptions as they blindly ignore the costs of using servers in the cloud [1, 2, 3]. For example, they assume that a target server is always available for free of charge, the server's load is always idle or stable, the application has been previously profiled for the target server, and the postoffload performance matches the user-expected performance. As a result, the existing schemes naturally choose to offload a mobile application to a free server in the cloud. In addition, large memory capacity in cloud servers can be utilized to extend the memory size in a mobile device, which enables aggressive memory caching for fast application switches. However, such remote memory swapping incurs significant swap-in performance degradation.

On the other hand, cross-device resource sharing, a capability to utilize a remote device's resources (including I/O), becomes a desirable feature to enable interesting multi-device services. However, the existing cross-device resource sharing schemes suffer from several challenges. First, using network libraries explicitly imposes significant programming burden on developers [4, 5, 7] as they should follow a server-client model that involves careful task distribution between server and client processes. Distributed programming platform [8] may reduce the programming burden; however, they still impose the burden of efficiently partitioning an application. Second, remote I/O [6] can enable cross-device resource sharing without the programming burden. Unfortunately, it does not support computation (e.g., CPU, memory) and I/O sharing at the same time, which lim-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

its its applicability. More importantly, the existing schemes do not optimize the placement of tasks and hence suffer when running on slow wireless networks.

To overcome the problems of previous solutions, we propose a generalized mobile computing framework for efficient resource sharing between devices (and servers). First, to share computing resources in cloud servers, we design novel cost-effective mobile-to-cloud offloading platform, CMcloud [9], which works nicely under the real-world cloud environments. CMcloud minimizes both the server costs and the user service fee by offloading as many mobile applications to a single server as possible, while satisfying the target performance of all applications. To achieve such goals, CMcloud exploits novel architecture performance modeling and server migration techniques.

Second, to enable the efficient application caching using cloud memory, we introduce fast and robust swap mechanism for mobile devices, CloudSwap [10]. The key idea of CloudSwap is to use the fast local storage as a cache of read-intensive swap pages, while storing prefetch-enabled, write-intensive swap pages in cloud storage. To preserve the lifespan of the local storage, CloudSwap minimizes the number of writes to the local storage by storing the modified portions of the locally swapped pages in a cloud. To reduce the remote swap-in latency, CloudSwap exploits two cloud-assisted prefetch schemes; the app-aware read-ahead scheme and the access pattern-aware prefetch scheme.

Lastly, we propose DynaMix [11], a novel framework that realizes general and efficient cross-device resource sharing. DynaMix maximizes resource coverage by dynamically integrating computation and I/O resources of remote devices

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

with distributed shared memory and I/O request forwarding. Also, DynaMix obviates the need for multi-device programming by providing the resource sharing capability at the low level. More importantly, DynaMix minimizes interdevice 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.

The rest of this thesis is as follows. In Chapter II, we first summarize the background of remote resource sharing on the current mobile computing. In Chapter III, we describe why existing offloading schemes have unrealistic assumptions, and how CMcloud can provide the QoS-guaranteed offload service while minimizing costs. CMcloud reduces the number of service failures by 83% over a static high-load scheme, while even improving the throughput by 31%. In Chapter IV, we illustrate limitations of conventional swap mechanisms and offers CloudSwap to utilize cloud resources for efficient application caching while preserving the lifespan of the local flash storage. The performance of CloudSwap is comparable to a fast, but lifespan-critical local swap system, with only 18% lifespan reduction, compared to the local swap system's 85% lifespan reduction. In Chapter V, we introduce the design and use-cases of DynaMix to enables efficient cross-device resource sharing. Our example implementation on top of Android and Tizen devices shows that DynaMix enables efficient cross-device resource sharing in multi-device services. Lastly, we summarize the related work in Chapter VI, and conclude our work in Chapter VII.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

II. Remote Resource Utilization on Mobile Computing

In this chapter, we summarize how mobile devices have overcome their limited capabilities to provide better user experiences by utilizing remote resources. In addition, we present the overview of three mechanisms proposed in this thesis.

2.1 Utilizing Cloud Resources

Cloud providers such as Amazon and Google have provided large and various resources at a low cost, and such resources can enable mobile users to utilize cloud resources to overcome resource limitations of mobile devices.

Recent advances towards mobile cloud computing propose a selective mobile-to-cloud offloading service by moving a mobile application from a slow mobile device to a fast server in the cloud during runtime. Figure 2.1 shows the example mechanism of code offloading, which executes a mobile application on a cloud by offloading compute-intensive code regions to powerful servers. These offload schemes [1, 2, 3] can improve both the performance of mobile applications and

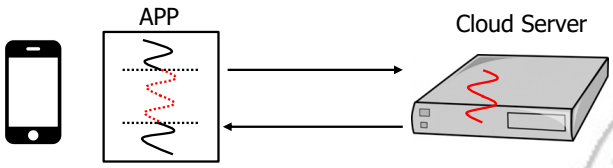
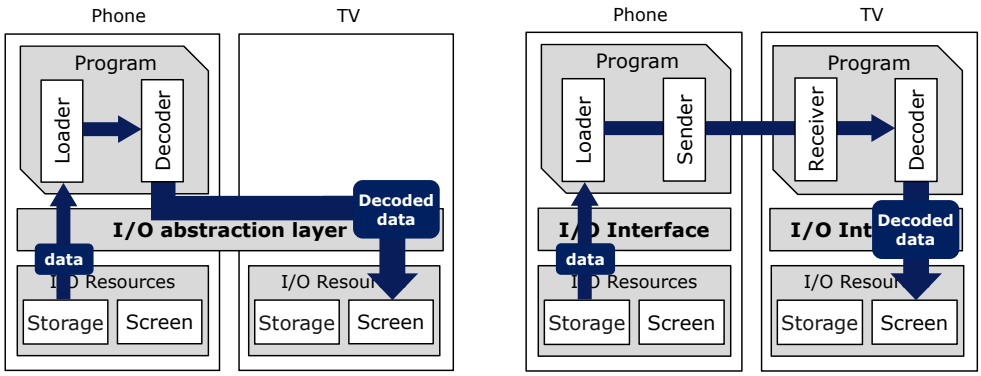


Figure 2.1: Example code offloading

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) I/O request forwarding (b) Manual programming

Figure 2.2: Example cross-device resource sharing (movie player)

the battery lifetime of mobile devices significantly.

Even if applications are not offloaded, mobile users can utilize massive resources in cloud servers to store personal data (e.g., figures, videos, documents) and take advantage of the trained information from other users. We have already experienced a tremendous number of cloud-based services for mobile computing such as storage (e.g., Dropbox, OneDrive, Google Drive), multimedia (e.g., Netflix, YouTube), and machine learning (e.g., Siri, Google Assistant).

2.2 Cross-device Resource Sharing

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. As shown in Figure 2.2, there are mainly two mechanisms

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

in mobile computing to support cross-device resource sharing: request forwarding and manual programming.

First, the request forwarding [6, 12] makes mobile applications access remote resources of any devices through an I/O abstraction layer. This layer provides easy programming environment, and thus programmers do not need to concern how their programs access the remote resources. However, the I/O abstraction layer does not guarantee high performance because unoptimized datapath generate high network traffic.

On the other hand, the manual programming [4, 5, 7] explicitly partitions the application into multiple programs using special APIs to communicate requests. In this scheme, the I/O abstraction layer is no longer used. It does not suffer from high traffic anymore, and it can achieve optimal performance because programmers can precisely determine how their programs access remote resources. However, it requires high programming effort because they have to manually optimize the partitioned programs.

2.3 Three Mechanisms to Efficiently Utilize Remote Resources

These solutions to utilize remote resources can provide various services to users. However, existing system software on mobile devices has a limited capability to efficiently support such services. Although some solutions can provide the functionality to access remote resources, they have suffered from severe performance degradation and inefficient resource utilization. Moreover, as mobile

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

devices use a slow and unreliable wireless network, naïve system software design often exposes the problems more seriously.

In the following chapters, we discuss the limitations of existing schemes to utilize remote resources of cloud servers and mobile devices, and present three novel mechanisms to dynamically and efficiently utilize remote resources. First, we propose a cost-effective mobile cloud platform for offloading, which works nicely under the real-world cloud environments (Chapter III). Second, we propose a fast and robust swap mechanism designed for mobile devices by utilizing cloud resources, which enables efficient application caching (Chapter IV). Lastly, we propose a cross-device resource sharing framework, which integrates a diverse resources and dynamically redistributes tasks (Chapter V).

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

III. Cloud Platform for Cost-Effective Offloading of Mobile Applications

In this chapter, we propose CMcloud, a novel cost-effective mobile-to-cloud offloading platform, which works nicely under the real-world cloud environments. CMcloud minimizes both the server costs and the user service fee by offloading as many mobile applications to a single server as possible, while satisfying the target performance of all applications. To achieve such goals, CMcloud exploits novel architecture performance modeling and server migration techniques.

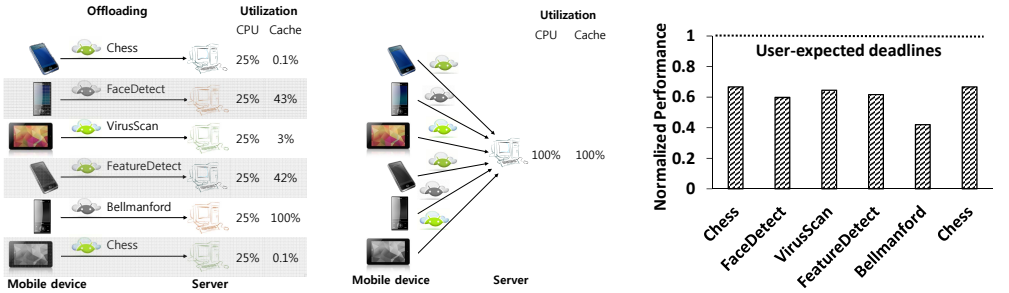
3.1 Existing Schemes and Limitations

To motivate our CMcloud platform, this section introduces conventional offload schemes and their key limitations.

3.1.1 Basic Offload Mechanisms

The recent seminal works on the mobile-to-cloud offloading [1, 2] propose to run mobile applications on high-performance servers. Even though their detailed implementations can differ based on the code modification scope (e.g., user application, kernel,) and the offload granularity (e.g., functions, threads,) they are generally implemented as follows. First, the cloud provider must have profiled the target application's performance and power consumption on both the mobile device and the target server. Next, on receiving an offload request, the

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Offloading to multiple idle servers

(b) Offloading to a busy server

(c) QoS failure due to incorrect profiling

Figure 3.1: Limitation of the existing mobile-cloud offloading schemes

cloud provider compares the application's profiled performance on the mobile device and the target server. If any performance improvement is expected, which is likely to be the case unless the communication latency becomes an obvious bottleneck, the cloud provider offloads the application to the target server, and moves it back to the mobile device after the user-specified execution region is completed.

3.1.2 Limitations of Existing schemes

However, as the existing schemes do not consider the user's service purchasing costs nor the cloud provider's server operation costs, they cannot be applied to the real-world cloud environments, where the cloud provider aims to maximize the server throughput or to minimize the server costs and charges the users based on their cloud resource usage.

Costs of offload services. The existing schemes completely ignore the costs of offload service by assuming that servers are provided for free and they run only

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

one mobile application or maintain a same static load per server. Therefore, they always perform offloading as long as any amount of performance improvement is expected, which is likely to be the case because a lightly loaded server is available and runs faster than a mobile device. Figure 3.1a shows a typical scenario in which each four-core server accepts only a single offload request to achieve the highest performance and guarantee the user-expected performance.

However, the real-world clouds are designed to run as many applications as possible on each server to maximize the server's throughput or to minimize the number of active servers [13]. Therefore, if the existing schemes run only a small static load per server, the costs of operating the server and thus the user service fee will be significantly increased, which makes the mobile cloud computing business infeasible. Figure 3.1b shows a scenario in which multiple offload requests are serviced on a single server with a tradeoff between the server utilization and the offload performance.

Costs of service failures. To reduce the costs of operating the cloud and the user service fee, the cloud provider must allow to offload as many mobile applications to each server as possible. However, offloading too many applications to each server incurs a new challenge in guaranteeing the user-expected offload performance because multiple applications come to contend for the sharing server resources such as cores and caches.

We define the number of offloaded applications completing within the user-expected deadline over the number of all offloaded applications as the offload service's quality of service (QoS). It should be noted that even a small QoS

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

violation is unacceptable in the cloud business, as the users only pay the fee as long as the expected performance is achieved. Figure 3.1c shows a scenario in which five applications in Figure 3.1a are now offloaded to a single four-core server and all applications fail to complete within the user-expected deadlines. In this case, five applications contend for four cores and the last-level cache (LLC) available on the server.

Costs of profiling. The existing schemes assume that performance has been previously profiled for the target server and the offloading always achieves the profiled performance. However, this assumption is broken when an application is now offloaded to a target server which is running other applications to reduce the server costs. To enable an accurate performance estimation, the existing schemes must have profiled for all possible load states of each server. However, it is unrealistic for the cloud provider to statically profile every application for all possible server load states.

3.2 CMcloud offloading

In this section, we first describe CMcloud's key design goals. Next, we present its basic operation model and architecture model consisting of three key components.

CMcloud must satisfy the following design goals to enable a cost-effective offload service. First, CMcloud must target a real-world commercial cloud environment, where servers are highly utilized by running multiple applications per server, Second, the cloud provider must be able to find the most cost-effective

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

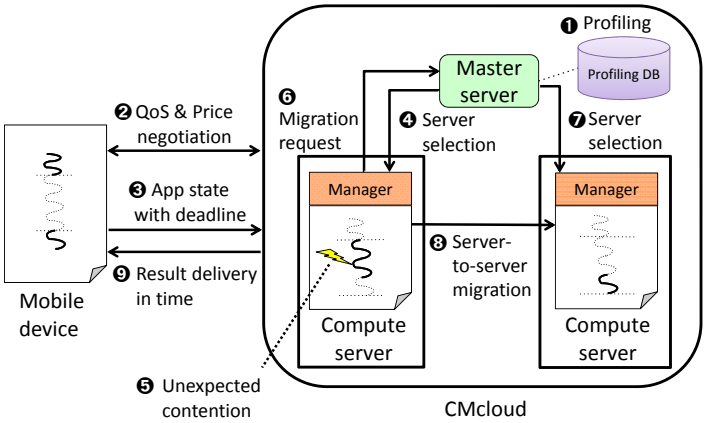


Figure 3.2: CMcloud's example operation model

target server whose remaining resource is just large enough to achieve the target performance, regardless of its utilization. Finally, once an application is offloaded to the cloud, CMcloud must deliver the user-expected performance by considering the QoS success as a primary requirement.

3.2.1 Operation Model

Figure 3.2 illustrates how CMcloud performs an offloading once a user agrees to purchase the offload service. Therefore, the cloud provider now has a target deadline for each application to be completed by also considering a variation in the mobile-to-cloud transfer latency. The cloud provider must satisfy the deadline using the minimum server resources.

(1) Profiling on a reference server. CMcloud chooses a reference-model server in the cloud which is used to profile all offload-enabled mobile applications. Any server can be chosen as a reference-model server as long as it is equipped with a basic set of performance counters. CMcloud profiles the application's

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

execution when the reference-model server is idle, and stores the information in the profiling DB. It should be noted that the same applications with different inputs are considered as independent applications, as also proposed by [1, 2]. Section 3.3.1 describes the profiling mechanism in more detail.

(2)–(3) Offloading the application. The user requests an offload service, agrees on the service fee, and transfers the application with the termination point and the target deadline. Section 3.2.2 describes CMcloud's mobile-to-cloud offload mechanism in more detail.

(4) Selecting a target server. The cloud provider finds the most cost-effective server to complete the application within the target deadline using the minimum amount of resources. At this step, CMcloud applies a performance modeling methodology to estimate the application's performance on the target server by differences in server specifications (e.g., clock frequency, cache size) and load states between the reference-model server and the target server. Section 3.3.2 describes the modeling mechanism in detail.

(5) Detecting a QoS failure. While running the application, the target server monitors the application's progress to detect a potential failure of completing the application within the target deadline, due to either an unexpected performance contention or inaccurate performance estimation. Section 3.3.3 describes the monitoring mechanism in detail.

(6) Migrating to another server. On detecting a potential QoS failure, CMcloud accelerates the application by migrating it to a faster server. Section 3.3.4 describes the server-to-server migration mechanism in detail.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

(7)–(8) Migration server selection. Similar to the step (2)–(4), the cloud provider selects the best target server based on the cost effectiveness and migrates the application to a new server. The cloud provider can repeat the steps from (5) to (8) to maximize the server throughput, while satisfying the QoS requirement.
(9) Completion. On reaching the offload termination point, the application is migrated back to the mobile device.

As a result, the user always achieves the expected performance for the paid service fee, while preserving the mobile device's battery. At the same time, the cloud provider can increase the server utilization to reduce both the datacenter operation costs and the offload service fee.

3.2.2 Architecture Model

In this section, we describe our CMcloud architecture, which consists of a single *master server* and the rest of servers as *compute servers*, as shown in Figure 3.3.

Master server. The master server consists of three components: *profiling DB*, *performance estimator*, and *target selector*. First, the profiling DB contains the profiled execution information on all offload-enable mobile applications on the reference-model server. Next, the performance estimator predicts the application's performance on a current candidate target server analyzing the profiled information on the reference-model server, and differences in server specifications and utilization between the reference-model server and the candidate target server. Finally, the target selector finds the most cost-effective target server which will deliver the user-expected performance at the minimum costs.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

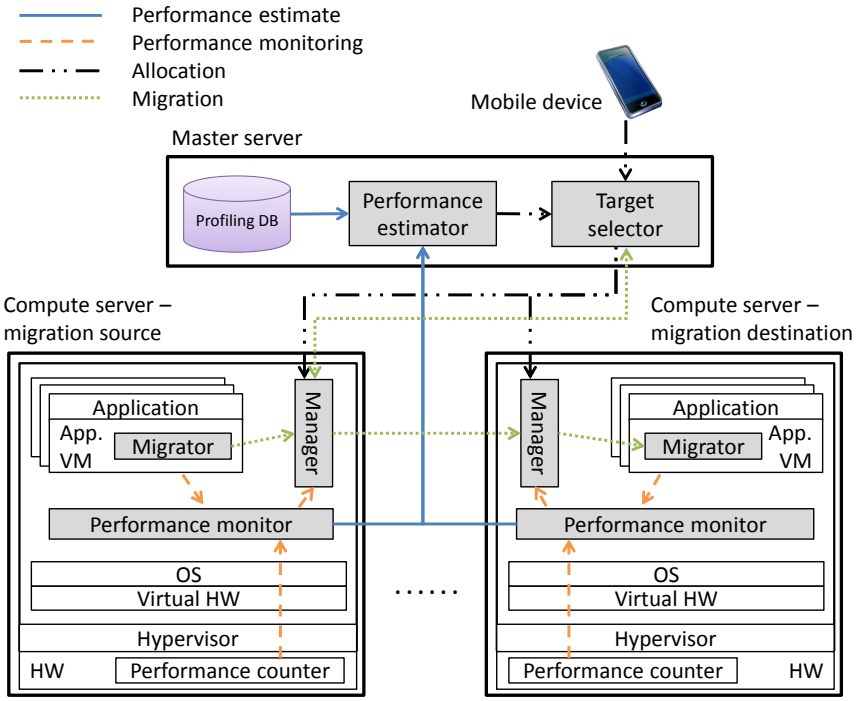


Figure 3.3: CMcloud's basic architecture model

Compute server. The compute server consists of three components: *manager*, *performance monitor*, and *migrator*. First, the manager communicates with other components and servers by handling requests and replies. Next, the performance monitor measures the application's on-going performance to detect a potential QoS failure (i.e., failing to meet the user-requested deadline) by exploiting the current server's performance counters and the execution profile stored in the profiling DB. Finally, on detecting a potential QoS failure, the migrator embedded in the application virtual machine (VM) suspends the application's execution, migrates its execution state, and continues to execute on a new target server.

Offload-ready mobile device. The user's mobile device and operating system

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

must be able to offload a mobile application to the cloud. In this work, we implemented the MAUI-like offloading model as proposed in [1]. For example, the offload handler predetermines offload-enabled regions as remote-executable methods (RM). Therefore, the master server must profile the RM methods and store the profiled information in the profiling DB. Even though we used the MAUI-like model for this work as it does not require to modify the operating system, CMcloud implementation is orthogonal to the mobile-to-cloud offload implementation. CMcloud focuses on providing the cost-effective cloud platform. Therefore, CMcloud can be implemented with other mobile-to-cloud offload models.

Network modeling. We modeled 3G and Wi-Fi networks between mobile devices and the cloud using normal distributions of the bandwidth with empirically observed average and deviations. The detailed information is described in Section 3.4.

3.3 CMcloud mechanism

In this section, we describe CMcloud operation mechanisms in detail: reference-server profiling, performance estimation and monitoring, and migration techniques.

3.3.1 Reference-model Server Profiling

The existing offload schemes assume that the offloaded application's performance has been previously profiled for the target server so that they can estimate the application's post-offload performance before making an offload decision. However, if the target server runs different sets of applications from

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

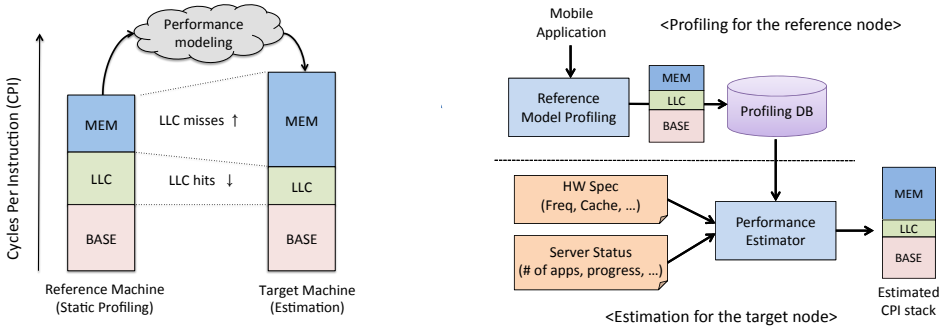
the profiling time, which is the basic operation model of CMcloud, the existing schemes must perform an unbounded number of profiling processes for all kinds of different utilization status even for a single server.

On the other hand, CMcloud still performs a static profiling on a single reference-model server, which can be later translated to the performance for a different target server running any combination of applications. To enable such performance estimation, CMcloud collects the following statistics on the reference-model server using HW performance counters and a memory access tracer.

- **CPI stack.** Execution time breakdown to each performance bottleneck component (Section 3.3.2.)
- **Temporal locality information.** Memory access patterns affecting cache hit and miss rates (Section 3.3.2.)
- **Runtime progress.** Performance progress information collected per second (Section 3.3.3.)

CMcloud can choose any machine equipped with basic performance counters as a reference-model server. However, as our performance modeling assumes that the server's pipeline microarchitecture (e.g., branch predictor, issue order) is maintained, CMcloud must profile an application on all reference-model servers representing unique pipeline microarchitecture families (e.g, one reference machine for all Sandy Bridge family processors.) Other than the pipeline structure, CMcloud does not require extra profiling due to different clock speeds or differ-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Performance estimation using CPI stack (b) Performance estimation process

Figure 3.4: CMcloud's performance estimation process using architecture performance modeling

ent sizes of last-level caches (LLC). More importantly, CMcloud does not require extra profiling due to different server utilization status. Thus, CMcloud's static profiling overhead is much smaller than that of existing offload schemes [1, 2] required to estimating the post-offload performance when severs are highly utilized.

3.3.2 Performance Estimation

In this section, we explain how CMcloud estimates the performance for a different server with different utilization, based on the profiled information using the reference-model server.

A. Performance Analysis using CPI Stack

CPI stack [14, 15] is a performance analysis tool widely used to understand how much each performance losing events (e.g., cache miss, branch misprediction) contributes to the overall performance. As cycle-per-instruction (CPI) explains how many cycles are spent to execute a single instruction on average, it is pos-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

sible to separate the different impacts from different bottlenecks. If the CPU experiences performance losing events such as a cache miss, the final CPI can be obtained by adding the ideal CPI and the extra CPI caused by the cache miss. Therefore, if we are aware of how each event's CPI impact would change on a target architecture, it is possible to construct a target CPI, as shown in Figure 3.4a.

B. Estimation for different idle servers

CMcloud applies the CPI stack method to predict the target application's post-offload performance on a target server, using the profiled performance on the reference-model server. CMcloud first takes the CPI stack collected on the reference-model server, analyzes how key performance losing events will change on a target server, and constructs a new CPI stack to measure the post-offload performance, as shown in Figure 3.4b.

In this work, CMcloud focuses mainly on four performance impact factors, CPU frequency, LLC hit, LLC miss, and store buffer full, because the number of memory instructions and LLC miss rates affect the overall performance most significantly. Even though we consider only four major performance factors in this work, CMcloud can apply more fine-grain bottleneck components as proposed in [15, 16, 17, 18].

Once such CPI stack becomes available, CMcloud can estimate the performance on a target server by adjusting the impact of each CPI stall event as follows. First, CMcloud breaks the overall CPI down to a combination of four sub-CPI events (i.e., ideal latency (*base*), last-level cache hit (*llc*), memory access

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

(mem), store buffer full ($sfull$)) as follows.

$$CPI = CPI_{base} + CPI_{llc} + CPI_{mem} + CPI_{sfull} \tag{3.1}$$

Next, CMcloud measures CPI adjusting factors, $CPI_{ratio,mem}$, $CPI_{ratio,llc}$, and $CPI_{ratio,sfull}$. The factors are used for adjusting the corresponding CPI event for the target server.

If the CPU clock frequency of the target machine is different from that of the reference server, both CPI_{llc} and CPI_{mem} are scaled for the target CPU. If the target server's memory access latency is different from that of the reference server, the ratio is applied to CPI_{mem} as well.

$$\begin{aligned} Freq_{ratio} &= Freq_{target}/Freq_{ref} \\ CPI_{ratio,llc} &= LLC_Hit_{ratio} \times Freq_{ratio} \\ CPI_{ratio,mem} &= LLC_Miss_{ratio} \times Freq_{ratio} \times Lat_{ratio} \end{aligned} \tag{3.2}$$

where $Freq$ is a CPU clock frequency, Lat is a memory access latency, and LLC_Hit and LLC_Miss are the number of LLC hits and misses, respectively.

However, it is difficult to estimate the number of LLC hits and misses, when cache architectures differ between the reference server and the target server. To address the issue, we assume that the size of an LLC differs by the degree of associativity and its cache block replacement policy is based on an LRU policy. In fact, as modern LLCs exploit variations of index hashing mechanisms to effectively increase the degree of associativity, even caches scaled by the number of sets show similar hit and miss patterns as the caches scaled by the associativity.

To discover the application's temporal locality, we leverage the reuse distance (RD) analysis [19], in which RD is the number of distinct and different memory

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

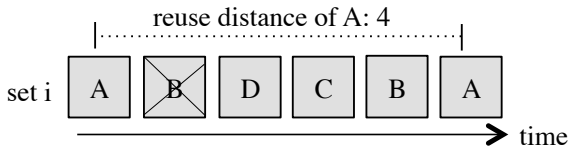


Figure 3.5: An example reuse distance of four for A

accesses between two consecutive references. Figure 3.5 shows an example reuse distance of four between two consecutive memory accesses to A. To collect the reuse distances, we use our memory tracing scheme implemented in the QEMU emulator.

With the LLC scaling and the reuse distances available, LLC hit and miss rates can be estimated for differently sized LLCs. For example, when the LLC's associativity increases from x to y, the number of LLC misses decreases by $\sum_{n=x+1}^{y} C_{RD=n}$ where $C_{RD=n}$ is the number of accesses with the reuse distance of n. Therefore, LLC_Miss_{ratio} can be calculated as follows:

$$LLC_Miss_{ratio} = 1 \pm \sum_{n=x+1}^{y} C_{RD=n}/LLC_Miss_{ref} \tag{3.3}$$

Finally, the penalty caused by the store buffer full depends on some factors including the issue width (W), in-flight store instructions, memory latency and clock frequency. Frequent LLC misses of store instructions can incur a high penalty by filling up the store buffer, which stalls the entire piepline. We estimate such store buffer full cycles using the measured CPI_{sfull} and average store instructions per cycle. We approximately calculate the changed penalty of store

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

buffer full event as follows.

$$CPI_{ratio,sfull} = 1 + \frac{Freq_{ratio} \times Lat_{ratio} - 1}{W \times \%stores} \tag{3.4}$$

By combining equations for each CPI event, we obtain the final target CPI estimation model for different, but idle target servers:

$$CPI_{target} = CPI_{base} + \sum(CPI_{ratio,event} \times CPI_{event}) \tag{3.5}$$

C. Estimation for Different Utilization

In highly utilized cloud environments, each server is highly utilized to achieve the maximum throughput, and thus it will be difficult to find an idle target server for offloading. If an application is offloaded to a target server currently running other applications, the available CPU clock cycles and LLC capacity will be smaller due to the resource sharing among applications.

To calculate the available clock cycles with a core contention, we simply scale the baseline frequency down by the number of applications. We assume that all applications are evenly scheduled with same priorities. If the operating system applies different priorities, this method can be easily adjusted to consider the relative weights as cycles available.

In addition, to estimate the miss rates of the LLCs experiencing a contention, we exploit the miss rate estimation model as proposed in [20]:

$$LLC_Misses = C_{RD>A} + \sum_{x=1}^{A} P_{miss}(x) \times C_{RD=x} \tag{3.6}$$

where A is the LLC's associativity, $C_{RD=x}$ is the number of accesses with the reuse distance of x, and $P_{miss}(x)$ is the possibility of miss for the access with the reuse distance of x.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

P_{miss} depends on which applications are co-located in the same server. This estimation requires the histogram information such as per-application reuse distances. In our work, as the phase of each application varies over time, we collect the information periodically (e.g, one billion instructions.) Then, we adjust the LLC miss estimation model by considering the progress of background applications in the server where a new application is offloaded.

Once such information becomes available, we apply the modified frequency and miss information to the formulas developed in the Section 3.3.2-B.

3.3.3 Performance Monitoring

In this section, we describe CMcloud's performance monitoring mechanism. The monitoring mechanism detects the applications' potential QoS failure caused by either an incorrectly estimated post-offload performance or a resource contention in the servers.

A. Performance Evaluation

The *performance monitor* shown in Figure 3.3 exploits hardware performance counters to check the progress of the target application. Our implementation collects million instructions per second (MIPS) of each application using a modified version of `perf` [21]. Based on the performance estimation model described in Section 3.3.2, the performance is periodically measured and compared as the number of retired instructions for the given period (e.g., one second.)

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

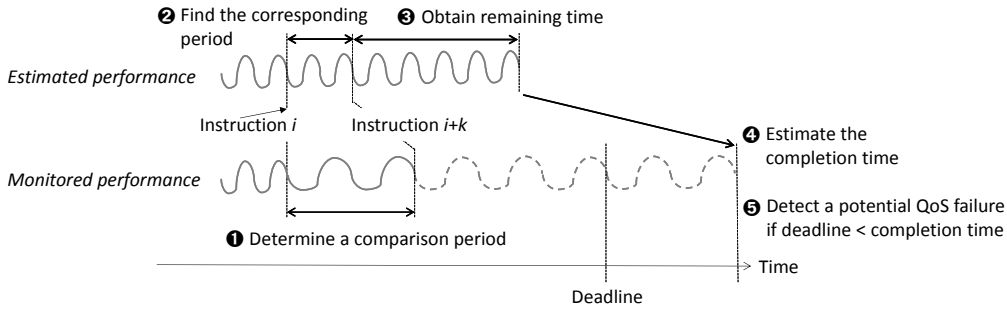


Figure 3.6: Performance monitoring

B. QoS Violation Detection

The performance monitor detects a QoS violation as follows. First, as CM-cloud profiles applications only on a single idle reference-model server, the performance monitor estimates the expected performance on the current target server using the model described in Section 3.3.2. Next, the performance monitor periodically compares the application's target MIPS and the profiled MIPS. Figure 3.6 describes the QoS violation detection method as follows:

(1) Determine a comparison period. The performance monitor determines a small time period (e.g., three past seconds) to compare the MIPS. We use few-second comparison periods to tolerate sudden fine-grain performance variations.

(2) Find the same period for the expected progress. The performance monitor finds the corresponding period from the expected progress, and then measures the relative performance difference during the period.

(3) Obtain the original completion time. The performance monitor computes the time spent to complete the application based on the originally estimated

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

post-offload performance.

(4) Compute the newly expected completion time. By applying the relative performance difference between the expected post-offload performance and the currently monitored performance, the performance monitor can estimate the application's expected completion time.

(5) Detect a QoS failure. The performance monitor can now detect a potential QoS failure by comparing the newly expected completion time against the target deadline agreed between the user and the cloud provider.

3.3.4 Migration

On detecting a potential QoS failure, CMcloud guarantees the application's QoS requirements by migrating the corresponding applications to a faster server. The *migrator* shown in Figure 3.3 performs a low-cost live VM migration.

Destination selection. On detecting a QoS failure, the migrator must find a right destination server. When a migration request is forwarded to the target selector, the target selector finds a right destination node using the performance estimator, as described in Section 3.3.2. The performance estimator exploits not only the status of each server (e.g., the number of active cores, the current server utilization,) but also the application-specific information (e.g., the number of retired instructions, the execution time.)

Performance overhead. Migration can incur non-trivial performance overhead when the large amount of data is transferred over the network. Therefore, CMcloud performs fast inter-server live migrations to minimize a downtime. We assume that servers already contain key application binaries to avoid migrating

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

binaries.

3.3.5 Cost-aware Application Scheduling in Cloud

To minimize the datacenter operation costs, CMcloud targets to improve the server utilization, while maintaining only a small number of active servers in the cloud. To achieve the goal, CMcloud first starts with a small number of nodes and populates the small pool with offloaded applications. Next, on receiving a mobile-to-cloud offload request, the performance estimator collects the estimated performance from the servers. Using this information, the target selector finds the most cost-effective server whose remaining resources are just enough to satisfy the agreed post-offload performance. If the target selector cannot find such server, a new server is activated and added to the current pool of active servers.

3.4 Evaluation

In this section, we first explain our evaluation platform and workloads, and next evaluate CMcloud's accurate performance modeling and its overall cost effectiveness.

Server platform. Our datacenter consists of eight server nodes connected with 10Gbps network, where each server has two CPU sockets. All servers run Ubuntu 12.04 64-bit with Linux Kernel 3.5.0 and KVM [22] with qemu 1.4.0. The KVM release supports both hypervisor and users to access low-level performance counters. To support offloading between mobile phones and x86 servers, we use Android-x86 [23] VMs to run an Android application on a server.

Table 3.1 lists CPU architectures used as reference-model and target servers.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

We use reference models for different pipeline micro-architecture CPU families (e.g., Nehalem, Sandy Bridge) to avoid inaccurate performance estimation across different micro-architectures. As a result, we use two unique reference CPU models in this work because the target servers use one of the pipeline architectures, but differ in the clock frequency and the cache size.

Network. We modeled a Wi-Fi network using a normal distribution of the bandwidth with empirically obtained 18.5Mbps average and a 3.5 standard deviation. Each offload request obtains a unique bandwidth following the distribution. We modeled both 3G and Wi-Fi networks, but used Wi-Fi environments to focus more on the server-side performance for evaluation. CMcloud can be equally applied to 3G network as well.

Workloads. We implemented five real-world mobile applications listed in Table 3.2. We carefully selected these workloads for the reasonable execution latency, while they contend for the shared resources (e.g., last-level cache.) Chess calculates the latency for a computer to find the next move, FaceDetect and FeatureDetect identify human faces and various features from a given image, VirusScan compares 1K virus signatures with 1GB of cloud data, and Bellmanford

Table 3.1: CPUs used for tests

	Processor	Frequency	Cache Size
Reference	Intel Core i7-930¶	2.80 GHz	8 MB
	Intel Core i7-2600‡	3.40 GHz	8 MB
Target	Intel Xeon X5650¶	2.66 GHz	12 MB
	Intel Xeon E5-2630‡	2.30 GHz	15 MB
	Intel Xeon E5-2670‡	2.60 GHz	20 MB

¶Nehalem (Westmere) processor, ‡Sandy Bridge processor

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

finds the shortest path based on a NY-city map. To support offloading, we modified these workloads as proposed in [1]. We also applied Native Interface (JNI) to evaluate memory-intensive workloads.

Clients. We modeled clients as an inflow of offloading requests based on Poisson distribution with 30 requests per minute for the 16-socket cloud. To finish 30 applications per minute, we configured the application ratio as Chess (19.2%), VirusScan (9.7%), FaceDetect (32.3%), FeatureDetect (32.3%), and Bellmanford (6.5%.)

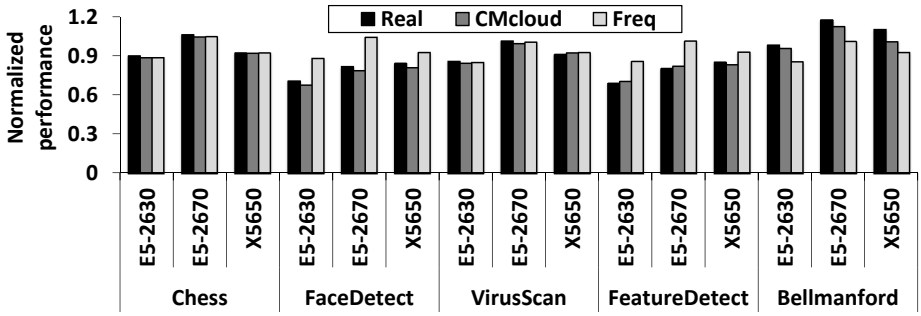
3.4.1 Estimating Target CPU Performance

We first evaluate the accuracy of the proposed performance estimation method using idle target servers by with the reference-model profiling described in Section 3.3.2. Figure 3.7a compares the estimation accuracy between the real performance obtained on the target server and the estimated performance of CMcloud. The x-axis indicates three target-server runs for six workloads, whereas the y-axis shows the performance normalized to the reference machine as shown in Table 3.1. Real bar indicates the actual post-offload performance, while CMcloud bar in-

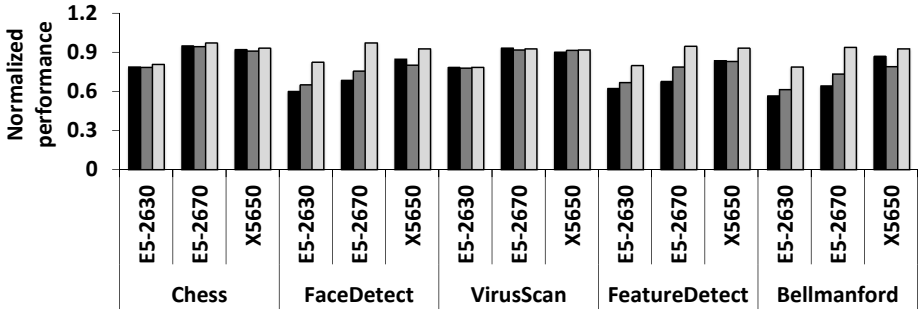
Table 3.2: Workloads (in i7-2600)

	Execution time	Total insts	LLC refs/sec	LLC misses/sec
Chess	26.0 s	126 B	1.2 M	0.1 M
FaceDetect	38.0 s	203 B	17.9 M	5.3 M
VirusScan	74.5 s	503 B	1.0 M	0.6 M
FeatureDetect	41.7 s	222 B	13.9 M	4.8 M
Bellmanford	147.7 s	508 B	24.1 M	6.0 M

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Idle server



(b) Busy server running the five background jobs

Figure 3.7: Accuracy of the performance prediction

dicates the predicted performance. The results indicate that CMcloud predicts the performance of idle target servers with the average error of only 2.9%. Freq bar indicates the performance only when the CPU frequency is considered for the estimation, which leads to the average error of 10.3%.

Next, we repeat the same experiments when each target server runs a group of five baseline applications in background. Figure 3.7b indicates that CMcloud's performance estimation is also accurate even for the highly utilized target servers. The results indicate that CMcloud predicts the performance of busy target servers with the average error of only 5.3%, compared to the 13.4% error of the frequency-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

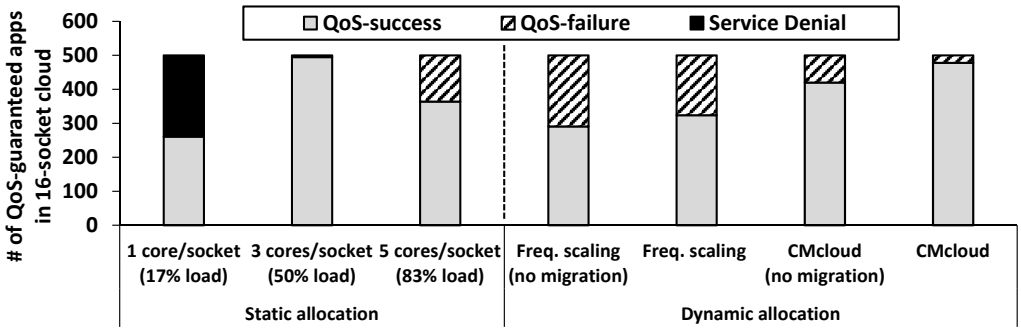


Figure 3.8: Datacenter throughput (out of 500 requests)

only estimation.

3.4.2 Cost Effectiveness with QoS requirements

This section evaluates the cost effectiveness of CMcloud by analyzing the improved server throughput and reduced server costs.

Improved server throughput. Figure 3.8 compares the performance and costs of CMcloud against conventional static server allocation schemes. The X-axis lists seven target server allocation schemes: three static allocation schemes and four dynamic allocation schemes including CMcloud. For static allocation schemes, we configured the cloud provider to assign only one application to each socket (17% load,) three applications to each socket (50% load), and five applications to each socket (83% load.) For dynamic allocations schemes, we evaluated a frequency-only estimation model and CMcloud with/without intra-server migration capability. The Y-axis shows, among 500 offload requests, the number of requests successfully completed within the user-agreed deadline (QoS-success) for the entire cloud, the number of requests violating the deadline (QoS-failure,)

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

and the number of requests turned down by the cloud due to insufficient servers.

Among the static allocation schemes, the 17% load scheme shows the lowest per-socket throughput by utilizing only one core per 6-core socket. The 17% load rejects almost half of the requests due to insufficient servers. On the other hand, the 83% load scheme achieves 75% server throughput, while 25% of workloads fail to complete within the deadline. Even though the 50% load shows the maximum throughput in return of 50% server efficiency, this sweet spot will change for different workloads. Therefore, considering the server under-utilization and the QoS failure are unacceptable for the cloud business, the static allocations cannot be applied as a cost-effective offload scheme.

Among dynamic allocations, CMcloud achieves almost the ideal throughput and even CMcloud without migration capability outperforms two frequency-only estimation models. The result shows that CMcloud improves the server throughput by 84% over the 17% load scheme. Compared to the 83% load scheme, CMcloud reduces the number of service failures by 83%, while even improving the throughput by 31%. The results also show that both the performance modeling and inter-server migration of CMcloud contributed to the improved server throughput separately.

Reduced server costs. Figure 3.9 shows the number of sockets running applications for the first 1000 seconds. In this experiment, we evaluate the server costs of CMcloud against three static load schemes. As expected, the higher-load allocation policies utilize a smaller number of sockets than lighter-load allocation policies. However, CMcloud only activates the minimum number of sockets by

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

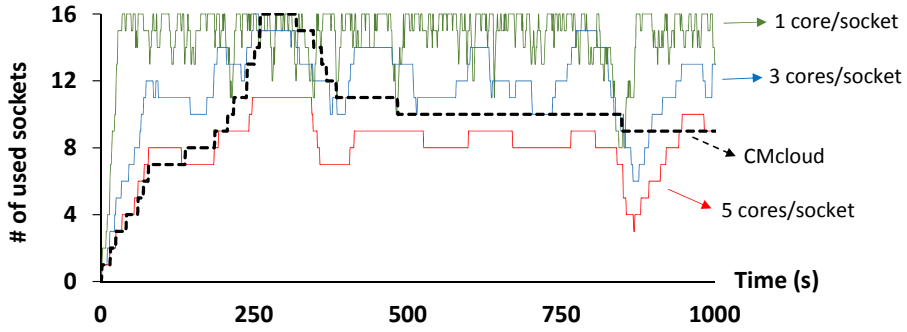


Figure 3.9: Datacenter utilization (out of 16 sockets)

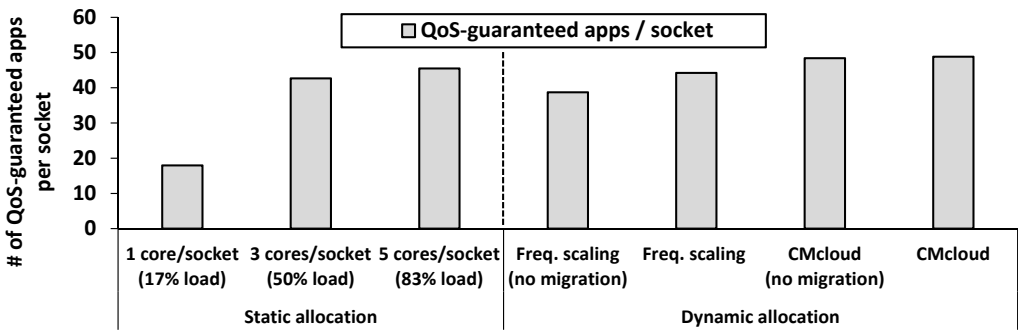


Figure 3.10: Per-socket cost effectiveness

maximizing the throughput, as long as the QoS of applications is not violated. Considering the CMcloud's high throughput shown in Figure 3.8, it is clearly shown that CMcloud consistently operates at lower costs than the 13% and 50% static allocation schemes.

Cost effectiveness. Considering the improved throughput and reduced server costs of CMcloud, Figure 3.10 compares the cost effectiveness of CMcloud against the static allocation schemes. In this figure, we measure the cost effectiveness of the number of applications successfully completed within the deadline per socket, which indicates each socket's cost effectiveness. The results show

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

that CMcloud outperforms all schemes significantly. CMcloud provides a 2.7x higher per-socket throughput over a static light-load scheme (i.e., 17% load.) It should be noted that the relatively high cost-effectiveness of high-load static allocation policy (i.e., 83% load) comes with many QoS failures. On the other hand, CMcloud does not incur unacceptable QoS failures as shown in Figure 3.8.

3.4.3 Offloading/migration Overhead

Table 3.3 shows the overhead of performance monitoring, inter-server migration, and reference-model profiling. Both monitoring and profiling overheads are normalized to the execution latency without profiling. The monitoring overhead is small and thus shown in percentage.

Once applications are offloaded to servers, CMcloud must monitor all applications to detect the potential QoS violations and trigger server-to-server migrations to improve the performance. We use KVM's native live migration method, which can migrate an application paying only the minimum performance loss. By modifying the KVM's live-migration source code, we measure the latency from when the VM stops at the source node to when it restarts at the destination

Table 3.3: CMcloud/Offloading overheads

	Monitoring overhead	Migration downtime	Profling overhead
Chess	1.58%	45 ms	x150
FaceDetect	0.14%	21 ms	x127
VirusScan	4.15%	14 ms	x180
FeatureDetect	0.71%	12 ms	x124
Bellmanford	3.66%	50 ms	x136

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

node. The table 3.3 shows that both monitoring and migration overheads are minimal.

The static profiling can take a long time as it includes the reuse distance analysis obtained by QEMU emulator. However, it is only a one-time overhead paid by the cloud provider and the overhead is not exposed to users. Moreover, CMcloud requires only a single reference-machine profiling, regardless of its current utilization. It should be noted that a similar kind of static profiling is also required by the existing seminal works [1, 2]. Many proposals to reduce the profiling overhead has been proposed, which is orthogonal to our work.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

IV. A Cloud-Assisted Swap Mechanism for Mobile Devices

In this chapter, we propose CloudSwap, a fast and robust swap mechanism for mobile devices to enable the memory-oblivious application caching. The key idea of CloudSwap is to use the fast local storage as a cache of read-intensive swap pages, while storing prefetch-enabled, write-intensive swap pages in a cloud storage. To preserve the lifespan of the local storage, CloudSwap minimizes the number of writes to the local storage by storing the modified portions of the locally swapped pages in a cloud. To reduce the remote swap-in latency, CloudSwap exploits two cloud-assisted prefetch schemes; the app-aware read-ahead scheme and the access pattern-aware prefetch scheme.

4.1 Existing Schemes and Limitations

To motivate our CloudSwap platform, we present existing mechanisms in modern mobile operating systems to enable fast application switches, and discuss their limitations.

4.1.1 Application Caching

Modern mobile device operating systems including Android and iOS implement *application caching* to enable fast application switches. With application caching enabled, the operating systems neither terminate the background ap-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

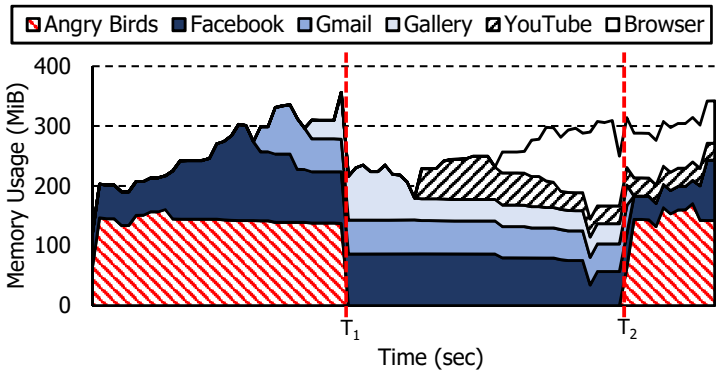


Figure 4.1: Applications killed by Low Memory Killer (LMK) [26] in an example multitasking scenario on an Android device

plication nor reclaim the memory pages allocated to the application. Instead, application caching maintains the application's context and entire memory footprint in the mobile device's physical memory. In this way, application caching can enable fast application switches at the cost of the physical memory.

However, application caching requires a prohibitive amount of physical memory which is unavailable on the low-cost mobile devices. For example, the memory footprint of representative mobile applications can be as large as 133 MB (e.g., Angry Birds in Figure 4.1). It indicates that a device equipped with 1 GB of physical memory cannot cache more than 7 such applications at a time.

The simplest solution to cache more applications is to increase the size of physical memory on mobile devices; however, it is infeasible to keep increasing the size of physical memory on the low-cost mobile devices. Alternatively, the operating systems can compress memory pages (e.g., zram [24], zswap [25]). However, unless the compression ratio continues to increase, the compression is a short-term solution.

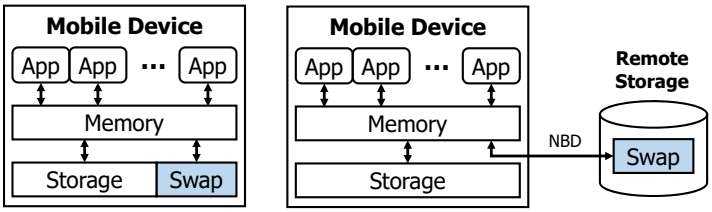
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

4.1.2 Application Termination

When the available physical memory becomes small, modern mobile-device operating systems end up terminating background applications to free the memory pages allocated to them (e.g., Android's Low Memory Killer (LMK) [26], iOS' Jetsam [27]). Figure 4.1 shows how much physical memory each mobile application consumes during an example multitasking scenario (see Section 4.4.1). In the early phase (from 0 to T_1), the mobile device successfully caches four applications (i.e., Angry Birds, Facebook, Gmail, Gallery) as the accumulated memory footprint does not exceed the size of physical memory. However, when Gallery allocates additional memory pages at T_1, the mobile device runs out of free memory and performs LMK to terminate Angry Birds. LMK then terminates Gallery and Gmail at T_2 to provide the freed memory to Angry Birds.

In addition, such an abrupt application termination poses additional burdens to the mobile application programmers. As the operating systems can kill their applications any time, the application programmers must provide extra features not to lose any critical information (e.g., unsaved e-mail drafts) and to be able to resume the execution. For example, Android requires application programmers to provide "killable" methods, i.e., `onPause()`, `onStop()`, `onDestroy()`, for their applications, so the critical state information can be backed up before LMK terminates the applications.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Local Swap (b) Remote Swap

Figure 4.2: Possible swap mechanisms for mobile devices

4.1.3 Possible Swap Mechanisms on Mobile Devices

In order to cache a number of applications, it is essential to implement *swapping* on mobile devices. Swapping provides memory spaces larger than a mobile device's physical memory by allocating a swap partition in the secondary storage. Since then, the mobile device can evict the memory pages of background applications from the physical memory to the swap partition, or bring the requested pages back to the memory. The mobile device can then maintain a large number of applications by caching only their working sets in the physical memory.

As shown in Figure 4.2, there are two possible swap mechanisms for mobile devices, based on where they create the swap partition. First, *Local Swap* allocates a large swap partition in the mobile device's local flash-based storage such as eMMC. Alternatively, *Remote Swap* allocates a large swap partition in a remote server's storage, and accesses to the storage via wireless networks, e.g., network block device (NBD) or network file system (NFS) via Wi-Fi. Both swap mechanisms provide memory spaces much larger than the mobile device's physical memory.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

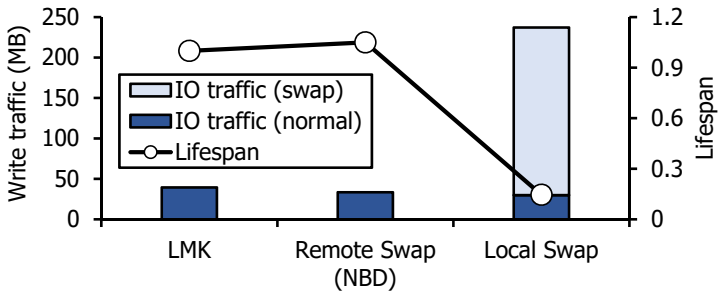
However, these swap mechanisms cannot be applied to the mobile devices due to the fundamental limitations. First, Local Swap significantly reduces the lifespan of the flash-based local storage because flash memory lifespan is inversely proportional to the amount of write traffic [28]. As the conventional swap mechanisms significantly increases the amount of write traffic, it results in the lifespan degradation. We evaluate the lifespan degradation by using a flash storage simulator [29] which takes an I/O trace collected with blktrace [30]. Figure 4.3a shows that swapping can increase the write traffic to the local storage by 7.9 times and reduces the lifespan by 85% as swapping generates a large number of write operations to the storage. On the other hand, Remote Swap can completely eliminate the lifespan problem as it converts all swap-related write operations to network packets. But, it significantly increases the swap-in latency. Figure 4.3b shows that application switches can take up to 20+ seconds. In this case, it will be much faster to re-start the applications from termination.

4.2 CloudSwap

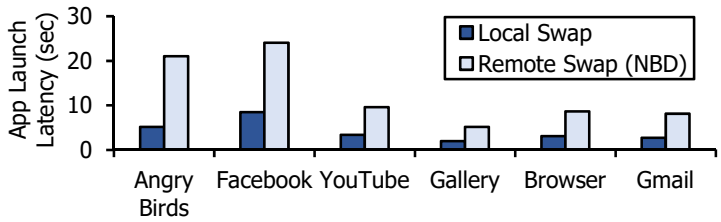
Motivated by the limitations of the conventional swap mechanisms, we define the key design goals of a new swap mechanism as follows:

- It should be able to cache a large number of applications to take full advantage of application caching.
- It should enable fast application switches, comparable to the speed of Local Swap mechanism.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Short lifespan of flash storage with Local Swap



(b) Slow application launch with Remote Swap

Figure 4.3: Limitations of the conventional swap mechanisms

- It should preserve the original lifespan of the mobile device's flash-based local storage.
- It should be easily applicable to the existing mobile devices equipped with small physical memories.

4.2.1 Baseline Remote Swap-In/Out Mechanism

CloudSwap caches a large number of applications by swapping pages in a mobile device out to a large cloud storage. Therefore, mobile-device operating systems using CloudSwap no longer need to terminate their applications due to the physical memory shortage. In addition, mobile devices do not suffer from any lifespan reduction of the flash-based local storage by swapping pages to the cloud

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

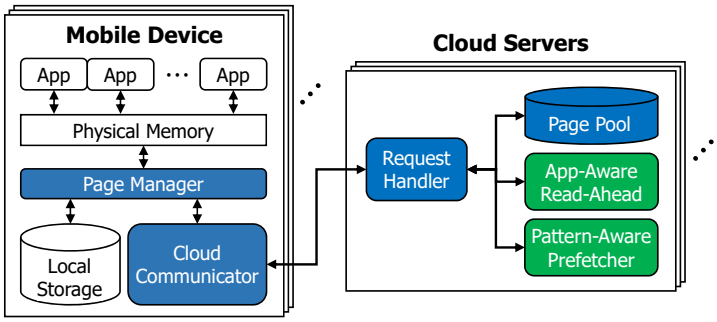
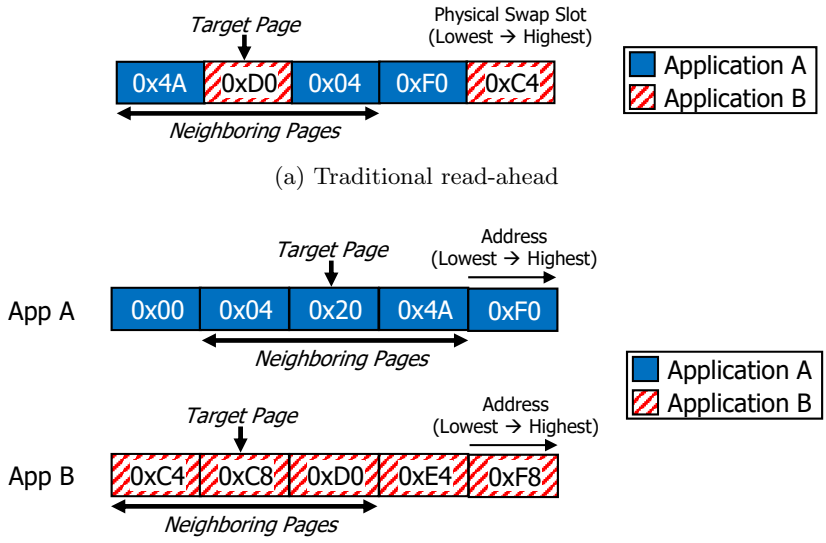


Figure 4.4: CloudSwap system architecture

storage instead of the local storage. Finally, the programmability also improves as application developers do not have to prepare for any unpredictable crash.

The basic design of CloudSwap implements the following software components on top of mobile devices and cloud servers, as shown in Figure 4.4: *page manager*, *cloud communicator*, *request handler* and *page pool*. The page manager is responsible for 1) compressing evicted from the physical memory and 2) intercepting page faults for the remote swap-in requests. The cloud communicator then forwards either the compressed swap-out pages or the remote swap-in requests to the cloud. On the cloud side, the page pool is the cloud-side storage which stores the swap-out pages, and the request handler interacts with the cloud communicator. The request handler receives the compressed page and stores it to the page pool. Similarly, when receiving a swap-in request, the request handler reads the requested page (which is compressed) from the page pool and sends the compressed page to the cloud communicator. Lastly, the page manager decompresses the compressed page and serves the page fault.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Traditional read-ahead

(b) App-aware read-ahead

Figure 4.5: Exploitation of locality with read-ahead

4.2.2 Swap-In using App-aware Read-Ahead (ARA)

Modern operating systems (e.g., Linux) employ *read-ahead* scheme, which swaps in not only the requested page but also a few additional pages at the same time. Swapping in a group of pages likely to be accessed soon can be effective in reducing the swap-in latency.

The traditional read-ahead schemes maintain the physical swap slot order (i.e., positions in the swap area) to take advantage of the disk head's physical position by reading physically contiguous pages in the swap area. However, the traditional read-ahead scheme is not suitable for reducing the application switch latencies as it fails to exploit intra-application locality. Figure 4.5a shows that the traditional system-wide read-ahead scheme is not effective in sending a group

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

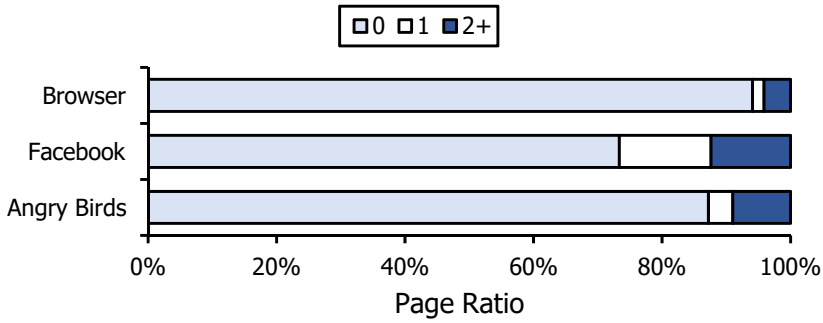


Figure 4.6: Swap-in frequencies of the swap-out pages

of pages belonging to the same application. Furthermore, such swap slots are not directly related to page addresses. As a result, it ends up wasting memory and IO bandwidth by reading pages that are unlikely to be needed soon.

Therefore, to exploit intra-application locality, we introduce ARA which maintains per-application swap-out page lists as shown in Figure 4.5b. The per-application page list can be easily maintained at the cloud side to maximize the spatial locality of adjacent swap-out pages by sorting them in the ascending order of their addresses. By exploiting the intra-application locality, ARA can effectively hide the long latency of remote swap-in process in CloudSwap.

4.2.3 Utilizing Local Storage with Hybrid Swap

To further reduce the overheads of wireless networks, CloudSwap proposes *Hybrid Swap* which uses the mobile device's local storage to store frequently accessed swap pages. To estimate the effectiveness, we profiled the swap-in frequency of each swap-out page (i.e., the number of times a once swap-out page is brought into the physical memory) of Angry Birds, Facebook, and Browser. Fig-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

ure 4.6 shows that only a small fraction of the swap pages is frequently accessed, indicating the potential of using the local storage as a cache of the swap pages.

Hybrid Swap keeps track of the access locality with per-page swap-in counters. When a mobile device swaps a page out to the cloud for the first time, the page manager initializes the page's swap-in counter to zero. On every following swap-in and swap-out access, the page manager increments the swap-in counter by one. When the page manager detects a frequently swap-in page (i.e., 2+ counts), it copies the page's current content to the mobile device's local storage. After then, the mobile device can obtain the page from the local storage.

However, modifying the pages stored in the local storage can degrade the device's lifespan. Therefore, it is desirable to use the local storage as a read-only cache. For the purpose, on every attempt of modifying the cached pages, Hybrid Swap sends the pages' updated portion, *delta*, to the cloud. If the page is to be swapped in, Hybrid Swap should reconstruct the up-to-date page by combining the read-only page in the local storage and the delta in the cloud storage.

When the delta continues to grow, Hybrid Swap comes to suffer more from the increased number of remote transfer latency. Therefore, in case the delta exceeds a certain threshold (i.e., 2 KB in this work), Hybrid Swap replaces the page in the local storage with the reconstructed page. Once the page is overwritten, its delta becomes smaller again and the mobile devices come to minimize the network latency.

However, as the updates in the local storage can degrade the storage's lifespan, Hybrid Swap must carefully select the page reconstruction threshold. There-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

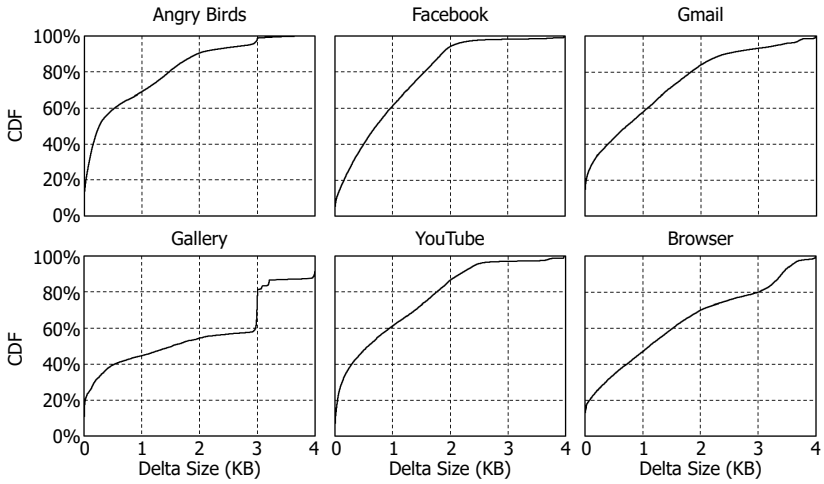


Figure 4.7: Delta sizes of frequently swap-in pages

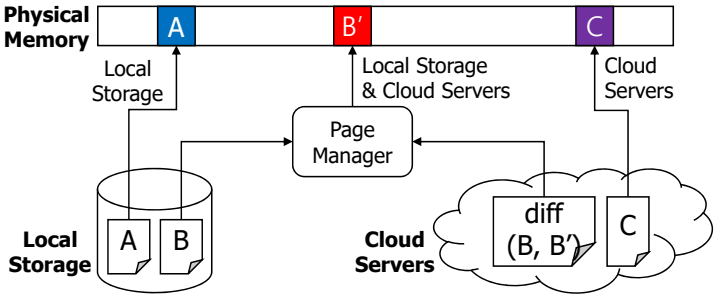
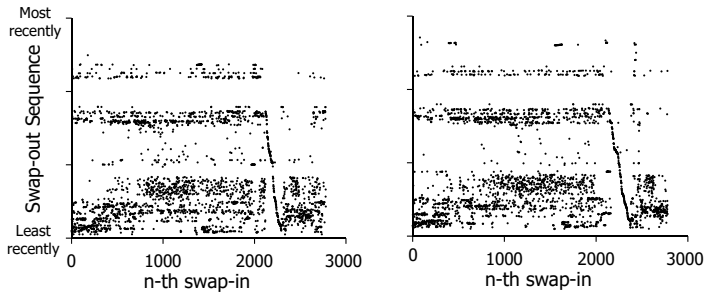


Figure 4.8: Sources of swap-out pages with Hybrid Swap

fore, we profiled how large per-page deltas are for the representative applications. Figure 4.7 shows that the CDFs of the delta sizes tend to saturate at around 2 KB in most of applications, and thus we set our reconstruction threshold to 2 KB for this study. Note that this value can be dynamically changed.

In summary, there are three cases of obtaining the requested swap page in Hybrid Swap, as shown in Figure 4.8.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Device A (b) Device B

Figure 4.9: Similar swap-in patterns of two different devices

4.2.4 Remote Swap-In using Pattern-Aware Prefetch (PAP)

As both Hybrid Swap and ARA can fetch pages only on page faults, CloudSwap uses Pattern-Aware Prefetch (PAP) to predict and prefetch the soon-to-be-accessed pages. PAP first assumes that the cloud maintains the per-user page-access patterns during application switches. Figure 4.9 shows that two different devices launching Angry Birds are expected to exhibit similar swap-in patterns (e.g., if the 100-th swap-out page turns out to be the 200-th swap-in page on device A, the 100-th swap-out page on device B is likely to be the 200-th swap-in page). The similar swap-in patterns infer that the information of a mobile device executing an application can become a valuable reference to other mobile devices which is about to execute the same application.

The goal of PAP is to predict the soon-to-be-accessed swap-out pages with 1) the context information unique to the application state (e.g., call stacks, program counters), 2) the swap-out stream of the mobile device, and 3) the swap-in streams of other mobile devices for the same context. We assume that the cloud

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

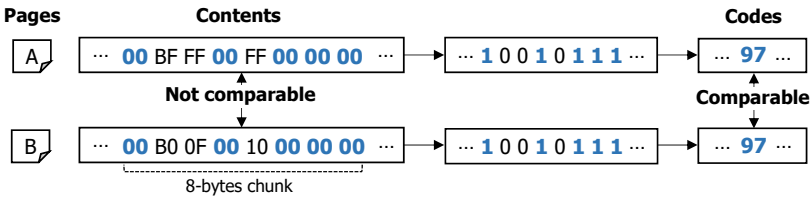


Figure 4.10: Null-frame encoding of two given pages

maintains the swap-in streams of other mobile devices.

First, PAP finds the most similar swap-in stream by comparing the device's current context to that of the available swap-in streams and selecting the most similar stream. As the definition of the context similarity can differ depending on the implementation, we use the activity stacks of Android applications as the context information (see Section 4.3.2-B).

Next, with the target swap-in stream, PAP should determine which pages to be swapped in. Unfortunately, such near-future access prediction is a challenging task due to the following reasons. First, the address-based similarity analysis is infeasible due to the randomized memory spaces of mobile devices. Second, the content-based similarity analysis techniques suffer from many time-varying variables (e.g., pointers, loop iterators, timestamps), user-dependent contents (e.g., e-mails, photos), and the page-content randomization.

To overcome the limitations, PAP converts the 4KB pages of the given swap-out and swap-in streams into *page codes*, 512B signature strings to enable fast and accurate similarity comparisons as follows. First, it exploits plaintext data (e.g., error messages) by checking if the chunk has a stream of printable ASCII characters whose length is larger than a pre-determined threshold. If such a

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

stream exists, it generates a 1-byte encoded character (e.g., CRC-8) of the chunk as the page code.

When failing on such text match, it applies null-frame encoding (Figure 4.10). This null frame is effective for finding the pages with similar data structures in the target page because null bytes can be an indication of separations between variables and we can ignore hex values which are intended to vary across contexts (e.g., address pointer). The figure shows that two different pages having zeros on the same locations can be treated as the similar pages. PAP then solves the longest common substring (LCS) problem on the page codes to identify the soon-to-be-accessed swap-out pages. We regard the page with maximum common length as the most similar one.

Finally, PAP should determine when to swap the predicted pages in to the mobile device. For this work, we applied a machine learning technique proposed by a recent study [31] to predict when a user will launch their applications.

4.2.5 Assumptions for Design and Operation

- We assume that mobile-device users connect to the cloud server via the free Wi-Fi network or the unlimited data plan. Therefore, we do not consider any fee charged for the extra data transfer. We believe that it is a valid assumption as many users are already enjoying data-intensive streaming services on their mobile devices.

- We do not consider the fee which cloud providers might charge for the proposed remote-swap service. It is beyond the scope of our work and

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

many research proposals make the similar assumptions for the new service proposed. In fact, Hybrid Swap and ARA can be implemented also on top of a home network, while PAP still requires the support from the cloud providers (e.g., user-access patterns).

- We rely on TCP/IP protocol to ensure that the pages are not corrupted or lost during the transfer. The inherent retransmission feature provides both data integrity and delivery guarantee. Therefore, we do not consider other unexpected scenarios related to page corruption or loss in this thesis.

- We also take network failures into an account as either short-term or long-term disconnections. For short-term disconnections, CloudSwap can resume its operations right after TCP/IP protocol automatically recovers. For long-term disconnections, CloudSwap can either wait for reconnection or fall back to the traditional LMK-based application caching if LMK-related codes are available (e.g., the existing LMK-based applications).

4.3 Implementation

4.3.1 Mobile Devices

A. Swap Information

We modify the existing swap-out mechanism to use both the cloud-side storage and the local storage (Figure 4.11). First, for each swap-out page, we add a *cloud page ID* to retrieve either the entire or the delta of a swap-out page. Second, we maintain *swap-in counters* to identify frequently swap-in pages. If a

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

swap-out page is frequently swapped in, we keep the page in the local storage and record the location using *local page ID*. In this way, a subsequent swap-in access needs to look up both the cloud page ID and the local page ID.

B. Swapping Pages Out

We modify the existing Linux kernel daemon (i.e., kswapd) to swap pages out to the cloud storage as well as the local storage. The daemon periodically checks remaining free pages, reclaims pages when memory is running low, and evicts them from the physical memory. Then, it computes the delta of the page by querying the cached page to the local storage. In addition, it overwrites the cached page if the delta size exceeds the page refresh threshold. If the cached page does not exist, the page itself becomes the delta.

To reduce the network bandwidth consumption by exploiting temporal locality, the computed deltas are then cached in zswap followed by a bulk buffer and are accessible by the page fault handler. However, as subsequent swap-out

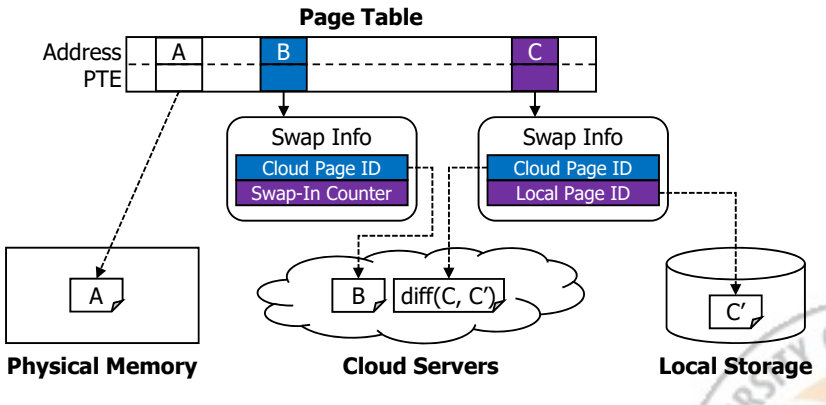


Figure 4.11: Modifications to swap information

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

requests progress, zswap and the bulk buffer runs out of free entries and flushing the deltas cached in the bulk buffer to the cloud becomes necessary. To further minimize the network bandwidth consumption, the kernel module first queries the list of the deltas which are not required by the cloud (i.e., the deltas whose contents are the same to those stored in the cloud). Then, it only sends the deltas required by the cloud after compressing them.

C. Swapping Pages In

We modify Android kernel's page fault handler to access the extended swap information upon a swap-in request. By referring to the extended swap information, the page fault handler correctly fetches the target page's contents from the cloud and/or the local storage. The modified page fault handler also detects a frequently swap-in page by referring to the swap-in counter of the page's swap information. When such a page is detected, it stores the page to the local storage, receives the local page ID, and stores the local page ID to the page's swap information.

To prevent the page fault handler from handling additional pages sent by the cloud (due to app-aware read-ahead and pattern-aware prefetch), we implement a new kernel thread which swaps the additional pages in to the physical memory. Thanks to the new kernel thread, the page fault handler can timely serve urgent page faults as the new kernel thread asynchronously swaps the pages in.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

4.3.2 Cloud Servers

A. Page Pool

We implement the page pool as a key-value (KV) store whose keys are the *cloud page IDs* and values are the swap-out deltas. The cloud page ID is the cloud-assigned identifier which is unique to each swap-out page.

The KV store enables *page sharing*, which merges the values of two different keys if the values have the same page contents. By doing so, page sharing prevents the same page content from being transferred to the cloud multiple times, reducing the network bandwidth consumption.

B. Performing the Cloud's Tasks

On each cloud server, we implement and execute a centralized daemon which performs all the cloud's tasks (i.e., the interaction with mobile devices, app-aware read-ahead, pattern-aware prefetch).

The daemon serves the swap-out requests of mobile devices by receiving the deltas as the first step. Then, the daemon forwards the deltas to the page pool to store the deltas. If some deltas are new to the page pool, the daemon sends the cloud page IDs assigned to such deltas to the mobile devices. Serving the swap-in requests of mobile devices is also simple as the daemon only needs to query the page pool and send the requested deltas to the mobile devices.

The daemon implements app-aware read-ahead by updating the per-app swap-out page lists upon a swap-out request and sending the eight neighboring pages of the requested page upon a swap-in request. The per-app swap-out page

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

lists reside in the cloud server's memory. We choose to send eight neighboring pages following the default configuration of the traditional read-ahead.

The daemon implements pattern-aware prefetch in cooperation with mobile devices as follows: When a mobile device performs an application switch, it sends the information (i.e., class name, screen thumbnail) on the top-of-stack activity of the mobile device's activity stack to the daemon. As soon as the daemon receives the activity information, it stores the activity information as the context information of the switched-out application and searches for the reference swap-in stream (i.e., the swap-in stream which has the most similar activity information) among the already-available swap-in streams collected from other mobile devices. Then, for each swap-out page of the switched-out application, the daemon predicts the corresponding swap-in page by solving the LCS problem using GPUs. Lastly, the daemon swaps the predicted pages in to the mobile device upon the mobile device's request which we currently implement as a manually-controlled signal; however, the signal can be replaced with automated one using recent prediction mechanisms (e.g., [31]). We present a sensitivity study on the effects of when the signal is received by the daemon in the evaluation. This procedure repeats whenever the switched-out application becomes inactive again due to later application switches.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

4.4 Evaluation

4.4.1 Experimental Setup

Table 4.1 lists the parameters we used for the evaluation. To distinguish the I/O traffic of swap operations from that of normal operations, we allocated either a swap space or a KV store as a separate partition.

We carefully selected six representative applications as our workloads. Table 4.2 describes each workload and the corresponding user behavior we simulated. Our workloads span across six application categories: gaming, social media, video streaming, photo viewing, web surfing, and emailing. We automated the user behaviors using Android's monkeyrunner tool [32]. To estimate the expected storage lifespan, we employ a flash storage simulator [29] which takes in as input an I/O trace collected with blktrace [30].

Mobile Devices	
Model	Galaxy Nexus
CPU	ARM Cortex-A9 (dual-core, 1.2 GHz)
Memory	1 GB
Flash Storage	16 GB
Network	Wi-Fi (802.11g), effective B/W & latency: 20 Mbps, 16 ms
Operating System	Android version 4.3 with Linux kernel version 3.0.72
Cloud Servers	
CPU	Intel Xeon E5-2630 (12-core, 2.30 GHz)
GPU	GeForce GTX 560 Ti
Memory	16 GB
Operating System	Ubuntu 12.04.1 with Linux kernel version 3.8.0-35

Table 4.1: Parameters of mobile devices and the cloud

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

Workload	Type	User Behavior
Angry Birds	Gaming	Select and play three stages of the game
Facebook	Social media	Load the 10 latest global feeds and the latest feeds from 10 friends
YouTube	Video streaming	Play two recommended videos each for 50 seconds each
Gallery	Photo viewing	Open 12 pictures saved on the mobile device
Browser[1]	Web surfing	Gradually visit 10 widely-accessed websites
Gmail	Emailing	Read two emails and search for the emails from a friend

Table 4.2: Target workloads and their user behaviors

4.4.2 Application Response Latency

In this experiment, we evaluate the performance of the memory management schemes by inspecting their app response latencies with respect to the reference deadline defined in [34] (i.e., five seconds). We define the app response latency as how long it takes for an application to show its contents on the screen and be responsive to user inputs. For accurate evaluation, we first evict all anonymous pages from the physical memory. Then, we measure how long it takes for the number of page faults to become saturated. We note that comparing the app response latency is a valid performance comparison as minimizing the app response latency is crucial for rapid and responsive user-application interaction [31].

We consider the following versions of CloudSwap. First, ARA only implements ARA. Second, ARA+Hybrid adds Hybrid Swap on top of ARA. Third, ARA+PAP adds PAP on top of ARA. Fourth, ARA+Hybrid+PAP implements

[1]The default web browser in the Android Open Source Project [33].

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

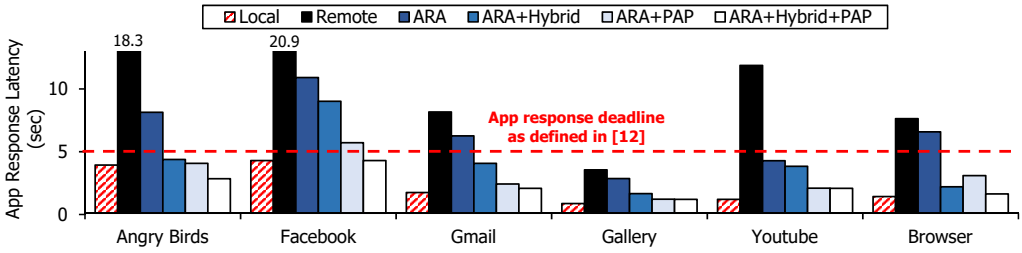


Figure 4.12: Application response latencies of Local Swap (Local), Remote Swap (Remote), and CloudSwap

complete CloudSwap. We compare these schemes against baseline swap mechanisms such as Local Swap and Remote Swap[2]. For ARA+PAP and ARA+Hybrid+PAP, we exclude the prefetch overhead of PAP from the app response latency to take only the prefetch accuracy of PAP into an account; however, note that future implementations can easily eliminate the prefetch overhead by predicting user behaviors (e.g., fast app launching using predictive user context [31]). The impact of prediction accuracy is evaluated in Section 4.4.4.

Figure 4.12 shows the app response latency of each scheme. First, we observe that ARA significantly outperforms Remote Swap for all applications by reducing the app response latency by up to 63.7%. This is because the traditional locality-oblivious read-ahead fails to hide the high network latencies by not exploiting intra-application locality of an application's swap-out pages. On the other hand, ARA successfully exploits the intra-application locality by maintaining per-app swap-out page lists, resulting in a high-quality set of neighboring pages.

Second, we observe that both Hybrid Swap and PAP independently offer greater app response latency improvements as ARA+Hybrid and ARA+PAP

[2]CloudSwap with only the page transfer optimizations between mobile devices and the cloud.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

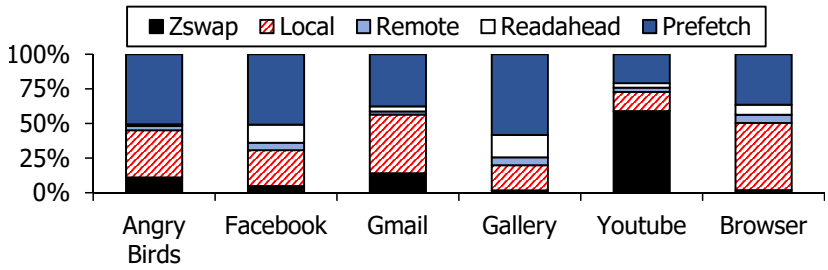


Figure 4.13: Breakdown of the sources of swap-in pages

achieve shorter app response latencies than ARA. However, the improvements provided by Hybrid Swap and PAP significantly differ for some applications. For instance, Browsers benefits more from Hybrid Swap; Facebook and Gmail benefit more from PAP. Therefore, it is desirable to achieve the shorter of the two different app response latencies for such applications.

Lastly, we observe that ARA+Hybrid+PAP outperforms both ARA+Hybrid and ARA+PAP. Furthermore, ARA+Hybrid+PAP even outperforms Local Swap for some applications. This is because Hybrid Swap and PAP complement each other; Hybrid Swap can reduce the latency of swapping the pages predicted by PAP in by caching the deltas of the pages; PAP can reduce the high latency overheads which Hybrid Swap incurs during its training phase. The per-application breakdown of the sources of swap-in pages (Figure 4.13) clearly demonstrates the complementary relationship of Hybrid Swap and PAP. For instance, PAP supplements Hybrid Swap for Gallery while Hybrid Swap supplements PAP for Browser.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

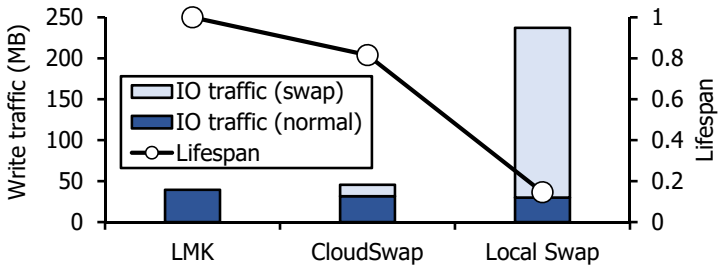


Figure 4.14: The amount of write I/O traffic and the expected storage lifespan of LMK, CloudSwap, and Local Swap

4.4.3 Flash Storage Lifespan

In this experiment, we evaluate how the memory management schemes influence the lifespan of a mobile device's storage. To estimate the expected storage lifespan, we employ a flash storage simulator [29] which takes in as input an I/O trace collected with blktrace [30]. The I/O trace only records accesses to the mobile device's storage. To collect a meaningful I/O trace, we cycle through all the six applications multiple times to warm up Hybrid Swap. We used the I/O trace collected during the fourth iteration for the evaluation.

Figure 4.14 shows the amount of write I/O traffic and the expected storage lifespan of LMK, CloudSwap, and Local Swap. The expected storage lifespan is normalized to that of LMK, the default memory management scheme on Android. We observe that CloudSwap achieves a 18% reduction in the storage lifespan by removing 93.3% of Local Swap's swap traffic. On the other hand, Local Swap continues to suffer from an intolerable amount of swap traffic, degrading the storage lifespan by 85%. CloudSwap removes the large amount of the swap traffic by effectively preventing the mobile devices from writing their swap-out pages to

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

the storage. The remaining 16.7% of write traffic is due to the updates to the delta stored in the storage. In addition, CloudSwap also reduces the normal traffic of LMK by 19.6% as mobile devices no longer need to backup critical application contexts (e.g., write operations in `onDestroy()`).

4.4.4 Prefetch Accuracy and Overhead of PAP

To evaluate the prefetch accuracy and overhead of PAP, we compare PAP against a naïve prefetch mechanism which sends the most recently swap-out pages first (MRS). For a fair comparison, we limit the number of swap-out pages MRS prefetches to that of PAP. The experimental results presented in Table 4.3 show that PAP achieves an average prefetch accuracy of 67.3% while MRS only achieves an average prefetch accuracy of 17.3%, proving the effectiveness of PAP's similarity analysis technique.

Despite its high prefetch accuracy, PAP may become less effective due to its prefetch overhead; however, a recent study [31] hints that mobile devices can eas-

Workloads	**# Prefetched Pages**	**Prefetch Overhead**	**Hit Ratio**	
			PAP	**MRS**
Angry Birds	4051	1.8 sec	60.8%	6.8%
Facebook	5440	2.9 sec	74.8%	39.7%
Gmail	1784	0.7 sec	80.6%	50.4%
Gallery	957	0.5 sec	65.5%	4.6%
Youtube	1816	0.7 sec	78.4%	76.5%
Browser	1609	0.6 sec	49.3%	5.5%
Average			67.3%	17.3%

Table 4.3: Prefetch accuracy and overhead of PAP

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

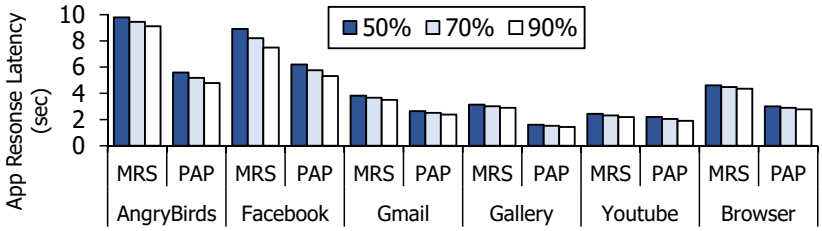


Figure 4.15: App response latencies of PAP and MRS with various user-behavior prediction accuracies

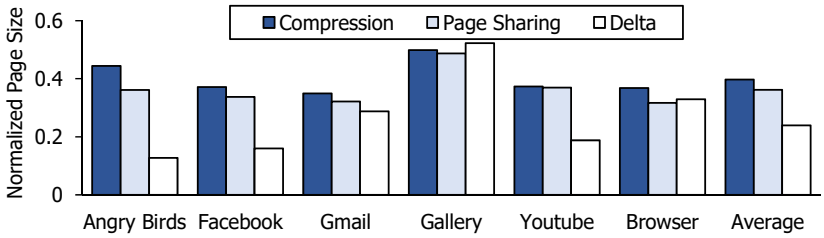


Figure 4.16: Reductions in the swap-out page sizes

ily eliminate the prefetch overhead by accurately predicting which application will be launched next using machine learning. Nevertheless, we evaluate how the accuracy of user-behavior prediction influences PAP. Figure 4.15 compares the app response latencies achieved by PAP and MRS with different user-behavior prediction accuracies. As the prediction accuracy increases, both prefetch schemes show better response latencies because their prefetch overheads are minimized. PAP significantly outperforms MRS for all applications by reducing the app response latency by 30.8% on average.

4.4.5 Network Bandwidth Consumption

In this experiment, we examine how effective CloudSwap is in reducing the network bandwidth consumption. The amount of network bandwidth consump-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

tion is a key metric to determine the overall performance and power consumption of swapping. We also examine how much each of the three main sources of network bandwidth reduction in CloudSwap (i.e., compression, page sharing, deltas) contributes to the overall network bandwidth reduction.

Figure 4.16 shows the average size of network bandwidth consumption per a swap-out page. Compression significantly reduces the network bandwidth consumption by 61% on average as pages have lots of zero bytes positively affecting the compression ratio. In addition to compression, page sharing saves network bandwidth more by 3.6% on average by preventing already-swap-out pages from being sent to the cloud multiple times. Applications having a large amount of frequently-accessed data (e.g., textures) such as Angry Birds tend to benefit more from page sharing. Lastly, deltas introduced by Hybrid Swap further reduce the network bandwidth consumption by 12% on average. Deltas make CloudSwap only send the modified portions of a page instead of the entire page, leading to higher network bandwidth saving. However, for some applications such as Gallery, deltas may incur a negative effect when their sizes become too large.

4.4.6 Power Consumption Overhead

We evaluate the amounts of power a mobile device consumes with Local Swap, Remote Swap, and CloudSwap against LMK by measuring the power consumed while cycling through all the applications twice. We used Monsoon power monitor [35] to measure the power consumed by the mobile device. CloudSwap may consume extra power by transferring data between mobile devices and the cloud. Actually, WiFi communications are known to consume lot of energy. How-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

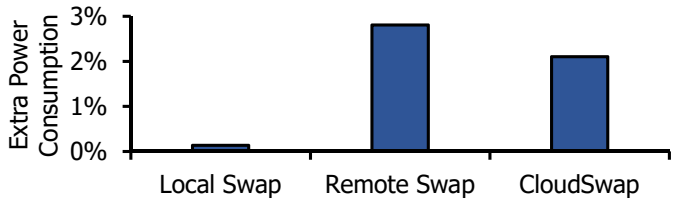


Figure 4.17: Power consumption overheads of Local Swap, Remote Swap, and CloudSwap

ever, we expect CloudSwap to consume a negligible amount of extra power as most swap operations occurs for only a short amount of time (i.e., when a user switches between applications) and other power-hungry components (e.g., screen, GPU, CPU) already consume a huge portion of the overall power consumption [36].

Figure 4.17 shows the additional power a mobile device consumes with Local Swap, Remote Swap, and CloudSwap when compared to LMK. CloudSwap consumes 2.1% more power than LMK which is 0.7 percentage point smaller than the power consumed by Remote Swap. Such a reduction in power consumption is due to Hybrid Swap and PAP which effectively reduce the amount of network bandwidth consumption. Although Local Swap incurs the smallest power overhead, we claim that the slight difference in the extra power consumption is much more tolerable than the huge reduction in the storage lifespan of Local Swap.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

V. Dynamic and Efficient Cross-device Resource Sharing

In this chapter, 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.

5.1 Existing Schemes and Limitations

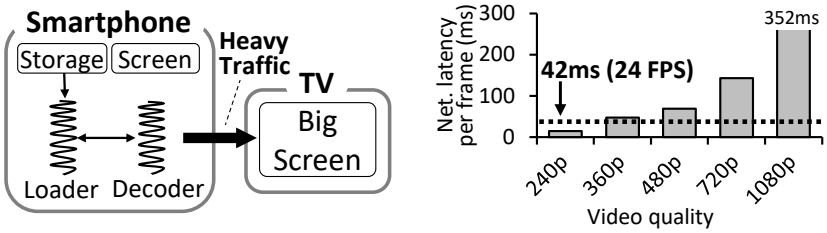
We first present the limitations of existing resource sharing schemes, which motivate the design goals for an ideal resource sharing framework.

5.1.1 Limitations of Existing Schemes

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

I/O Request Forwarding. The I/O request forwarding is a method to utilize

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) An example setup (b) Network latency

Figure 5.1: An example setup to play a video on a remote screen and network latency to send a single frame

remote I/O resources (e.g., camera, screen, audio, sensor) by forwarding I/O requests 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 [6] forwards I/O requests at the kernel level to a remote device which then performs the delivered I/O requests. M+ [12] 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 [4, 5, 7] 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[3]. 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 5.1a shows an

[3]M+ [12] restrictively uses CPU and memory resources for specific platform services.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

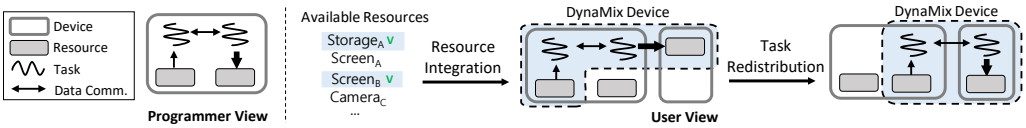


Figure 5.2: An example workflow of the DynaMix framework. Though programmers develop single-device applications, users can configure their services (i.e., DynaMix devices) by selecting desired resources to access remote resources.

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 5.1b 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 [2, 1] and distributed computation [3] 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 [3] implements a software-based distributed shared-memory (DSM) framework to support efficient thread offloading among devices.

However, the applicability of the existing code offloading schemes is limited as follows. First, they support only computation resources for cross-device shar-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

ing, 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 [8] 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.

5.1.2 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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 [4, 5, 7] 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 services 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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

5.2 DynaMix Framework

5.2.1 Overview

Figure 5.2 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 Section 5.3.1. 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 5.2) 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.

5.2.2 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

resource integrator integrates both computation and I/O resources (or constructs a DynaMix device) by applying a distributed shared memory (DSM) model and I/O request forwarding together (Section 5.2.2-A). The thread migrator monitors both inter-thread communication and device connectivity, and dynamically redistributes threads to their optimal locations to minimize the communication overhead (Section 5.2.2-B).

A. 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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

B. 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.

5.3 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 5.3 illustrates the overall architecture.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

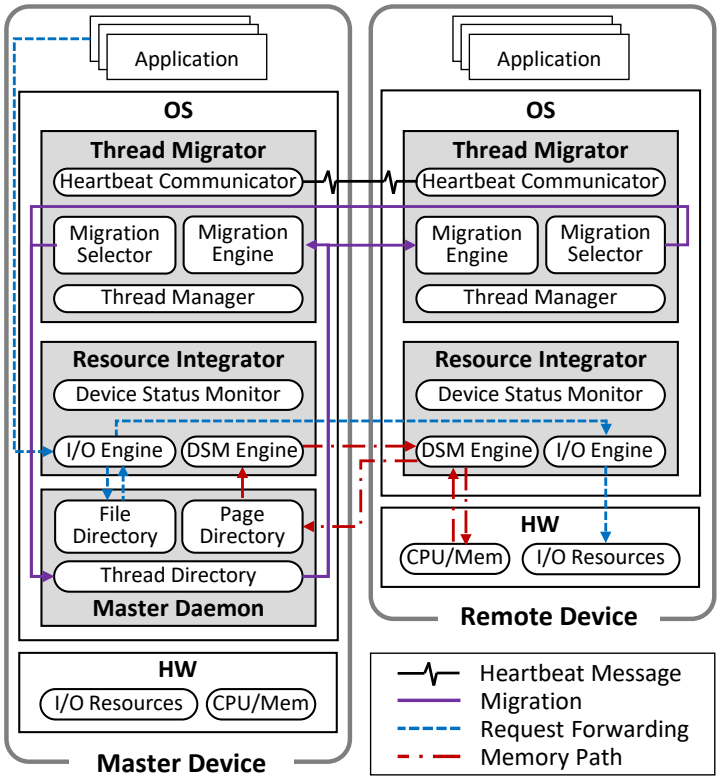


Figure 5.3: DynaMix architectural overview

5.3.1 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.

A. 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,

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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 (Section 5.3.3-B). The DSM engine applies three performance optimizations as follows. First, it adopts a lazy release consistency (LRC) model [37] to safely delay memory synchronization within acquire-release block, similar to previous work [3, 38]. 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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

B. 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 [6, 39]. 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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.

C. 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 [4] (e.g., 30% in our environment). On detecting such contentions, the device status monitor immediately notifies the

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

master daemon to initiate thread redistributions.

5.3.2 Thread Migrator

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

A. 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 (i.e., thread-to-thread and thread-to-resource)[5] 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

[4]This conservative detection using the static threshold works well in DynaMix because the migration selector (Section 5.3.2-B) considers the tradeoffs of all candidates and eventually decides the best migration target.

[5]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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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)
```

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 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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

B. 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 $Perf_{dest,source}$

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

is 0.9, the destination device's CPU is slower than the source's CPU by 10%.

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 (Section 5.3.2-A). 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:

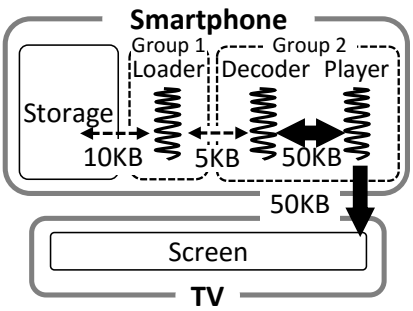
$$T_{ct} = \sum_{i \in tg} \sum_{j \in tcom} NAcq_{i,j} \times (lat_{D(i),D(j)} - lat_{dst,D(j)})$$

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:

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

Example Victim/Destination Selection Scenario. We illustrate example operations of the migration selector. Figure 5.4a shows the data communication status collected by the thread manager, and Figure 5.4b shows the status of de-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(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 5.4: Migration victim and destination selection

vices collected by the device status monitor and the heartbeat communicator. Figure 5.4a 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 5.4c), and reports the best group and destination (i.e., Group 2 and TV) to the master daemon.

C. 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 mi-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

gration. 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 Section 5.3.1-A), 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 [40]) 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.

D. 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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 (Section 5.3.2-A).

5.3.3 Master Daemon

A DynaMix device has a single master daemon[6] 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.

A. Thread Directory

The thread directory manages the global states of threads such as thread locations, and arbitrates the thread migration process. It collects resource con-

[6]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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

tention 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.

B. 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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

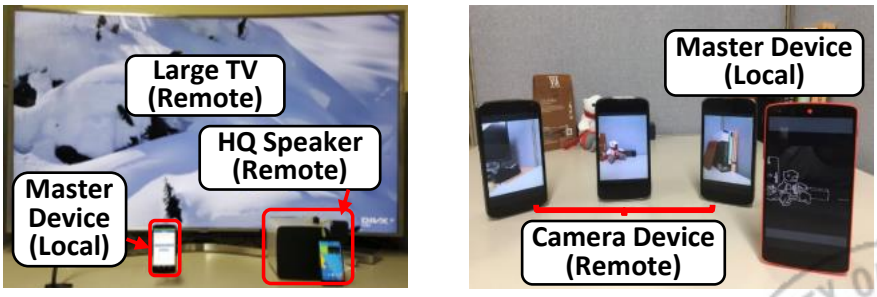
C. File Directory

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.

5.4 Evaluation

5.4.1 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 (Cyanogen-



(a) Home Theater (b) Smart Surveillance

Figure 5.5: Two example DynaMix applications

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

Mod 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 5.5a shows the example home theater setup with three devices. We used FFmpeg [41] 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 5.5b shows the example smart surveillance 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 [42] 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.,

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

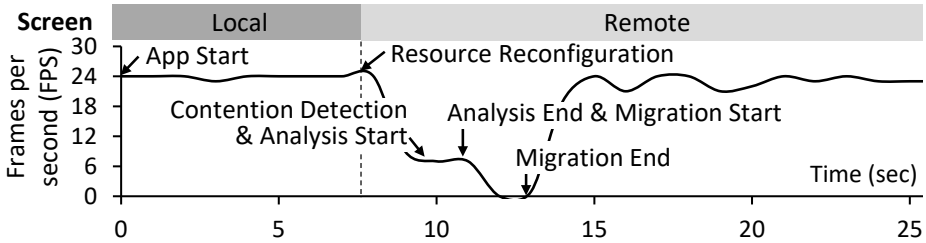


Figure 5.6: Perf. timeline of the home theater application

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 [43]) using a TensorFlow [44] library. Each thread classifies its assigned photos with the classifier and reports the results to the collector thread.

5.4.2 Operation Models

This evaluation revisits the basic operation models of DynaMix in Section 5.2.2. We use the multi-threaded home theater with loader, decoder, and player threads. It runs on the DynaMix device (Figure 5.5a) 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 5.6 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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.

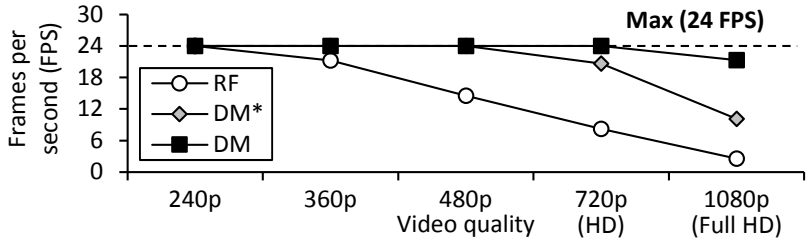
5.4.3 Service Quality

We now evaluate three use cases (i.e., home theater, smart surveillance, and photo classification) to verify that 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 [6]), which accesses a remote resource and receives the result via a wireless network. We also evaluate DynaMix without memory prefetching.

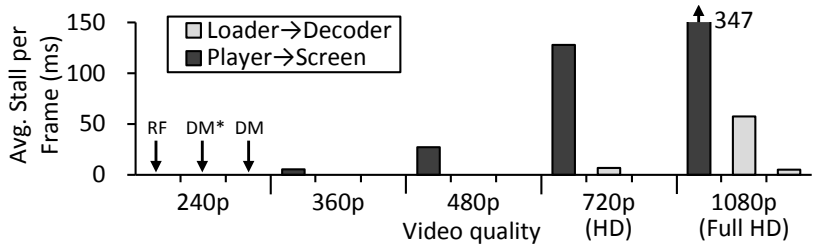
Home Theater. We configure a DynaMix device as explained in Section 5.4.2, and measure the throughput (i.e., frames per second) of the home theater on the DynaMix device. RF forwards decoded frames to the remote device because all threads run on the master device.

Figure 5.7a compares the throughput (FPS) of the home theater for RF and DynaMix. While RF suffers from increasing throughput degradations with the

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Throughput (FPS)



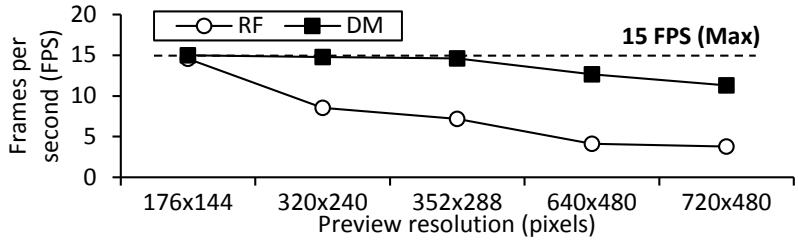
(b) Frame-average network stall time

Figure 5.7: The home theater performance for Request Forwarding (RF) and DynaMix (DM) (*: no prefetching)

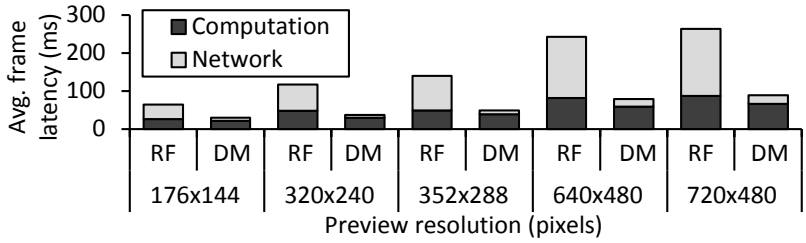
target video quality improved, DynaMix successfully achieves the target throughput up to the decent quality (480p) even without memory prefetching. Enabling the prefetching further enhances the throughput, which makes DynaMix achieve the throughput close to the maximum for the full HD quality (1080p). DynaMix achieves 8.3x higher throughput than RF, while paying only 11% performance drop from the maximum throughput for 1080p.

Figure 5.7b 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교



(a) Throughput (FPS)



(b) Frame-average detection latency breakdown

Figure 5.8: The smart surveillance performance for Request Forwarding (RF) and DynaMix (DM)

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 5.5b. As this application allows a user to select a target remote camera, we randomly choose one camera as the current input feeder. RF

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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 5.8a 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 5.8b 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 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

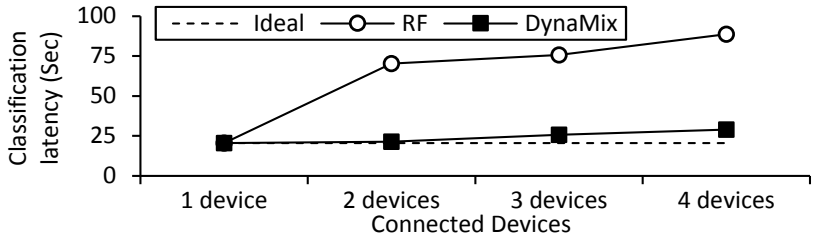


Figure 5.9: The total latency of the photo classification for all photo files on the shared storage system

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 5.9 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 (Section 5.3.2-B). To emphasize the point, Table 5.1 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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

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

(i.e., >1MB) to avoid the transfer overhead.

5.4.4 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 5.10 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.

5.4.5 Power Consumption

In this experiment, we measure the power consumption 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 [35] to measure the power consumption.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

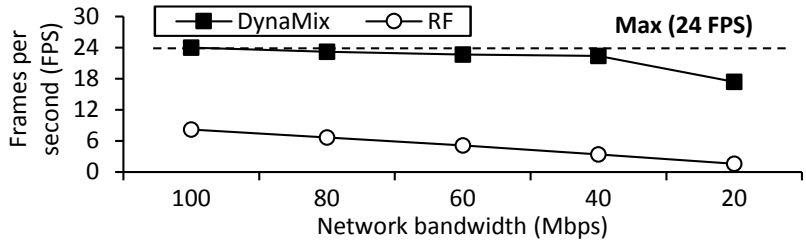


Figure 5.10: The home theater performance on various network bandwidth to play an HD video

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

Table 5.2: The power consumption of the home theater for Request Forwarding (RF) and DynaMix

Table 5.2 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 5.10) and their power consumptions are similar (Table 5.2), 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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

VI. Related Work

Dynamic code offloading and performance prediction. Dynamic code offloading techniques, MAUI [1] and CloneCloud [2], allow users to execute a mobile application on a cloud. However, these schemes are not suitable for the real-world cloud environment due to the lack of the QoS guarantee of applications and a cost model [45, 46]. ThinkAir [47] proposes an on-demand resource allocation for user-side cost and parallel method execution of a mobile application for the QoS guarantee, but focuses on one automatically parallelizable application instead of simultaneous execution of several applications. [48] focuses on reduction in migration overhead by transferring only essential heap objects. Instead, CMcloud targets to mobile cloud computing for simultaneous execution of several applications, the QoS guarantee of applications, and minimization of server cost.

In heterogeneous multi-core systems, PIE [49] and Regression analysis [50] estimate the performance of other cores and assign an appropriate application to an optimal core. These schemes assume that caches have the same size and there is no resource contention. Bubble-Up [51] and Bubble-Flux [52] guarantee QoS of a latency-sensitive application. However, the former performs many sensitivity tests with various memory pressures in advance, and the latter does not allow co-location of multiple latency-sensitive applications. Mantis [53] can automatically estimate the application performance on various inputs by extracting features related to the performance from an application. For an application with different

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

inputs, we can apply this technique to reduce inaccuracy of profiling.

Managing small memories and swap mechanisms. In the constrained memory environments such as mobile systems, the demand for a large memory has stimulated the development of efficient memory management schemes. For example, Android has adopted several aggressive schemes such as LMK [26], kernel samepage merging [54], and Dalvik VM's heap management [55, 56]. Especially, a set of memory techniques including zram [24], zswap [25], and other similar approaches [57, 58, 59] provide a memory slightly larger than the physical memory by compressing pages. Studies on compilers such as [60] target to reduce the memory size requirement of applications. CloudSwap is orthogonal to these schemes and thus can also take advantage of them.

Several studies propose optimized swap mechanisms to efficiently use a mobile device's flash-based storage as a swap space. FlashVM [61] reduces the number of swap-out operations by using zero page sharing and clean page sampling. However, its effectiveness can be restricted on mobile devices due to their fine-grained page management policies. There are several flash-aware optimizations targeting swap spaces on embedded systems, including clean-first LRU page replacement policy [62], compiler-assisted paging [63], write-buffering and subpaging [64] Although these techniques slightly lower the lifespan degradation of flash storage, they cannot be fundamental solutions due to the excessive amount of writes. In contrast, CloudSwap avoids unnecessary writes to flash storage by sending them to the cloud instead. Performing demand paging with remote servers can be an alternative solution to eliminate the lifespan concerns. Ex-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

isting studies assume extremely fast wired-network environments (e.g., InfiniBand) [65, 66]; however, mobile devices use slow wireless networks. Thus, the existing studies suffer from unacceptable application response latencies. CloudSwap overcomes the slow wireless networks by proposing ARA, Hybrid Swap, and PAP.

Cross-device Resource Sharing. Single system image (SSI) [67, 68, 69] 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 [2, 1, 3] offload compute-intensive tasks to powerful servers, even for heterogeneous ISAs [70, 71]. Solutions to utilize other resources such as GPU [72], screen [73], storage [74, 75, 76], generic I/O resources [6] and platform-level services [12] 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[77, 78] 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.

To facilitate easy application development in the multi-device environment, [8] allows programmers to develop unit objects and automatically deploys them across devices. [38] also provides a DSM platform and APIs for multi-device applications. Such platforms, however, still force programmers to explicitly par-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

tition applications with special APIs. [79] 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.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

VII. Conclusion

In the IoT environment, mobile users expect more interesting and better experiences in mobile computing beyond traditional use-cases which considers only a single mobile device. This is because the users now can access massive resources of cloud servers and an increasing number of heterogeneous mobile devices. However, existing system software on mobile devices has a limited capability to support such experiences. Although some solutions can provide the functionality, they have suffered from performance degradation and inefficient resource utilization.

In this thesis, we introduced the mobile computing framework to dynamically and efficiently utilize multiple remote resources from possibly different devices. The key idea was to identify performance-critical resources and possible performance bottlenecks in advance, and to timely redistribute tasks (or resources) across devices. First, we proposed CMcloud to maximize the throughput of computation offloading in cloud servers. We estimated post-offload performance and perform load balancing for offloaded tasks while minimizing resource contentions. Second, we proposed CloudSwap to increase the duration of application caching in a mobile device. We extended a device's memory capacity and timely prepare essential memory pages using the cloud-assisted memory swap mechanism. Lastly, we proposed DynaMix to improve the performance of I/O sharing between devices for multi-device services. We seamlessly integrated multiple resources in

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

different devices and adaptively redistribute tasks between devices.

We implemented three techniques of our framework on top of Android and Linux servers. The evaluation results showed that our proposed framework outperforms conventional software solutions regarding resource efficiency and performance.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

요 약 문

IoT 환경에서 모바일 컴퓨팅은 더이상 로컬 자원에만 국한되지 않는다. WiFi, LTE와 같은 무선 네트워크의 빠른 발전은 모바일 유저들이 다양한 종류의 다중 기기들에 존재하는 여러 자원들을 쉽게 접근할 수 있게 하였다. 결과적으로 이러한 환경은 모바일 컴퓨팅이 단일 기기로는 불가능했던 흥미롭고 새로운 다중 기기 기반 서비스들을 지원하는 것이 가능하게 되었다. 예를 들어 한 유저는 계산위주의 작업을 클라우드 서버에 전송하여 결과를 받음으로써 고성능/저전력 컴퓨팅을 수행할 수 있다. 또한, 여러 기기에 존재하는 카메라, 스크린 등과 같은 I/O 장치들을 동시에 사용하도록 설정하여 유저가 새롭게 서비스를 정의하고 제공받을 수 있다.

하지만 모바일 환경에서의 기존 시스템 소프트웨어들은 이러한 다중 기기 기반 서비스를 제대로 지원하지 못하고 있다. 비록 몇몇 솔루션들이 특정 기능을 제공할 수 있다 하더라도, 성능 저하 및 자원 활용 최적화 실패 등을 겪곤 한다. 또한, 모바일 사용자들은 느리고 불안정한 무선 네트워크를 사용하기 때문에 이러한 문제가 더욱 부각될 수 있다.

본 논문에서는 다중 기기에 존재하는 다양한 원격 자원들을 동적/효율적으로 활용하는 모바일 컴퓨팅 프레임워크를 제안한다. 제안된 방법에서는 성능에 크게 영향을 끼치는 자원과 가능한 성능 저하요소를 미리 파악하고, 자원이나 기기들이 효율적으로 배치되도록 하는것이다. 세부적으로는 클라우드 자원가 다중 기기 자원을 활용하기 위해 크게 세가지 기술을 제안한다.

우선 클라우드의 계산 자원의 효율성을 높이기 위해, 오프로드 이후의 성능을 예

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

측하고 이를 기반으로 여러 오프로드 요청들에 대해 QoS를 만족하도록 서버 배치를 수행한다. 또한, 모바일 기기의 어플리케이션 메모리 캐싱의 효율성을 높이기 위해, 클라우드 메모리 자원을 기기의 메모리를 확장하기 위한 용도로 활용하고 제 시간에 필요한 메모리들을 가져와서 성능 저하를 막는다. 마지막으로, 다중 기기의 컴퓨팅 및 I/O 자원들을 동시에 활용하는 서비스의 성능을 높이기 위해, 효율적으로 자원을 통합하고 여러 작업들을 성능이 최적화되도록 기기들에 재배치한다.. 실험 결과 본 프레임워크의 기술들은 기존 소프트웨어 솔루션의 단점들을 극복하고 자원 효율성 및 성능에 있어 큰 향상을 이룰 수 있음을 보인다.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

References

[1] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[3] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, Hollywood, CA, 2012. USENIX.

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

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

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[6] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong. 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*, MobiSys '14, pages 259–272, New York, NY, USA, 2014. ACM.

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

[8] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D Gribble, Arvind Krishnamurthy, and Henry M Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[9] D. Chae, J. Kim, J. Kim, J. Kim, S. Yang, Y. Cho, Y. Kwon, and Y. Paek. Cmcloud: Cloud platform for cost-effective offloading of mobile applications. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)(CCGRID)*, volume 00, pages 434–444, May 2014.

[10] Dongju Chae, Joonsung Kim, Youngsok Kim, Jangwoo Kim, Kyung-Ah Chang, Sang-Bum Suh, and Hyogun Lee. Cloudswap: A cloud-assisted swap mechanism for mobile devices. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, pages 462–472, 2016.

[11] Dongju Chae, Joonsung Kim, Gwangmu Lee, Hanjun Kim, Kyung-Ah Chang, Hyogun Lee, and Jangwoo Kim. Dynamix: Dynamic mobile device integration for efficient cross-device resource sharing. In *2018 USENIX*

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

Annual Technical Conference (USENIX ATC 18), pages 71–84, Boston, MA, 2018. USENIX Association.

[12] Sangeun Oh, Hyuck Yoo, Dae R. Jeong, Duc Hoang Bui, and Insik Shin. 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.

[13] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. VMware Distributed Resource Management: Design, Implementation, and Lessons Learned. *VMware Technical Journal*, pages 45–64, 2012.

[14] P. G. Emma. Understanding some simple processor-performance limits. In *IBM journal of Research and Development*, May 1997.

[15] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate cpi components. In *ASPLOS '06*.

[16] Brian A. Fields, Rastislav Bodik, Mark D. Hill, and Chris J. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(3):272–304, September 2004.

[17] Intel 64 and ia-32 architectures optimization reference manual. (248966-026), April 2012.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[18] Qi Liang. Performance monitor counter data analysis using counter analyzer. In *IBM developerWorks*, Feb 2009.

[19] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI '03*.

[20] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, 2005.

[21] A. C. de Melo. The new linux 'perf' tools. Slides from Linux Kongress, `http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf`, 2005.

[22] Avi Kivity, Yaniv Kama, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[23] Android-x86. `http://www.android-x86.org`.

[24] Nitin Gupta. Compcache: in-memory compressed swapping. `https://lwn.net/Articles/334649/`.

[25] The zswap: compressed swap cache. `https://www.kernel.org/doc/Documentation/vm/zswap.txt`, 2012.

[26] Android Developers. Low RAM. `https://source.android.com/devices/tech/low-ram.html`.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[27] Jonathan Levin. *Mac OS X and iOS internals: To the Apple's Core.* 2012.

[28] Gokul Soundararajan et al. Extending ssd lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[29] Nitin Agrawal et al. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference (ATC)*, 2008.

[30] Linux man page. blktrace. `http://linux.die.net/man/8/blktrace`.

[31] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[32] Android Developers. monkeyrunner. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

[33] Android OpenSource Project. `https://source.android.com/`.

[34] Android Developers. Keeping Your App Responsive. `http://developer.android.com/training/articles/perf-anr.html#anr`.

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

[36] Aaron Carroll and Gernot Hieser. An analysis of power consumption in a smartphone. In *USENIX Annual Technical Conference (ATC)*, 2010.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[37] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[38] Jason Gao, Anirudh Sivaraman, Nishant Agarwal, HaoQi Li, and Li-Shiuan Peh. DIPLOMA: Consistent and coherent shared memory over mobile phones. In *Proceedings of the 30th International Conference on Computer Design*, 2012.

[39] Ardalan Amiri Sani, Kevin Boos, Shaopu Qin, and Lin Zhong. 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.

[40] Christoper Clark, Keir Fraser, Steven Hand, and Jacob Gorm Hansen. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation*, 2005.

[41] FFmpeg. `https://ffmpeg.org/`.

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

[43] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[44] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[45] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 2011.

[46] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, January 2013.

[47] Sokol Kosta, Andrius Aucinas, and Richard Mortier. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM '12*.

[48] Seungjun Yang, Yongin Kwon, Yeongpil Cho, Hayoon Yi, Donghyun Kwon, Jonghee Youn, and Yunheung Paek. Fast Dynamic Execution Offloading for Efficient Mobile Cloud Computing. In *PerCom '13*.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[49] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE). In *ISCA '12*.

[50] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite Cores: Pushing Heterogeneity into a Core. In *MICRO '12*.

[51] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations Categories and Subject Descriptors. In *MICRO '11*.

[52] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA '13*.

[53] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic Performance Prediction for Smartphone Applications. In *ATC' 13*.

[54] Andrea Arcangeli and Izik Eidus and Chris Wright. Increasing memory density by using KSM. Linux Symposium, 2009.

[55] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March)*, 2010.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[56] Android Developers. Android Memory Management. `https://developer.android.com/training/articles/memory.html`.

[57] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Winter USENIX Conference*, 1993.

[58] Paul R Wilson, Scott F Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference (ATC)*, 1999.

[59] Irina Chihaia Tuduce and Thomas R Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference (ATC)*, 2005.

[60] Nathan Coorprider and John Regehr. Offline Compression for On-Chip RAM. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2007.

[61] Mohit Saxena and Michael M Swift. Flashvm: Virtual memory management on flash. In *USENIX Annual Technical Conference (ATC)*, 2010.

[62] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *International Conference on Compilers, architecture and synthesis for embedded systems (CASES)*, 2006.

[63] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min. Compiler-assisted demand paging for embedded systems with flash memory. In *International Conference on Embedded Software (EMSOFT)*, 2004.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

[64] Han-Lin Li, Chia-Lin Yang, and Hung-Wei Tseng. Energy-aware flash memory management in virtual memory system. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 2008.

[65] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to remote memory over infiniband: An approach using a high performance network block device. In *IEEE Cluster Computing*, 2005.

[66] Hiroko Midorikawa, Motoyoshi Kurokawa, Ryutaro Himeno, and Mitsuhisa Sato. Dlm: A distributed large memory system using remote memory swapping over cluster nodes. In *IEEE Cluster Computing*, 2008.

[67] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[68] David Cheriton. The v distributed system. *Commun. ACM*, 31(3), 1988.

[69] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[70] Gwangmu Lee, Hyunjoon Park, Seonyeong Heo, Kyung-Ah Chang, Hyogun Lee, and Hanjun Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

[71] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. Enabling cross-isa offloading for cots binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017.

[72] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using GPU offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015.

[73] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: A virtual display architecture for thin-client computing. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 2005.

[74] Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramana, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation*, 2006.

[75] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[76] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V Madhyastha, and Cristian Ungureanu. Simba: Tunable end-to-

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

end data consistency for mobile apps. In *Proceedings of the 10th European Conference on Computer Systems*, 2015.

[77] Jiaheng Wang, Daohua Zhu, Chunming Zhao, James CF Li, and Ming Lei. Resource sharing of underlaying device-to-device and uplink cellular communications. *IEEE Communications Letters*, 17(6):1148–1151, 2013.

[78] Chia-Hao Yu, Klaus Doppler, Cassio B Ribeiro, and Olav Tirkkonen. Resource sharing optimization for device-to-device communication underlaying cellular networks. *IEEE Transactions on Wireless communications*, 10(8):2752–2763, 2011.

[79] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Acknowledgements

가장 먼저 지난 7년간 부족한 저에게 아낌없는 가르침을 주신 김장우 교수님께 진심으로 감사의 말씀을 드립니다. 박사 과정 동안 교수님의 애정어린 지도와 끊임없는 지원 덕분에 제가 다양한 연구를 접할 수 있었으며 무사히 성과를 낼 수 있게 되었습니다. 연구 외적으로도 교수님으로부터의 대학원 생활을 함에 있어 필요한 마음가짐과 사회생활에 필요한 여러 가지 조언들은 저의 내면을 더욱 성장시켰습니다. 그리고 세계적인 경쟁력을 가진 연구실 동료들과 함께 일할 수 있도록 좋은 연구실 환경을 만들어주시고 귀중한 연구 철학에 대해 말씀해주신 김종 교수님과 홍성제 교수님, 함께 연구를 진행하며 다양한 조언을 주신 김한준 교수님께 감사의 말씀을 드립니다. 또한, 바쁘신 와중에도 졸업논문심사에 참여하여 좋은 말씀을 해주신 김재준 교수님, 이재욱 교수님, 김광선 교수님께도 깊은 감사의 말씀을 드립니다.

7년에 달하는 대학원 생활은 제 인생에서 가장 중요했던 시기임과 동시에 가장 힘든 시기였기도 하였습니다. 우선 제가 대학원 박사 과정을 무사히 마무리지을 수 있었던 가장 큰 원동력은 HPC/HPCS 연구실의 우수하고 멋진 동료들 덕분이 아닌가 싶습니다. 특히 함께 밤을 지새우며 연구를 진행하고 논문을 작성했던 공저자들 준성, 영석, 광무, 동업, 재형, 아민, 재원, 수헌과 연구에 대해 다양한 조언과 여러 가지 도움을 주었던 연구실 동료들 한휘, 재언, 규현, 평수, 성화, 은진, 헌준, 동문, 일권에게 진심으로 고맙다는 말을 남기고 싶습니다. 졸업 이후에도 연구는 물론 개인적인 교류가 지속되기를 간절히 바랍니다. 또한, 대학원 마지막 학기에 Microsoft Research Asia에서의 인턴 기회와 많은 가르침을 주신 Yongqiang Xiong 박사님과 Peng Cheng 박사님께도 감사의 말씀을 드립니다.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

다음으로 긴 시간 포항에서 함께 수업을 듣고 젊음을 공유했던 7분반 그리고 컴퓨터공학과의 동기들과 선후배님들, 또 자랑스러운 보안 동아리 PLUS에서 즐거운 활동을 함께한 동아리원들에게 고맙다는 말을 전하며, 앞으로도 좋은 인연을 이어나갔으면 합니다. 자주 가지는 못했지만 부산에 내려갈 때마다 저를 반갑게 맞아준 고등학교 친구들과 중학교 친구들에게도 고마움을 전합니다. 모두 자신의 자리에서 최선을 다하여 자주 만나서 안부를 묻고 서로를 챙겨줬으면 합니다.

마지막으로 사랑하는 가족들에게 감사의 말을 전달하고 싶습니다. 정말 오랜 시간을 고생하고 기다려주시며 저의 버팀목이 되어 주신 어머니, 영원한 친구이자 든든한 후원자인 누나, 저를 항상 지켜보고 계실 것이라 믿는 아버지에게 감사드립니다. 그리고, 여러 가지 조언과 아낌없는 도움을 주신 삼촌, 이모, 사촌 형제들을 포함한 여러 친·외가 가족분들에게도 감사드리며, 앞으로 받은 은혜에 보답할 수 있도록 하겠습니다.

다시 한번 도움을 주신 여러분들께 감사드립니다.

채동주 드림

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Curriculum Vitae

Name : Dongju Chae

Education

2012 – 2019 **Pohang University of Science and Technology**

Ph.D. in Computer Science and Engineering

2008 – 2012 **Pohang University of Science and Technology**

B.S. in Computer Science and Engineering

Industry Experience

2018. 9. – 2018. 11 **Microsoft Research Asia, Networking Research Group**

Full-time Intern (Mentor: Yongqiang Xiong)

Publications

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

1. Youngsok Kim, Joonsung Kim, **Dongju Chae**, Daehyun Kim, and Jangwoo Kim, "Achieving Minimal On-Device Inferencing Latency Using Cooperative Single-Layer Acceleration", 14th ACM European Conference on Computer Systems (EuroSys), Mar. 2019 (to appear)

2. **Dongju Chae**, Joonsung Kim, Gwangmu Lee, Hanjun Kim, Kyung-Ah Chang, Hyogun Lee, and Jangwoo Kim, "DynaMix: Dynamic Mobile Device Integration for Efficient Cross-device Resource Sharing", 2018 USENIX Annual Technical Conference (USENIX ATC), July 2018

3. Dongup Kwon, Jaehyung Ahn, **Dongju Chae**, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim, "DCS-ctrl: A Fast and Flexible Device-Control Mechanism for Device-Centric Server Architecture", 45th ACM/IEEE International Symposium on Computer Architecture (ISCA), Jun. 2018

4. **Dongju Chae**, Joonsung Kim, Youngsok Kim, Jangwoo Kim, Kyung-Ah Chang, Sang-Bum Suh, Hyogun Lee, "CloudSwap: A Cloud-Assisted Swap Mechanism for Mobile Devices", 16th ACM/IEEE International Sympsium on Cluster, Cloud and Grid Computing (CCGrid), May 2016

5. **Dongju Chae**, Jihun Kim, Jangwoo Kim, Jong Kim, Seungjun Yang, Yeongpil Cho, Yongin Kwon, and Yunheung Paek, "CMcloud: Cloud Platform for Cost-Effective Offloading of Mobile Applications", 14th ACM/IEEE International Sympsium on Cluster, Cloud and Grid Computing (CCGrid), May 2014

6. Jihun Kim, **Dongju Chae**, Jangwoo Kim, and Jong Kim, "Guide-Copy:

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

Fast and Silent Migration of Virtual Machine for Datacenters", 25th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Nov. 2013

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교