

Enhancing Mobile Device System Using Information from Users and Upper
Layers

Dung Nguyen Tien

Prague, Czech Republic

Bachelor of Science, Charles University in Prague, Czech Republic, 2008
Master of Science, Suffolk University in Boston, United States of America, 2010

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

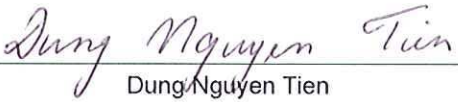
Department of Computer Science

The College of William and Mary
May 2016


APPROVAL PAGE


This Dissertation is submitted in partial fulfillment of
the requirements for the degree of


Doctor of Philosophy

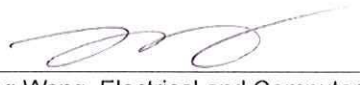

Dung Nguyen Tien

Approved by the Committee, February 2016


Committee Chair
Associate Professor Gang Zhou, Computer Science
The College of William and Mary


Professor Virginia Torczon, Computer Science
The College of William and Mary


Professor Qun Li, Computer Science
The College of William and Mary


Professor Haining Wang, Electrical and Computer Engineering
University of Delaware


Associate Professor Guoliang Xing, Computer Science and Engineering
Michigan State University

ABSTRACT

Despite the rapid hardware upgrades, a common complaint among smartphone owners is the poor battery life. To many users, being required to charge the smartphone after a single day of moderate usage is unacceptable. Moreover, current smartphones suffer various unpredictable delays during operation, e.g., when launching an app, leading to poor user experience. In this dissertation, we provide solutions that enhance systems on portable devices using information obtained from their users and upper layers on the I/O path.

First, we provide an experimental study on how storage I/O path upper layers affect power levels in smartphones, and introduce energy-efficient approaches to reduce energy consumption facilitating various usage patterns. At each layer, we investigate the amount of energy that can be saved, and use that to design and implement a prototype with optimal energy savings named SmartStorage. We evaluate our prototype by using the 20 most popular Android applications, and our energy-efficient approaches achieve from 23% to 52% of energy savings compared to using the current techniques.

Next, we conduct the first large-scale user study on the I/O delay of Android using the data collected from our Android app running on 2611 devices within nine months. Among other factors, we observe that reads experience up to 626% slowdown when blocked by concurrent writes for certain workloads. We use this obtained knowledge to design a system called SmartIO that reduces application delays by prioritizing reads over writes. SmartIO is evaluated extensively on several groups of popular applications. The results show that our system reduces launch delays by up to 37.8%, and run-time delays by up to 29.6%.

Finally, we study the impact of memory on smartphone user-perceived performance. Our heap usage investigation of 20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries, up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Moreover, limited heap may not only cause an app to crash, but may even prevent an app from launching. Therefore, we present iRAM, a system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. The evaluation indicates that iRAM reduces application crashes by up to 14 percent.

TABLE OF CONTENTS

Acknowledgments	v
Dedication	vi
List of Tables	vii
List of Figures	viii
1 Introduction	2
1.1 Problem Statements	3
1.1.1 Exploring Impact of Storage on Smartphone Energy Consumption	3
1.1.2 Improving Application Performance through I/O Optimizations	4
1.1.3 Improving Application Performance through Memory Optimizations	4
1.2 Contributions	6
1.3 Dissertation Organization	8
2 Related Work	10
3 SmartStorage: Storage-aware Smartphone Energy Savings	13
3.1 Introduction	13
3.2 Background and Motivation	15
3.2.1 I/O Path Components	15
3.2.1.1 Cache	16

3.2.1.2	File system	17
3.2.1.3	Block Layer	17
3.2.1.4	Device Driver	17
3.2.1.5	Flash	18
3.2.2	Motivation	18
3.2.2.1	Benchmarks	19
3.2.2.2	Block Layer Level	19
3.2.2.3	Device Driver Level	20
3.2.2.4	Cache	21
3.2.2.5	Optimal Consumption	22
3.3	SmartStorage Design	23
3.3.1	System Architecture	23
3.3.1.1	Kernel Space	23
3.3.1.2	User Space	25
3.3.2	Implementation	26
3.4	Performance Evaluation	27
3.4.1	Experiment Setup	28
3.4.2	Energy Savings	29
3.4.2.1	Energy Savings	29
3.4.2.2	Real-time Power Consumption	30
3.4.2.3	Cost	32
3.4.3	Performance	33
3.4.3.1	I/O Pattern Matching	33
3.4.3.2	Performance Penalties	33
3.4.3.3	Application Delay	35
3.5	Conclusions and Future Work	36

4	SmartIO: Reducing Smartphone Application Delay through Read/Write Isolation	39
4.1	Introduction	39
4.2	Measurement Study	42
4.2.1	Measurement Setup	43
4.2.2	Storage Contribution	44
4.2.3	I/O Slowdown	45
4.2.4	Slowdown Asymmetry	48
4.2.5	Concurrency	50
4.2.6	Summary	51
4.3	System Architecture	52
4.4	Implementation	55
4.5	Performance Evaluation	59
4.5.1	lowait	60
4.5.2	Benchmark Performance	60
4.5.3	Scheduler Comparison	63
4.5.4	Application Performance	65
4.5.5	User-Perceived Performance: Facebook	70
4.6	Discussion and Future Work	72
5	iRAM: Sensing Memory Needs of My Smartphone	75
5.1	Introduction	75
5.2	Measurement Study	78
5.2.1	Free Memory	79
5.2.2	Heap Usage	80
5.2.3	Application Launch	82
5.2.4	Summary	83

5.3	System Architecture Overview	84
5.4	Heap Manager Design	86
5.4.1	Heap Usage Prediction with ARX	87
5.4.2	ARX Parameters	88
5.4.3	Global Heap Threshold	90
5.5	Priority Manager Design	92
5.6	Performance Evaluation	94
5.6.1	Crash Rate	96
5.6.2	Free Memory	98
5.6.3	High Priority Processes Selection	99
5.6.4	Launch Delay	100
5.6.5	User-Perceived Performance: Facebook	102
5.6.6	Overhead	104
5.7	Discussion and Future Work	105
5.8	Conclusion	105
6	Conclusion and Future Work	106

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of many kind, talented, and hardworking people. I would first like to thank my advisor, Professor Gang Zhou, for all of his insight and guidance over the years. Many thanks go to Professor Virginia Torczon, Professor Qun Li, Professor Haining Wang, and Professor Guoliang Xing for serving on my Ph.D. committee.

Next, I would like to extend my thanks to LENS lab members and research collaborators, including Xin Qi, Ge Peng, Qing Yang, Andrew Pyles, Zhen Ren, Daniel Graham, Hongyang Zhao, George Simmons, Matthew Keally, Shuangquan Wang, Zijiang Hao, Jianing Zhao, Tommy Nguyen, and Duy Le.

Lastly, I would like to thank our Computer Science Department Chair, Professor Robert Michael Lewis, and the wonderful Computer Science administration team, including Vanessa Godwin, Jacquelyn Johnson, and Dale Hayes.

This work was supported in part by U.S. National Science Foundation under grants CNS-1250180 and CNS-1253506 (CAREER).

I would like to dedicate this dissertation to my parents, Dr. Nhien Nguyen Duy and Kim Anh Truong, for their endless support, and my wife Mai Anh Do for all the love and sacrifice.

LIST OF TABLES

3.1	Benchmarks	18
3.2	Benchmark I/O Patterns	21
3.3	AnTuTu Benchmark Performance Scores	23
3.4	The 20 Applications Used in Evaluation	29
3.5	Workload Parameters	34
5.1	Application Launch	82
5.2	Angry Birds' RMSE	91

LIST OF FIGURES

3.1 Kernel Components on the I/O Path.	16
3.2 Power for Default Configurations	20
3.3 Power for Queue Depth 4	21
3.4 Power for Write-through Cache	22
3.5 SmartStorage Architecture	24
3.6 Screen shots of the GUI and Tools for Advanced Users	26
3.7 Energy Savings on Nexus One	30
3.8 Energy Savings on Nexus 4	31
3.9 Real-time Power	32
3.10 SmartStorage Throughput and I/O Performance	35
3.11 Application Delay	36
4.1 StoreBench Storage Benchmark	42
4.2 Iowait Values	45
4.3 I/O Slowdown	46
4.4 Response Time ECDF of 2611 Devices	48
4.5 Storage Performance of Top 20 Models	49
4.6 Samsung S5 Speedup over Serial I/O	51
4.7 SmartIO	53
4.8 Dispatch Example	54
4.9 Iowait Before and After	59
4.10 I/O Slowdown	61
4.11 Scheduler Comparison	64

4.12 Launch and Run-time Delay	67
4.13 Power Consumption	69
4.14 User-Perceived Performance of Facebook	71
5.1 Free Memory	79
5.2 Heap Usage	80
5.3 Kernel Components and iRAM	84
5.4 iRAM Architecture	84
5.5 Angry Birds' Heap Usage	87
5.6 Heap Usage Prediction	89
5.7 Crash Rate and Free Memory	95
5.8 Crash Behaviors	96
5.9 Prediction Accuracy	99
5.10 Launch Delay	101
5.11 Launch Delay and Overhead	103

Enhancing Mobile Device System Using Information from Users and Upper Layers

Chapter 1

Introduction

Continual advancements in the technology of smartphones have become an important, if not essential, aspect of our daily life. This is unsurprising since a single mobile device has the ability to call and text family members, check status updates on social media sites, access news and information on the Internet, and play a variety of games for entertainment. However, a common complaint among smartphone owners is the poor battery life. To many such users, being required to charge the smartphone after a single day of moderate usage is unacceptable. In a 2011 market study conducted by ChangeWave [1] concerning smartphone dislikes, 38% of the respondents listed that battery life was their biggest complaint, with other common criticisms such as poor 4G capacity and inadequate screen size lagging far behind. The result of such a study demonstrates the necessity for solutions which address the issue of energy consumption in smartphone devices.

The number of smartphones used worldwide increases each year. According to International Data Corporation, smartphone vendors will ship a total of 918.6 million smartphones in 2013, up 27.2% from the 722.4 million units shipped in 2012 [23]. With their increasing use, smartphone users tend to demand better performance. Moreover, smartphone users are increasingly using phones for work-related activities such as processing emails, reading documents, etc. A study by Forrester Research [18] found that one quarter of work devices were smartphones and tablets. Therefore, it is crucial to study

application performance in smartphones. In particular, reducing the application delay can greatly improve user productivity. In addition, a recent analysis [72] indicates that most user interactions with smartphones are short. Specifically, 80% of the applications are used for less than two minutes. With such brief interactions, applications should be rapid and responsive. However, the same study reports that many apps incur significant delays (up to 10 seconds) during launch and run-time. Our study reveals that Android devices spend a significant portion of their CPU active time (up to 45%) waiting for storage I/Os to complete. This negatively affects the smartphone's overall application performance, and results in slow response time. Therefore, in order to improve the application performance, it is essential to investigate possible reasons of such waits.

1.1 Problem Statements

In this dissertation, we investigate the direct impact of smartphone storage techniques on energy consumption and application performance. Specifically, we answer two key research questions: 1) *How can we optimize smartphone storage in order to save energy?* 2) *How can we improve smartphone application performance with I/O optimization techniques?* 3) *How can we improve application performance with memory optimization techniques?* Answers to these research questions will help engineers come up with smartphone designs that are more energy efficient and at the same time provide better application experience, affecting as many as 1.038 billion smartphone users around the globe (as of September 2012 [24]).

1.1.1 Exploring Impact of Storage on Smartphone Energy Consumption

We address the problem by evaluating smartphone power efficiency at various layers of the I/O path, such as the block layer and device driver. We provide evidence which highlights that the energy consumption of a smartphone can differ depending on storage techniques employed. Different scheduling algorithms on the block layer or different

queue lengths on the device driver impact the total energy consumption differently. We find for 8 benchmarks the combinations of scheduling algorithms and queue lengths with optimal energy savings. In order to save energy, we design our SmartStorage system and implement it on the Android platform. SmartStorage tracks smartphone's I/O pattern in run-time and matches with the benchmark with closest I/O pattern. After having matched with a benchmark, the system dynamically configures an optimal storage configuration to achieve lower energy consumption.

1.1.2 Improving Application Performance through I/O Optimizations

We address the problem by studying the behavior of read and write I/Os. First, the slowdown of reads in the presence of writes is investigated. This slowdown can be one of the main reasons causing the slow launch of applications due to the dominance of reads while launching. Next, the difference in the slowdown of one I/O type due to another may require better I/O scheduling and prioritizing. Therefore, this slowdown asymmetry is researched. Finally, we look at the speedup of concurrent I/Os over serial ones. This provides insights into what type of I/Os benefit more from concurrency. To improve application performance, we design and implement a system prototype called SmartIO on the Android platform. SmartIO measures optimal concurrency parameters for each type of I/O, and issues I/Os with the use of the obtained concurrency parameters. The system reduces the application delay by applying a set of I/O optimizations. Specifically, it assigns higher priority to reads, lower priority to writes, and groups the I/Os based on these priorities.

1.1.3 Improving Application Performance through Memory Optimizations

We address the problem by studying memory usage of several groups of applications. In particular, we identify the amount of memory available before and after their launch. Little available memory may result in delayed I/O operations or frequent communication with

much slower flash disks, which essentially causes slow application response. Insufficient memory may even prevent an app from launching. Next, we investigate heap usage of applications. High heap usage of games and other rich multimedia apps may increase crash rates and likelihood of erroneous behaviors. Finally, we design and implement a system prototype called iRAM on the Android platform. iRAM efficiently maximizes free memory levels, cleans low-priority processes, and maintains optimal heap size limits. The system learns which apps are of high priority for a particular user, and keeps them in the main memory. The launch of such apps is then much faster, since it corresponds to warm launch. iRAM also applies a prediction model to predict heap usage of a set of apps, and dynamically adjusts the heap size based on predicted values. With this set of simple optimizations, iRAM reduces application delays and decreases likelihood of erroneous behaviors.

We found a few works in the research community closely relating to ours. The work of Kim et al. [54] presents an analysis of storage performance on Android smartphones and external flash storage devices. Their discovery of a strong correlation between storage and application performance degradation serves as motivation for our work. Carroll et al. [45] measure the breakdown of energy consumption by the main hardware components in the device. Their direct measurements of each component's current and voltage are used to calculate power. This is done on a smartphone used for scientific purposes only, and many experiments cannot be replicated on commercially available smartphones. We take a different approach based on the precise analysis of the I/O activities between the application layer and the flash storage. Our work is also motivated by cross-layer I/O analysis studied by the authors in [65, 69], which has not been done in smartphones. At this stage, there has been no direct study of the correlation between storage techniques and energy consumption within smartphone devices. We believe our work can help other researchers realize the importance of storage and perhaps trigger more exciting solutions to the smartphone energy consumption problem. Yan et al. [72] propose a system predicting application launch using context such as user location and temporal access patterns.

Their system then provides system support for application prelaunching that reduces perceived delay. However, the proposed system does not solve the slow application launch from the root, but instead lessens its impact. Moreover, the application launch is its only focus.

1.2 Contributions

The overall result of this dissertation are following solutions contributing to improving smartphone energy saving and reducing application delay.

Storage-aware Smartphone Energy Savings. We first focus on investigating the impact of storage on smartphone energy consumption. Based on the findings of this study, we design and implement a system that saves energy. Our main contributions are:

- First, we provide an experimental study on how storage techniques impact energy consumption on smartphones.
- Second, we design and implement the SmartStorage system that tracks I/O pattern of smartphones in run-time and dynamically configures storage parameters with optimal energy savings.
- Third, we evaluate our solution with an Android-based smartphone on the 20 top free applications from Android market and show that our system can save from 23% to 52% of energy. This is achieved with 2.5% energy overhead from running SmartStorage and a difference of 3% in terms of application delay.

Reducing Application Delay through Read/Write Isolation. Next, we explore the impact of storage on application performance. With the use of obtained knowledge, we design and implement a system that reduces application delay in smartphones. Our main contributions are:

- First, through a large scale measurement study on data collected using an Android app we developed, we find that Android devices spend a significant portion of their CPU active time (up to 45%) waiting for storage I/Os to complete. This negatively affects the smartphone's overall application performance, and results in slow response time. Further investigation reveals that a read experiences up to 626% slowdown when blocked by a concurrent write. Additionally, the results indicate significant asymmetry in the slowdown of one I/O type due to another. While the slowdown ratio of a read is up to 6.15, the slowdown ratio of a write is only up to 1.6. Finally, the speedup of concurrent I/Os over serial ones is investigated.
- Second, we design and implement a system prototype called SmartIO that shortens the application delay by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized concurrency parameters.
- Third, we evaluate our system using 20 popular applications from four groups (sensing, regular, streaming, and games) and we show that SmartIO reduces launch delays by up to 37.8%, and run-time delays by up to 29.6%. Moreover, SmartIO also reduces power consumption by 6%.

Sensing Memory Needs of a Smartphone. Finally, we explore the impact of memory on application performance. With the use of obtained knowledge, we design and implement a system that reduces application delay and minimizes erroneous behaviors in smartphones. Our main contributions are:

- First, through a measurement study we find that facilitating warm launch of just five applications is extremely expensive, using up to 36 percent of memory. The resulting little memory left can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Therefore, in order to improve the application performance, we investigate how each application consumes the memory. Our heap usage study of

20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries. This mainly applies to games that require up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Finally, further investigation reveals that limited heap may not only cause an app to crash, but may even prevent an app from launching. While on Samsung S4, all five games fail to launch until the heap size of 64MB, on Nexus 4, all five games fail to launch until the heap size of 128MB. Therefore, the heap size directly affects success or failure of application launch.

- Second, we design and implement iRAM, a system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays.
- Third, our evaluation on memory hungry applications indicates that iRAM reduces application crashes by up to 14 percent. In addition, the results confirm that iRAM increases free memory levels by up to 4.8 times. The evaluation using 40 popular applications from four groups (games, streaming, miscellaneous, and sensing) also shows that iRAM reduces launch delays by up to 78.2 percent. This performance gain comes with 3.5 percent of CPU overhead and 0.9 percent of power overhead.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 presents a detailed survey of related work on smartphone energy and performance. In Chapter 3, we present Smart-Storage, a system that tracks I/O pattern of smartphones in run-time and dynamically configures storage parameters with optimal energy savings. In Chapter 4, we expand on our smartphone optimization approaches to present SmartIO, a system prototype that shortens the application delay by prioritizing reads over writes, and grouping them based on

assigned priorities. Chapter 5 studies the impact of RAM on smartphone user-perceived performance. Finally, we present our conclusion and future work in Chapter 6.

Chapter 2

Related Work

The previous work can be divided into four categories: smartphone storage, smartphone application delay, smartphone power consumption, Linux I/O schedulers, and enterprise solutions.

Smartphone Storage. Kim et al. [55] present an analysis of storage performance on Android smartphones and external flash storage devices. Their discovery of a strong correlation between storage and application performance degradation serves as motivation for our work. We take one step further and investigate possible reasons of such performance degradation, and propose a system to reduce application response using smart I/O optimizations. Nguyen et al. [60] study the impact of the flash storage on smartphone energy efficiency, while the main focus of our work is the application performance. We use obtained knowledge from the study of I/O behaviors to design and implement a system that improves the response time by prioritizing reads over writes, and grouping them based on assigned priorities.

Smartphone Application Delay. Yan et al. [72] propose a system that predicts which apps are to be launched using the context such as user location and temporal access patterns. Their system then provides effective application prelaunching that reduces perceived delay. Parate et al. [62] propose another prediction algorithm to reduce the launch delay. Compared to the previous work, their approach does not require prior training or

additional sensor context. However, mis-predictions of the proposed approaches will lead to significant memory and energy overhead. We address the problem of slow application launch by analyzing possible reasons of the slowdowns in the granularity of read and write I/Os. With this knowledge, we design a system that improves the response time by prioritizing reads over writes. This has a positive impact on the application performance beyond delay.

Smartphone Power Consumption. Works have been done to analyze the power consumption of network traffic in smartphones. Gupta et al. [51] measure the WiFi power consumption by various network activities in smartphones. Balasubramanian et al. [40] measure the power consumption characteristics of GSM, WiFi, and 3G. In this work, we focus on the power consumption analysis of the storage system in smartphones. Several works measure the power consumption of different components in the smartphone. Carroll et al. [45] presents a detailed power consumption analysis of different smartphone subsystems. However, the smartphone used is only for scientific purpose rather than practical usage. In [59, 53], the authors measure and model the power consumption of several hardware subsystems, including CPU, display, graphics, GPS, audio, microphone, and WiFi. In contrast, our work focuses on investigating the impact of the storage system on energy efficiency.

Linux I/O Schedulers. The default I/O scheduler since Linux kernel version 2.6 is the Complete Fair Queuing scheduler (CFQ) [37]. This scheduler has also been adopted as the default one in most Android smartphones, including the ones used in our experiments. However, not optimized for smartphone environments, CFQ may cause long application response time that is the main focus of our work. Other available I/O schedulers (Noop and Deadline [12]) are only used for specialized workloads.

Enterprise Solutions. Flash technology has been recognized in enterprise systems. This is mainly due to its technical merits highlighted in [36, 48], including low power consumption, compact size, and fast random access. This motivated researchers to propose I/O schedulers for flash memory based Solid State Drives in computer storage systems

[50, 56, 57]. We instead study I/O characteristics of smartphones that require different optimization approaches. Our proposed solution is simple, and reduces application delays by up to 37.8%, while still being power efficient. Other enterprise solutions focus on fairness policies [63, 67]. SmartIO builds upon the default Linux I/O scheduler, and adds an additional priority level that preserves the original priorities. Further fairness optimization is beyond the scope of this work.

Chapter 3

SmartStorage: Storage-aware Smartphone Energy Savings

3.1 Introduction

Continual advancements in the technology of smartphones have become an important, if not essential, aspect of our daily life. This is unsurprising since a single mobile device has the ability to call and text family members, check status updates on social media sites, access news and information on the Internet, and play a variety of games for entertainment. However, a common complaint among smartphone owners is the poor battery life. To many such users, being required to charge the smartphone after a single day of moderate usage is unacceptable. In a 2011 market study conducted by ChangeWave [1] concerning smartphone dislikes, 38% of the respondents listed that battery life was their biggest complaint, with other common criticisms such as poor 4G capacity and inadequate screen size lagging far behind. The result of such a study demonstrates the necessity for solutions which address the issue of energy consumption in smartphone devices.

In this chapter, we investigate the direct impact of smartphone storage techniques on total energy consumption and we answer two key research questions: *How does storage affect smartphone power efficiency?* and *How can we optimize smartphone storage*

in order to save more energy? By answering the first question, we find which and how each storage component contributes to the total energy consumption. Different storage techniques have different effects on application performance that results in varying power levels. That leads to our second research question that helps us find ways on optimizing storage approaches in order to save more energy. Answers to these research questions will help engineers come up with more sophisticated storage designs better tailored to modern smartphones and more efficient savings, affecting as many as 1.038 billion smartphone users around the globe (as of September 2012 [24]).

In order to answer the first research question, we evaluate smartphone power efficiency at various layers of the I/O path, such as the block layer and device driver. We provide evidence which highlights that the energy consumption of a smartphone can differ depending on storage techniques employed. Different scheduling algorithms on the block layer or different queue lengths on the device driver impact the total energy consumption differently. We find for 8 benchmarks the combinations of scheduling algorithms and queue lengths with optimal energy savings. In order to address the second research question, we design our SmartStorage system and implement it on the Android platform. SmartStorage tracks smartphone's I/O pattern in run-time and matches with the benchmark with closest I/O pattern. After having matched with a benchmark, the system dynamically configures an optimal storage configuration to achieve lower energy consumption.

We found a few works in the research community closely relating to ours. The work of Kim et al. [54] presents an analysis of storage performance on Android smartphones and external flash storage devices. Their discovery of a strong correlation between storage and application performance degradation serves as motivation for our work. Carroll et al. [45] measure the breakdown of energy consumption by the main hardware components in the device. Their direct measurements of each component's current and voltage are used to calculate power. This is done on a smartphone used for scientific purposes only, and many experiments cannot be replicated on commercially available smartphones. We take a different approach based on the precise analysis of the I/O activities between the

application layer and the flash storage. Our work is also motivated by cross-layer I/O analysis studied by the authors in [65, 69], which has not been done in smartphones. At this stage, there has been no direct study of the correlation between storage techniques and energy consumption within smartphone devices. We believe our work can help other researchers realize the importance of storage and perhaps trigger more exciting solutions to the smartphone energy consumption problem.

In summary, our contributions within this chapter are the following:

- First, we provide an experimental study on how storage techniques impact energy consumption on smartphones.
- Second, we design and implement the SmartStorage system that tracks I/O pattern of smartphones in run-time and dynamically configures storage parameters with optimal energy savings.
- Third, we evaluate our solution with an Android-based smartphone on the 20 top free applications from Android market and show that our system can save from 23% to 52% of energy. This is achieved with 2.5% energy overhead from running SmartStorage and a difference of 3% in terms of application delay.

3.2 Background and Motivation

In this section, we introduce background and motivation of our work. Next, we explain the measurement and methodology. Afterwards, we proceed with the measurement on the cache, the block layer, and the device driver. Finally, we give summary of our measurement results.

3.2.1 I/O Path Components

In this work we take a look at energy efficiency of various storage techniques applied at several main components such as the cache, the block layer, and the device driver.

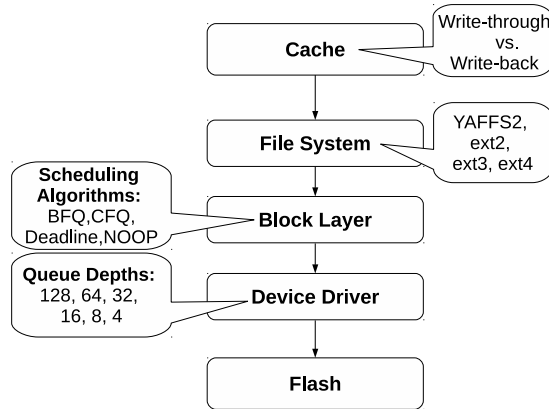


Figure 3.1: Kernel Components on the I/O Path.

In particular, our focus is to investigate the impact of different storage configurations on power level in smartphones. We illustrate the main kernel components affected by a block device operation on the I/O path in Figure 3.1. The figure is adapted from the literature [42].

3.2.1.1 Cache

The disk cache is a software mechanism allowing the system to keep in RAM some data normally stored on the disk. Further accesses to that same data can be granted without accessing the disk [42]. There are 2 classical caching policies, *write-back* and *write-through*. Write-back is the default approach used in smartphones which in practice means that the devices signal I/O completion to the operating system before data has hit the flash disk. In contrast, a write-through cache performs all write operations in parallel, with data written to the cache and the disk simultaneously. There are two mechanisms to control caching behavior of the storage devices, forced cache flush and force unit access. In the Android kernel, write-through is enforced by setting the true value to *REQ_FLUSH* and *REQ_FUA* parameters. If we want the phone to use one of the caching policies, after updating the parameters for the corresponding method in the kernel source code, the kernel needs to be rebuilt and re-flashed into the phone.

3.2.1.2 File system

There are several file system types used by smartphone vendors, each flash partition can be formatted in a different file system type before being properly mounted to given namespaces such as `/data`, `/system`, or `/cache`. Most frequently used file systems are YAFFS2, ext2, ext3, and ext4. YAFFS2 is used, for instance, in HTC Hero or Google Nexus One. Ext4 is employed in the most recent Android smartphones such as Google Nexus 4 or Samsung Galaxy S4. We can get information regarding the file systems in use by calling *mount* command on a rooted phone.

3.2.1.3 Block Layer

Block layer is another component on the I/O path. At this level, the main work is scheduling I/O requests from above and sending them down to the device driver. The Linux kernels on Android smartphones offer 4 scheduling algorithms: BFQ, CFQ, Deadline, and Noop. In BFQ (Budget Fair Queuing), each process is assigned a fraction of disk (budget) measured in number of sectors and the disk is granted to a process until the budget expires. CFQ (Complete Fair Queuing) attempts to distribute available I/O bandwidth equally among all I/O requests. The requests are placed into per-process queues where each of the queues gets a time slice allocated. Deadline algorithm attempts to guarantee a start time for a process. The queues are sorted by expiration time of processes. Noop inserts incoming I/Os into a FIFO fashion queue and implements request merging. In some Android phones, the default fixed scheduling algorithm is BFQ (Google Nexus One), others use CFQ (Samsung Galaxy Nexus, Samsung Nexus S).

3.2.1.4 Device Driver

The device driver gets requests from the block layer, and processes them before sending back a notification to the block layer. On the device driver, we are interested in a parameter called *queue depth* that is defined as the number of pending I/O requests for storage. The

queue depth is fixed to different values depending on vendors, usually 128 (e.g., Samsung Galaxy Nexus or Google Nexus One).

3.2.1.5 Flash

The last level to be reached by the I/Os is the storage subsystem that contains an internal NAND flash memory, an external SD card and a limited amount of RAM. The subsystem contains a number of partitions depending on vendors. The partitions can be found in the `/dev/block` directory.

3.2.2 Motivation

In this section, the first research question *How does storage affect smartphone energy efficiency?* is addressed by discussing preliminary measurements. We list the chosen benchmarks for our experiments and measure smartphone power level with default I/O path parameters. Afterwards, power level affected by each layer is investigated, starting with the block layer and moving further to the device driver layer. Caching policies are discussed at the end due to lower overall impact. All results are averaged over ten measurements with corresponding confidence intervals.

Benchmark	Properties
AnTuTu [3]	storage, memory, CPU, GPU
CF-Bench [6]	storage, memory, CPU, Java
GLBenchmark [7]	GPU
BrowserMark [5]	browser, JavaScript, HTML
AndroBench [2]	storage, SQLite
Quadrant [8]	storage, memory, CPU, GPU
Smartbench [9]	storage, memory, CPU, GPU
Vellamo [10]	storage, memory, CPU, GPU, browser

Table 3.1: Benchmarks

3.2.2.1 Benchmarks

We run 8 popular benchmarks on the Google Nexus One phone with Android platform under different storage configurations and measure power consumption levels with the Monsoon Power Monitor [17] (details given in Performance Evaluation). Each benchmark tests different phone subsystems and has its specific I/O pattern. The aim is to cover as many I/O pattern types as possible. The 8 chosen benchmarks with their properties are listed in Table 3.1. We do not use a synthetic benchmark that simulates I/O patterns, since we aim to use application benchmarks that reflect real Android application behavior.

3.2.2.2 Block Layer Level

The default file system, scheduling algorithm, queue depth, and caching policy for the Google Nexus One is YAFFS2, BFQ, 128, and write-back, respectively. Each benchmark is executed on the phone for each scheduling algorithm and the power level is measured. The parameters are fixed to the default values, including the queue depth 128 and write-back caching policy. The results are illustrated in Figure 3.2. The first observation says that for the same benchmark, different scheduling algorithms result in different power levels. For instance, the AnTuTu (1st benchmark) average power consumption level is 792mW with CFQ, 720mW with Deadline, 792mW with Noop, and 1080mW with the default BFQ. This is an expected outcome due to different I/O request reordering and merging of each scheduling algorithm [65]. Another observation is that none of the scheduling algorithms is optimal for all benchmarks. However, it is possible to find the optimal scheduling algorithm(s) for each benchmark and save relatively a lot of energy. For example, AnTuTu benchmark has the optimal power consumption with the Deadline algorithm, and more than 33% of energy on average can be saved compared to the default configuration with BFQ.

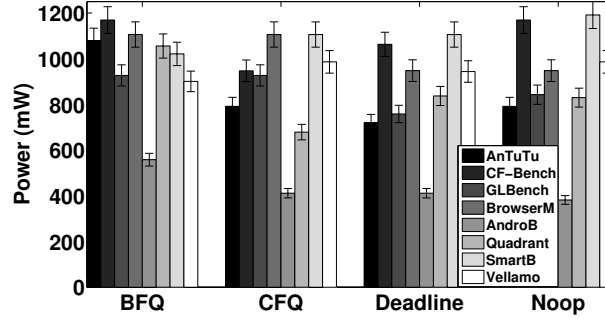


Figure 3.2: Power for Default Configurations

3.2.2.3 Device Driver Level

To investigate impact of the device driver level on energy consumption, we run the benchmarks with different queue depths and compare how different queue depths affect power levels. On Google Nexus One phone, the default queue depth is 128. The power consumption of this default queue depth is already illustrated in the previous Figure 3.2. Therefore, we investigate the power levels of the depth 4 in this section and compare with previous measurements. This way we can see the potential of how much more power efficient the system will be if we change the queue depth. Figure 3.3 shows the power levels for the depth 4 normalized to the consumptions with depth 128. Looking at AnTuTu, with BFQ and queue depth 4 (BFQ/4) the average power consumption is 720mW which corresponds to 66.7% of the default BFQ/128 consumption. That means by changing the queue depth to 4, the phone can save on average 33.3% energy. However, there are some exceptions such as in the case of Smartbench and Vellamo (last two benchmarks) that with smaller queue depth do not perform well and consume on average more power than expected. These two benchmarks are not storage intensive (Table 5.1), hence, the smaller queue causes higher overhead and as result, the higher consumption is observed.

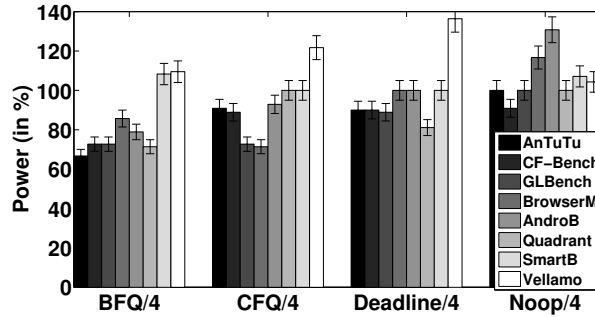


Figure 3.3: Power for Queue Depth 4

Benchmark	Reads/s	Writes/s	RP	Opt. Config.	Saving	Running Time(s)
AnTuTu	1108	1395	0.79	Deadline/4	40%	224.76
CF-Bench	104	1298	0.08	CFQ/4	27.27%	148.18
GLBenchmark	253	51	4.96	Deadline/4	27.27%	254.21
BrowserMark	185	115	1.61	CFQ/4	28.57%	278.14
AndroBench	2260	104	21.73	Noop/128	31.58%	45.1
Quadrant	301	400	0.75	BFQ/4	42.86%	129.11
Smartbench	26	2	13	BFQ/128	0	217.35
Vellamo	9	1	9	BFQ/128	0	49.82

Table 3.2: Benchmark I/O Patterns

3.2.2.4 Cache

This section attempts to find out how power level differs when using write-back and write-through caching approaches. The phone's consumption with the default caching policy (write-back) can be again determined from Figure 3.2. Hence, here for each benchmark we measure the power levels with write-through cache (scheduling and queue depth fixed). For easier reading, this is normalized to the power levels with the default write-back cache. Figure 3.4 shows that write-through caching consumes on average slightly less power. The difference is approximately 10%. This is due to limited queuing buffer space at the disk [65]. If write-back policy is in use, under heavy load the effective queues reach to the maximum allowable value, which is in our case 128. If the buffer queue is full, the device driver delays additional I/O requests. Consequently, that causes the system to slow down and consume more energy. For applications that require more reliability and consistency, write-through cache can be of help. The price for this small improvement in average power consumption requires rebuilding the Android kernel. For this reason, we decide not to include cache layer modifications into our design to provide better scalability

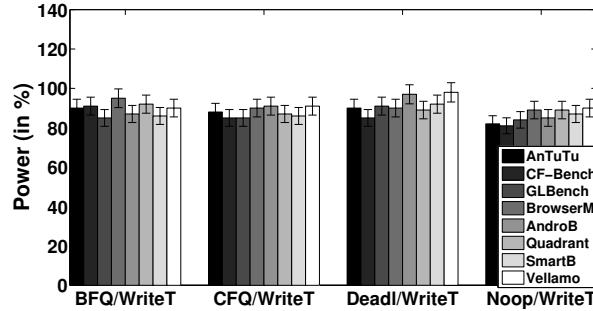


Figure 3.4: Power for Write-through Cache

and simpler deployment to ordinary users.

3.2.2.5 Optimal Consumption

In order to find optimal power consumption for all benchmarks with above knowledge, we run for each benchmark all 8 possible combinations of scheduling algorithms (BFQ, CFQ, Deadline, Noop) with queue depths (128, 4) researched. Table 5.1 shows the final combinations with optimal power consumptions for each benchmark. We can see that Quadrant consumes least power with the combination of BFQ/4, in which case it consumes 754mW and therefore, we save almost 43% compared to the default configuration (BFQ/128).

As with many other optimizations, this significant improvement in power saving has its trade-off that causes performance degradation. In particular, we observe worsen performance of the CPU, GPU, RAM, and I/O. This degradation has to be minimal for users to fully appreciate our proposed solution in the following section. For illustration, the performance scores of AnTuTu benchmark is listed in Table 3.3 for the default parameters (BFQ/128) and the parameters with optimal power consumption (Deadline/4). The higher benchmark scores, the better performance of a subsystem. For instance, the CPU performance score decreases from 958 to 946 after changing the default parameters to the optimal ones. Similarly, there is a slight GPU performance decrement from 880 to 865. This is expected due to the trade-off from the different I/O ordering and queue depth. However, the average performance degradation is only less than 2%.

Configuration	CPU	GPU	RAM	I/O
Default (BFQ/128)	958	880	317	270
Optimal (Deadline/4)	946	865	317	268

Table 3.3: AnTuTu Benchmark Performance Scores

3.3 SmartStorage Design

In order to address the second research question on how to optimize storage to save energy in smartphones, we present SmartStorage. Further in this section, an architecture is introduced and implementation details are discussed.

From previous sections, for each benchmark there exists a combination of a scheduling algorithm and queue depth that is most power efficient. This information can be reused. First, we investigate the I/O pattern of each benchmark. Next, we obtain a run-time I/O pattern from the phone and match it to a benchmark with the most similar I/O pattern. Finally, an optimal combination of a scheduling algorithm and queue depth is configured. We discuss details in the following subsections.

3.3.1 System Architecture

The architecture is illustrated in Figure 3.5. It is divided into kernel space and user space. Kernel space consists of two main modules: SmartStorage Core and Benchmark I/O Patterns. User space includes the graphics user interface (GUI) and Tools for Advanced Users. Following, we elaborate each module and its functionalities.

3.3.1.1 Kernel Space

SmartStorage Core. This module has three main functionalities. First, it obtains phone's run-time I/O pattern. Next, it gets a combination of a scheduling algorithm and queue depth with optimal power efficiency. Finally, it configures this combination in the block layer scheduler and the device driver of corresponding flash partitions.

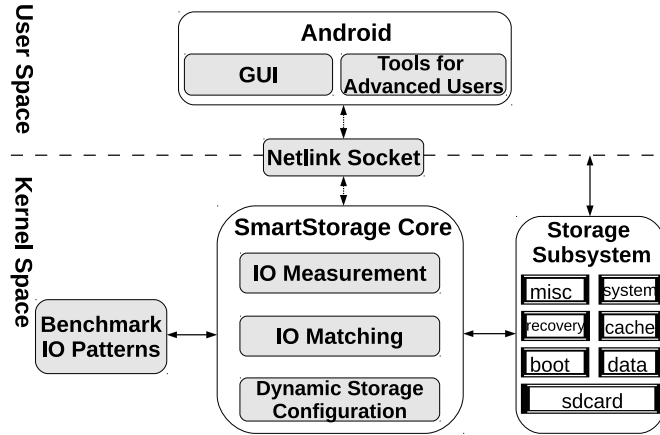


Figure 3.5: SmartStorage Architecture

The phone's run-time I/O pattern is obtained via blktrace [4, 43]. Blktrace is a block layer I/O tracing utility that provides information about request queue operations coming into storage subsystem. Blktrace is normally available in Linux distributions but it needs to be enabled in the Android system. Typically, the blktrace output includes a process ID, type of an I/O such as Read or Write, its time stamp, sequence number, etc. After gathering I/Os for a predefined time period, it calculates the run-time I/O pattern that is later used for matching with benchmark I/O patterns. The I/O pattern consists of rates of each I/O type per second. Note that such a pattern characterizes the I/Os of the whole phone, including those originating from background services. Therefore, this approach is not application-dependent.

Matching is done in the second phase after acquiring the phone's run-time I/O pattern. The phone's pattern is matched to a benchmark with the most similar I/O pattern. Since each benchmark has a combination of a scheduling algorithm and queue depth with optimal consumption, that combination is returned as a result of this phase. With power efficiency in mind, we want a computationally inexpensive matching approach that is at the same time precise. Having all types of I/Os coming to storage, simple intuition says that what matters most at the end are the total number of completed reads and number of

completed writes in a given interval. Furthermore, it is necessary to take into consideration differences between characteristics of read and write I/Os. Some partitions will serve reads better than writes or vice versa. Some partitions will be read-only, other allow both read and write. Motivated by this, we decide to expand the simple intuition, and do the matching based on the proportions of rates of completed reads and completed writes.

For clarity, let us define *RCRate* as *number of reads completed per second*. and *WCRate* as *number of writes completed per second*. Further, let us define *Rate Proportion (RP)* as $RP = RCRate / WCRate$. If the Rate Proportion (RP) of the phone's I/O pattern is close to the RP of a benchmark, a match is found. We find that this simple matching method is precise. More will be discussed in the Performance Evaluation section. Finally, the optimal scheduling algorithm is set in the block layer scheduler and the optimal queue depth is set in the device driver. This is done on all partitions.

Benchmark I/O Patterns. This includes a table with our benchmarks and their I/O patterns. See Table 5.1. Each benchmark is paired with a combination of a scheduling algorithm and queue depth with optimal power consumption obtained offline. The I/O pattern consists of rates of a specific type of I/O per second: Number of Reads completed per second and Number of Writes completed per second.

3.3.1.2 User Space

SmartStorage works naturally as a background service to save energy, without any need of interaction with users. It does not require popping up GUI or the Tools for Advanced Users. The two additional components are there only for convenience of interested users. The screen shots of the GUI and Tools for Advanced Users are included in Figure 3.6. We describe these two components below.

GUI. The graphic user interface provides the current status of the system. That includes information on which scheduling algorithm and queue depth are being used. It also informs of how long the system is being in use and how much energy has been saved.

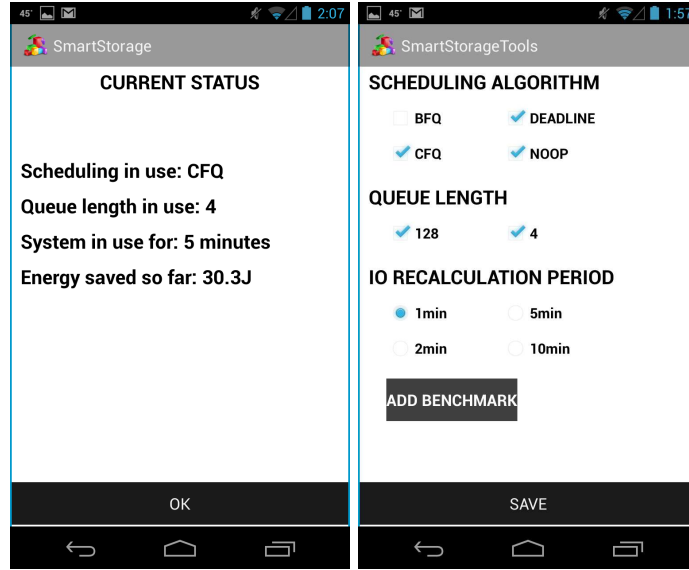


Figure 3.6: Screen shots of the GUI and Tools for Advanced Users

Tools for Advanced Users. This part is designed to serve researchers and advanced users who can possibly contribute to further project advancements. The tools let users edit after how long system should recalculate I/O pattern of the phone. It also allows setting preferred scheduling algorithms and queue lengths to be considered in dynamic configurations. With scalability in mind, it is possible to add new benchmarks to the Benchmark I/O Patterns table in the future. Finally, it communicates preferences to the SmartStorage Core in kernel space through a netlink socket [64].

3.3.2 Implementation

We implement our system on the HTC Google Nexus One smartphone with Android 2.3.7 and kernel 2.6.37.6. The implementation has 2 main parts, SmartStorage Core that is in the kernel space and the GUI that is developed as an application. As mentioned earlier, SmartStorage works naturally as a background service to save energy, without any need of interaction from the users. The additional user interface is only for the convenience of the researchers and interested users. This section highlights some of the important

implementation details of the SmartStorage Core and the GUI.

SmartStorage Core is implemented as a kernel module. It interacts with the application layer through a netlink socket. It sends up information regarding the current status, including the combination of a scheduling algorithm and queue depth in use. That scheduling algorithm is obtained by reading from a block layer scheduler. For instance, following we obtain the algorithm in use in the /data partition (mtdblock5): `cat /sys/block-/mtdblock5/queue/scheduler`. Similarly, the queue depth in use in the /data partition is obtained from the `nr_requests` (number of requests) parameter:

`cat /sys/block/mtdblock5/queue/nr_requests`.

SmartStorage calculates the phone's I/O pattern periodically each minute. After having found an optimal combination of a scheduling algorithm and queue depth via matching the phone's I/O pattern to the benchmark I/O pattern table explained earlier, the system enforces the use of the scheduling algorithm and queue depth found. For example, to set CFQ for the /data partition, the scheduler file of the block device is modified on-the-fly as follows:

`echo cfq > /sys/block/mtdblock5/queue/scheduler`. The queue depth of the /data partition can be changed on-the-fly to 4 by modifying its `nr_requests` parameter:

`echo 4 > /sys/block/mtdblock5/queue/nr_requests`.

In order to use blktrace, the support for tracing block I/O actions is enabled by changing the menuconfig file to support tracing block I/O actions. Afterwards, the makefiles are modified to include blktrace.

3.4 Performance Evaluation

This section evaluates the SmartStorage solution by a series of comprehensive experiments and answering the following questions. The first two questions are related to energy savings: (1) *How does SmartStorage save energy with a typical use case?* We address this by comparing energy usage of the 20 most popular applications from the Android

Market with and without SmartStorage. (2) *What is the overhead of SmartStorage?* We address this by measuring overhead in energy usage of SmartStorage compared to the case when it is not in use. The last two questions are related to performance issues: (3) *How precisely does SmartStorage match I/O patterns to storage configurations with optimal energy savings?* Here we show how many applications get matched with the correct combinations. (4) *Does our SmartStorage solution incur performance penalties?* This is determined using the AndroBench benchmarking tool testing throughput and I/O performance. In addition, we run ten most popular applications and evaluate their application delays.

3.4.1 Experiment Setup

In our experiments, we use the SmartStorage implementation in the HTC Google Nexus One phone. To measure energy consumption, the Monsoon Power Monitor [17] is utilized. We run the experiments with the top 20 free applications from the Android Market as of August 7, 2012. See Table 3.4. The applications include #1.Gmail, #2.YouTube, #3.Facebook, #4.Lookout Security, #5.Google Maps, #6.Twitter, #7.Tiny Flashlight + LED, #8.Yelp, #9.Amazon MP3, #10.Tango Video Calls, #11.Temple Run, #12.WhatsApp Messenger, #13.Adobe Flash Player, #14.Instagram, #15.Google Play Books, #16.Pandora Internet Radio, #17.ColorNote Notepad Notes, #18.Amazon Mobile, #19.GO SMS Pro, and #20.Voice Search. The Monsoon Power Monitor is configured by blocking the positive terminal on the phone's battery with electrical tape. The voltage normally supplied by the battery is supplied by the monitor. It records voltage and current with a sample rate of 5 kHz. During our experiments, all radio communication is disabled except for WiFi. The screen is set to stay awake mode with constant brightness and auto-rotate screen off. When SmartStorage is in use, it runs only in the background and its GUI is off.

Top 20 Apps	RP	SmartStorage Combination	Optimal Combination
#1 App	2.6	CFQ/4	CFQ/4
#2 App	0.09	CFQ/4	CFQ/4
#3 App	0.28	CFQ/4	CFQ/4
#4 App	14.7	BFQ/128	BFQ/128
#5 App	12.29	BFQ/4	BFQ/4
#6 App	4	Deadline/4	Deadline/4
#7 App	9.09	BFQ/4	BFQ/128
#8 App	0.79	Deadline/4	Deadline/4
#9 App	0.24	CFQ/4	CFQ/4
#10 App	1.36	CFQ/4	CFQ/4
#11 App	1.27	CFQ/4	CFQ/4
#12 App	0.28	CFQ/4	CFQ/4
#13 App	2.03	CFQ/4	CFQ/4
#14 App	11.2	BFQ/128	BFQ/128
#15 App	0.02	CFQ/4	CFQ/4
#16 App	2.5	CFQ/4	CFQ/4
#17 App	0.26	CFQ/4	CFQ/4
#18 App	0	CFQ/4	CFQ/4
#19 App	0.81	Deadline/4	Deadline/4
#20 App	4.94	Deadline/4	Deadline/4

Table 3.4: The 20 Applications Used in Evaluation

3.4.2 Energy Savings

As far as energy saving is concerned, it is our priority to save as much as possible, and at the same time, the system should not cause significant overhead. We show our results in this subsection.

3.4.2.1 Energy Savings

In order to address how much energy our solution saves in a typical use case, we run each of the 20 applications mentioned with SmartStorage in the background and compare with the case when the application is running with the default scheduling algorithm and queue depth (BFQ/128). A typical use case varies for applications. For instance, for Gmail, we read 20 emails and write 10 emails; for Amazon Mobile, we search for 20 products and read information about them; in Pandora, we listen to a channel for 30 minutes; on YouTube we search and listen to 5 songs; on Facebook, we read and write posts, etc.

The Android Monkey tool is utilized to allow repeating the same behavior more times with and without SmartStorage so as to ensure fairness. The results of the total savings are in Figure 3.7. We can observe that the energy savings vary from 23% to 52% and the largest savings are with Pandora application (52%). The three applications with no energy values have the optimal configuration identical to the default parameters of the phone.

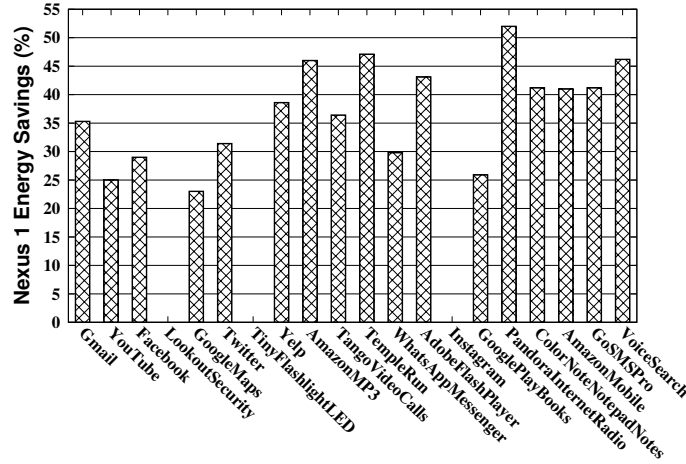


Figure 3.7: Energy Savings on Nexus One

As for validation, we also deploy our solution on the LG Google Nexus 4 smartphone with the latest Android OS 4.2.2 (Jelly Bean), kernel 3.4, and Ext4 file system. The results of the total savings on the Nexus 4 phone are in Figure 3.8. Gmail, YouTube, Facebook, and other applications with the optimal scheduling algorithm CFQ have slightly less energy savings on Nexus 4. That is expected, since the Nexus 4 phone has the default configuration CFQ/128, hence, additional savings are low.

3.4.2.2 Real-time Power Consumption

To show how SmartStorage saves energy in real-time, we conduct the following experiment. We utilize the Monkey tool to use five applications (YouTube, Pandora, Google Maps, Amazon Mobile, and Facebook) over 20 minutes (app per four minutes). This is done two times, once with SmartStorage enabled, the other time with SmartStorage disabled. Both times we make sure that the application cache is cleared before a new

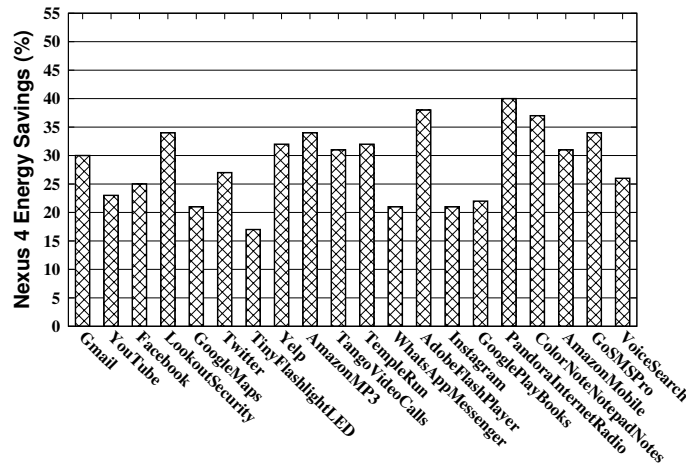


Figure 3.8: Energy Savings on Nexus 4

run. Monkey is run with the same seed, hence triggering the same series of events both times. This assures we get precise results without any delay caused by inconsistent user behavior. We use Monkey to launch YouTube and watch a video clip, the total time including both loading and watching is four minutes. After having closed YouTube (closing time negligible), we load Pandora and listen to a channel, again the total time is four minutes. Similarly, our Monkey script continues with Google Maps to find a few spots, loads Amazon Mobile to search several products, and finally uses Facebook to check the news feed.

The resulting power levels logged from the monitor are in Figure 3.9. Each four-minute segment shows data of an application. We can observe that after approximately two minutes of use of an application, the phone's power level drops. This is an expected behavior. Since the I/O pattern recalculation is done after each minute, and the loading time of an application takes a while, we should be looking for significant power drop after approximately two minutes. For YouTube (minute 0-4), the first spikes load the application, then the power reaches 2000 mW when SmartStorage is disabled. When SmartStorage is enabled, the values drop to around 50% after two minutes of use. Pandora (minute 4-8) has the power dropping after a little later than expected because an advertisement pops

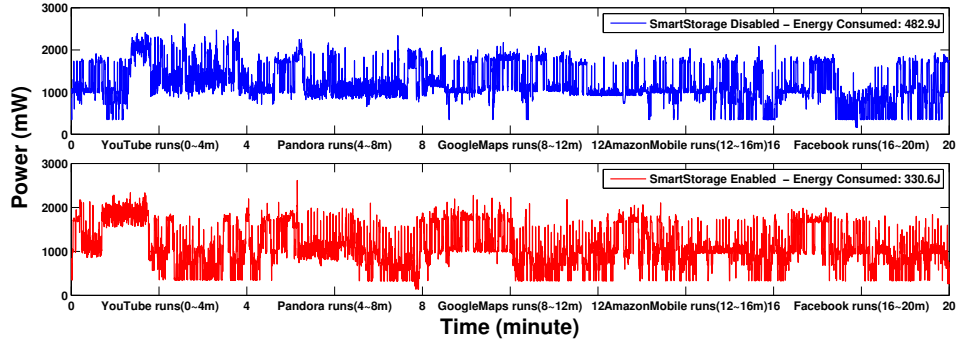


Figure 3.9: Real-time Power

up when SmartStorage is enabled. For Google Maps (minute 8-12), the number of high power spikes significantly decreases after slightly more than two minutes. When using Amazon Mobile (minute 12-16), even earlier than 14th minute, power drops to approximately 50%. Finally, Facebook (minute 16-20) shows a similar behavior, power level goes down during the 18th minute. The power levels here, however, are varying slightly more. We attribute this to the difficulty to replicate the exact behavior on Facebook that has more features that can affect the power levels than the other ones.

The total energy saving in this experiment is 152.3J which corresponds to 31.5% of the total energy consumption. In theory, according to the results in Figure 3.7, if we run each application separated, we will get 34% saving in total by averaging the savings of all five applications. The difference of 2.5% we account to the recalculation period of one minute that is not optimal. We discuss possible improvements of this approach later in Future Work.

3.4.2.3 Cost

To address the energy overhead, we run the 5 applications above separately with SmartStorage enabled and measure the energy consumption. These applications are running again with SmartStorage disabled and the measurement is repeated. Finally, the 2 con-

sumptions are subtracted and the overhead is obtained. The average overhead is 2.6% of energy consumption for Pandora, and 2.5% for other four applications. This is negligible considering the fact that with SmartStorage enabled we can save from 23% to 52% of energy consumption including the cost mentioned.

3.4.3 Performance

First, we are interested in the I/O pattern matching performance. Next, we ensure that SmartStorage incurs small performance penalties. Finally, we provide a discussion on application delays.

3.4.3.1 I/O Pattern Matching

The results of I/O matching are presented in Table 3.4. We can see the results of the final combinations of a scheduling algorithm and queue depth found by SmartStorage. In 19 cases out of 20, the matching method works correctly, which presents 95% accuracy. We note that there are 8 configurations (4 scheduling algorithm choices multiplying 2 queue length choices), each of which is the potential optimal configuration for some applications. However, the 8 selected benchmarks only cover 5 of them. Although the current I/O pattern matching method is precise for the 20 most popular applications we evaluate, in future we plan to explore a machine learning based method. More is discussed in the Future Work.

3.4.3.2 Performance Penalties

The AndroBench benchmarking tool [2] is utilized to evaluate the performance penalties of SmartStorage, since it heavily loads storage and provides flexible workload settings. The experiments are run on the Google Nexus One phone with 165MB of free space in the internal storage. We run the benchmark with four different workloads: a read-dominated, write-dominated, and a balanced workload of reads and writes of different working set

sizes (large and small). Each workload is run for the case when SmartStorage is enabled, and the case when it is disabled. The benchmark throughput in transactions per second (TPS) is measured, followed by the number of disk requests completed in each second (IOPS).

The used benchmark specifications are listed in Table 3.5. For each workload we set the read file size, write file size, and number of transactions. We aim to have significant difference between the read file size and the write file size for Read-dominated and Write-dominated workloads. In our case, one size is 16 times larger than the other one. Furthermore, we differentiate the large workload, i.e., occupying approximately 40% of free space, and the small workload, i.e., occupying a few percentage of free space. The number of transactions is set to a constant.

Workload	Read Size	Write Size	Transactions
Read-dominated	32MB	2MB	300
Write-dominated	2MB	32MB	300
Balanced Large	32MB	32MB	300
Balanced Small	2MB	2MB	300

Table 3.5: Workload Parameters

Each workload is run twice, once with SmartStorage enabled and once disabled. Figure 3.10 shows the benchmark throughput in number of transactions per second for both runs. This indicates negligible difference between the 2 cases. The throughput varies from 4% to 6%. The biggest penalties are for the case of write-dominated workload and balanced large workload.

The second part of Figure 3.10 illustrates the disk I/O performance for the case with SmartStorage enabled and the case with SmartStorage disabled. The performance in number of disk I/Os completed per second is demonstrated for all four workloads mentioned. Similar to the previous case, the performance penalties vary from 4% to 6%. The biggest penalty comes with the large workload of balanced reads and writes. This is another spot where the performance can possibly be improved. As in the previous case with

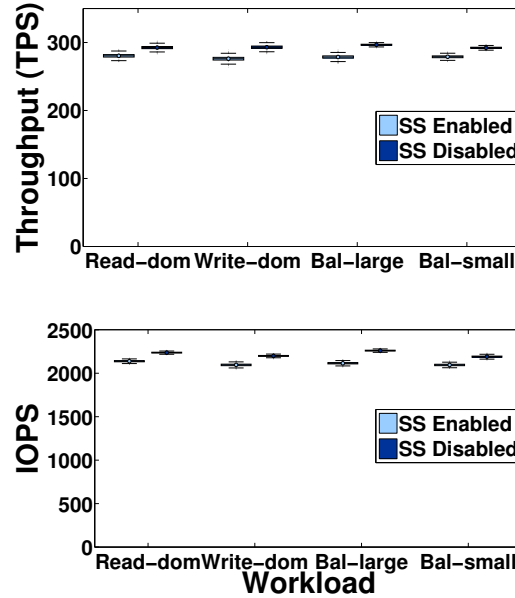


Figure 3.10: SmartStorage Throughput and I/O Performance

the number of transactions per second, the large workload seems to be problematic, this can be a key to research how to further optimize the system performance.

3.4.3.3 Application Delay

While saving energy is important, having solid performance with small application delays is equally important. In order to test delays of applications running on the phone with SmartStorage, we utilize the Android Monkey tool. Using it, we generate pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. We run the experiments with the 10 most popular Android applications on the Google Nexus One smartphone with SmartStorage enabled, and the second time with SmartStorage disabled. Each application has a predefined set of user activities triggered through the Monkey tool. We measure the time delay for both cases, when SmartStorage is enabled, and when it is disabled.

Monkey is a command-line tool that can send a stream of events into the phone's

system in a random yet repeatable manner. Each series of events we specify with the same seed value (10) in order to generate the same sequence of events. We insert a fixed delay between events (1000 ms) and adjust percentage of different types of events. We fix the number of events as a constant (500). The events are individually adjusted for each application to represent a typical usage, for instance, in Gmail we read and write an email, add a contact, change a label, etc. We run the experiments with the 10 most popular applications from Table 3.4, once with SmartStorage enabled, the next time with SmartStorage disabled. Each time we output the time delay in milliseconds and the results are illustrated in Figure 3.11. We can see that the difference is less than 3%, thus we can claim with confidence that the application delays caused by SmartStorage are negligible.

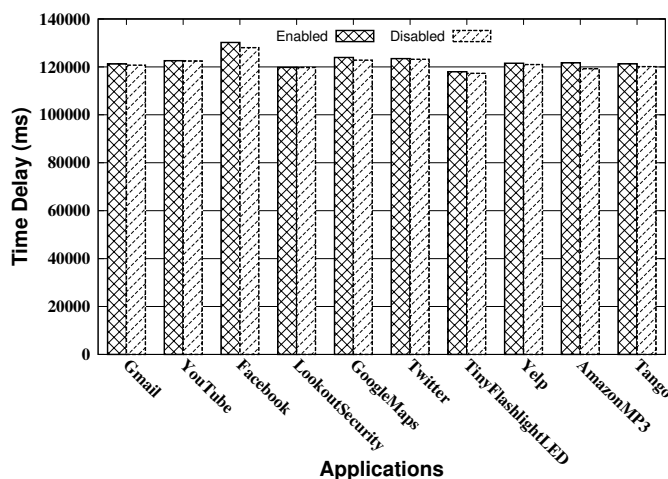


Figure 3.11: Application Delay

3.5 Conclusions and Future Work

In this chapter, we presented an experimental study of how storage parameters in the cache, device driver, and block layer affect the power levels of mobile devices running Android. In addition, we proposed a system called SmartStorage that dynamically tunes storage parameters to reduce energy consumption by matching the current I/O pattern to

a known pattern that we recorded from the eight benchmarks. Finally, we validated our dynamic tuning technique by showing that SmartStorage saved 23% to 52% of the energy consumption by running SmartStorage in the background with selected applications from the top 20 most popular apps in the foreground.

In this work, we noticed that the write-back policy consumes on average 10% more energy than write-through with the eight selected benchmarks. This could further increase our energy savings. However, this knowledge was not used in our implementation because switching from write-back to write-through or vice-versa would require rebuilding the kernel. That would be impractical for future deployment. Similarly, changing the file system and copying data around is counterproductive but could provide additional energy savings when considering the overall interaction with the remaining storage components and their parameters. In the future, we will model the trade-off between energy savings and performance degradation with write-back and write-through, and analyze how the file system interacts with the remaining components to understand and explore additional energy savings. In the device driver layer, we benchmarked the phone with two different queue depths and found significant differences in energy consumption. Naturally, more research on combining queue depths and scheduling algorithms may yield higher savings.

We proposed the *RP* metric that proved to be efficient at matching I/O patterns since what matters most is the number of writes and reads in a time interval and not the ordering of them. We plan to research the following machine learning based method in the future. In the method, each configuration is considered as a target class. We plan to collect I/O patterns from a large number of applications and label the I/O pattern of each application as its optimal configuration class. The optimal configurations of all applications should cover all 8 choices. With this data, we train either a supervised classification model or an unsupervised clustering model for run-time I/O pattern matching. Our pilot solution periodically measures the storage I/O and then matches the I/O fingerprint to that of benchmarks for locating the optimal storage policy to save energy. If this process happens too frequently, the cost may be unnecessarily high and the system may not be

stable since application performance may be impacted during highly frequent storage policy transitions, which we also plan to evaluate. If such a process happens too sparsely, we will not save much energy. Hence, we plan to monitor application events such as application started and terminated, and use them to adapt the measurement and matching frequency.

The conventional wisdom is that storage contributes little (approximately 30%) to the total energy consumption [45]. Our system with dynamic storage configurations saves from 23% to 52% of the total energy consumption. We need to emphasize that these savings are the savings of the whole smartphone, not only of the storage subsystem itself. We attribute this to the performance impact of storage on other phone components. We suspect that the interesting savings are triggered by the changes in storage, and further propagated into other components in the phone. This opens a new research question, and that is, how storage affects the performance of different smartphone subsystems. Kim et al. [54] already show how performance of smartphone applications is affected by storage performance, but do not consider energy performance. Therefore, still more research is required and we hope that our results will motivate a deeper look into this exciting area.

Chapter 4

SmartIO: Reducing Smartphone Application Delay through Read/Write Isolation

4.1 Introduction

The number of smartphones used worldwide increases each year. According to International Data Corporation, smartphone vendors shipped a total of 918.6 million smartphones in 2013, up 27.2% from the 722.4 million units shipped in 2012 [23]. With their increasing use, smartphone users tend to demand better performance. Moreover, smartphone users are increasingly using phones for work-related activities such as processing emails, reading documents, etc. A study by Forrester Research [18] found that one quarter of work devices were smartphones and tablets. Therefore, it is crucial to study application performance in smartphones. In particular, reducing the application delay can greatly improve user productivity. In addition, a recent analysis [72] indicates that most user interactions with smartphones are short. Specifically, 80% of the applications are used for less than two minutes. With such brief interactions, applications should be rapid and responsive. However, the same study reports that many apps incur significant delays (up to 10 sec-

onds) during launch and run-time.

Our study reveals that Android devices spend a significant portion of their CPU active time (up to 58%) waiting for storage I/Os to complete. This negatively affects the smartphone's overall application performance, and results in slow response time. Therefore, in order to improve the application performance, it is essential to investigate possible reasons of such waits. This chapter addresses two key research questions towards achieving rapid application response. (1) *How does disk I/O performance affect smartphone application response time?* (2) *How can we improve application performance with I/O optimization techniques?*

In order to address the first research question, we study the behavior of read and write I/Os. First, the slowdown of reads in the presence of writes is investigated. This slowdown can be one of the main reasons causing the slow launch of applications due to the dominance of reads while launching. Next, the difference in the slowdown of one I/O type due to another may require better I/O scheduling and prioritizing. Therefore, this slowdown asymmetry is researched. Finally, we look at the speedup of concurrent I/Os over serial ones. This provides insights into what type of I/Os benefit more from concurrency.

To address the second research question, we design and implement a system prototype called SmartIO on the Android platform. SmartIO measures optimal concurrency parameters for each type of I/O, and issues I/Os with the use of the obtained concurrency parameters. The system reduces the application delay by applying a set of I/O optimizations. Specifically, it assigns higher priority to reads, lower priority to writes, and groups the I/Os based on these priorities. The approach proves to have smaller performance improvement on launch delays of applications currently running in the background (warm launch). This is expected, since once an app is already in memory, its launch is much faster (on average by 65% based on our experiments). Because there is little I/O traffic going to the flash disk during warm launch, SmartIO reduces warm launch delays on average only by 6.8%. Our work focuses on reducing launch delays of applications currently

not running in the background (cold launch).

Little work in the research community directly relates to ours. Kim et al. [55] present an analysis of storage performance on Android smartphones and external storage devices. Their discovery of a strong correlation between storage and application performance degradation serves as motivation for our work. Yan et al. [72] propose a system predicting application launch using context such as user location and temporal access patterns. Their system reduces perceived delay through application prelaunching. However, the proposed system does not address the issue of slow application launch from the root, but instead lessens its impact.

In summary, the contributions of this chapter are as follows:

- First, through a large-scale measurement study based on the data collected from 2611 devices using an app we developed, we find that Android devices spend a significant portion of their CPU active time (up to 58%) waiting for storage I/Os to complete. This negatively affects the smartphone's overall application performance, and results in slow response time. Further investigation reveals that a read experiences up to 626% slowdown when blocked by a concurrent write. Additionally, the results indicate significant asymmetry in the slowdown of one I/O type due to another. While the slowdown ratio of a read is up to 6.15, the slowdown ratio of a write is only up to 1.6. Finally, we study the speedup of concurrent I/Os, and the results suggest that reads benefit more from concurrency.
- Second, we design and implement a system prototype called SmartIO that shortens the application delay by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized concurrency parameters.
- Third, we evaluate our system using 40 popular applications from four groups (games, streaming, miscellaneous, and sensing) and we show that SmartIO reduces launch delays by up to 37.8%, and run-time delays by up to 29.6%. Moreover, SmartIO also reduces power consumption by 6%.



Figure 4.1: StoreBench Storage Benchmark

4.2 Measurement Study

In order to understand how disk I/O performance affects smartphone application response time, we conduct a measurement study. First, we investigate what portion of the CPU active time is spent in storage waiting for I/Os to complete. When the time the CPUs spend in the storage subsystem is significant, this will negatively affect the smartphone's overall application performance, and result in slow response time. To identify what may be causing such waits, we learn more about I/O activities and their properties. The first property that may be a reason of such waits is I/O slowdown, which quantifies how one I/O type is slowed down due to presence of another. If one I/O activity (e.g., read) is slowed down by another (e.g., write), there will be certain cases in the application life cycle that will suffer from such slowdown (e.g., launch, since reads dominate during launch). The

impact of such slowdown on the application delay may vary depending on its ratio. This is studied in the slowdown asymmetry subsection. Another property to be researched is concurrency. Depending on hardware characteristics, different devices may benefit differently from concurrency. Therefore, in the last subsection we study the speedup of concurrent I/Os over serial ones. Finally, we discuss the measurement results and their implications.

4.2.1 Measurement Setup

In a small-scale study, a Samsung S5 phone with Android 4.4.2 is utilized. The phone is normally used daily by the first author. During measurements, our Samsung S5 has all radio communication disabled, and the screen is off. Additionally, no app is in the foreground or background, and the cache is cleared before each measurement. To verify small-scale key observations, we design and implement a storage benchmarking tool called StoreBench [20] as an Android app, and make it available for free download on Google Play [19]. StoreBench is utilized to collect data for a large-scale study.

In the large-scale study, through StoreBench we obtain data from 2611 Android devices (complete list at [21]) that installed our benchmark from Google Play (97% of the devices run Android 4.0 or higher) in the period of nine months (November, 2013 - July, 2014). StoreBench tests the I/O performance of the internal flash storage and external SD card. Specifically, the tool measures the I/O bandwidth, response time, and CPU active time spent waiting for disk I/Os to complete (*iowait*). Additionally, it measures the launch and run-time delay of 20 popular apps. With the permission of users, results are submitted to our online database for further analysis and performance ranking. Our app anonymizes all data to maintain users' privacy. Note that we do not collect or derive any data from human subjects. Instead, we only collect technical information of the devices. Therefore, no IRB approval is required in our case. The dataset of the large-scale storage performance study will be made available at [20]. StoreBench requires a rooted device

with Android 3.0 or higher, and installed BusyBox [11] on the device. The app's screenshot is in Figure 4.1.

4.2.2 Storage Contribution

To investigate what portion of the CPU active time is spent in storage, we use the *iostat* [14] shell command to output the I/O statistics of our Samsung S5 phone. The statistics from 30 days of use include detailed numbers of reads/writes of each block device in the flash disk. More importantly, the information includes the breakdown of the CPU active time spent in three domains:

- *iowait* - the percentage of time that the CPUs were idle during which the system had an outstanding disk I/O request, which simply means the time spent waiting for *disk* I/Os to complete. This does not include the wait for network I/Os.
- *user* - the percentage of CPU utilization that occurred while executing at the user level (application).
- *system* - the percentage of CPU utilization that occurred while executing at the system level (kernel).

The output of *iostat* for each domain is illustrated in Figure 4.2(a). The results show that a decent portion of time is spent in storage (19.4% of total active time), corresponding to 61.6% of system level time and 39.5% of user time. The output values observed are stable, and the standard deviation is as little as 0.1%. Note that the numbers are from the total use of all apps through the whole time period. Hence, some more I/O intensive apps can spend considerably longer than 19.4% waiting for disk I/Os to complete.

Since the measurements may be different from device to device, we also extract the *iowait* results from our large-scale study obtained through StoreBench to verify the pattern. The *iowait* empirical cumulative distribution function across 2611 Android devices is plotted in Figure 4.2(b). 40 percent of the devices have *iowait* values between 13% and

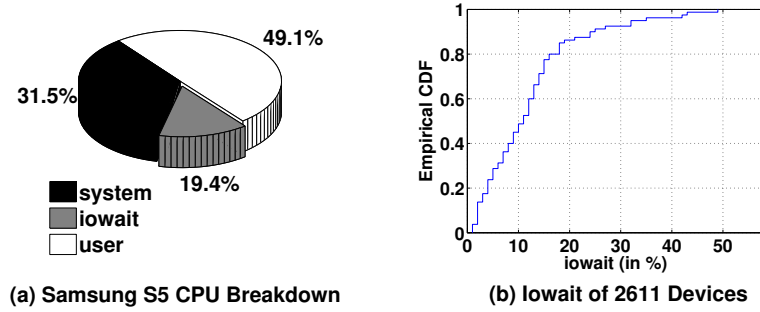


Figure 4.2: Iowait Values

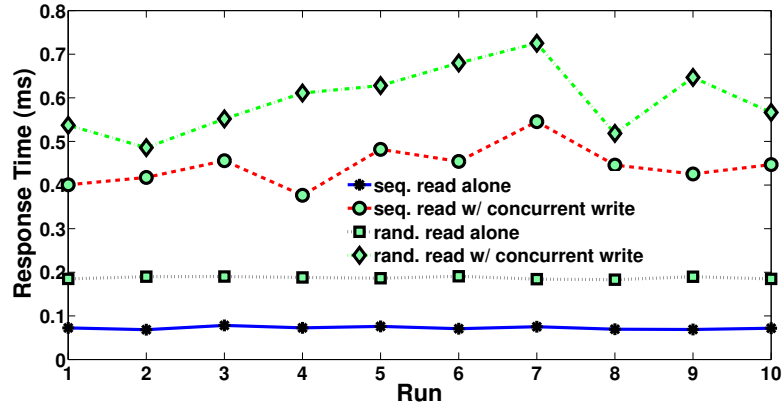
58%, which represents a significant portion of CPU active time. The averaged standard deviation is 0.1%. These results are also consistent with those of the Samsung S5.

Although the statistics vary for different devices and usage patterns, it is safe to say that CPUs in Android devices spend a significant amount of time waiting for disk I/Os. Then a following question is, what may be the main causes of such I/O waits? To answer this question, we study several important properties of Android I/O activities, including I/O slowdown, slowdown asymmetry, and concurrency.

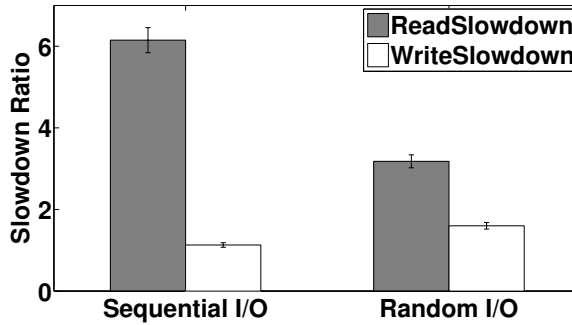
4.2.3 I/O Slowdown

In the following experiment, the goal is to understand how one I/O type is slowed down due to another, in particular, how reads are slowed down by concurrent writes. For this purpose, we utilize the Linux flexible I/O tester named *fio* [38] to issue read and write I/Os from/to the Samsung S5 phone's internal flash disk. We port *fio* to Android OS, patch the modifications to the original *fio* code, and cross-compile it. We make *fio*'s binary available for interested readers at [13].

First, we want to measure the response times of reads when they are running *alone*. We start by sequentially reading a 128MB file (32768 read I/Os, each I/O size of 4KB),



(a) Samsung S5 I/O Slowdown



(b) Samsung S5 Slowdown Asymmetry

Figure 4.3: I/O Slowdown

and calculating the average response time of a read I/O as the total response time divided by the number of I/Os. This is repeated for 10 runs. The average response time of a sequential *read when running alone* is $0.072ms$, and standard variation is 2.3%. The choice of a 128MB file is to ensure that this workload is large enough to provide statistically significant measurements but at the same time does not overwhelm the phone's storage capacity. We use this size throughout the chapter unless otherwise stated. The choice of the 4KB block size in our workloads is due to the fact that the default file system (Ext4) employed in recent Android devices utilizes this block size. Therefore, only 4KB is considered throughout this chapter, even though it has been reported that large block

sizes can improve performance [41]. Smartphone manufacturers use this small block size, since 4KB I/Os account for up to 65% of smartphone operations [58].

Next, we record the response times of reads in the presence of concurrent writes. We start by sequentially reading a 128MB file and concurrently writing a 256MB file (larger write size to assure there is concurrent write running when we read), and calculate the average response time of a read I/O. This is repeated for 10 runs. The average response time of a sequential *read in the presence of a concurrent write* is *0.445ms*, and standard variation is 3.1%. The two concurrent workloads are issued via *fiio* as two separate processes. Buffers and caches are bypassed to obtain native properties. The above experiment is repeated for random I/Os. The average response time of a random *read when running alone* is *0.187ms*, and standard variation is 3.3%. The average response time of a random *read in the presence of a concurrent write* is *0.595ms*, and standard variation is 3.7%. The results of the two experiments are illustrated in Figure 5.2(a). There are a few observations from the figure. A sequential read experiences on average 515% slowdown (6.15 times slowdown) and up to 626% slowdown when blocked by a concurrent write. Similarly, a random read experiences on average 218% (3.18 times slowdown) and up to 293% slowdown when blocked by a concurrent write. This is important since it can be one of the main sources of slow application launch, when loading data is being blocked by a concurrent write. The root cause of the slowdowns is the flash read/write speed discrepancy (reads take much faster to complete). Additionally, reads become less predictable and the response times vary significantly over runs in the presence of a concurrent write.

Finally, we can observe that random reads are about 2.6 times slower than sequential reads. Although there is no seek time as in conventional rotating storage, random I/Os still suffer from processing overhead. When random I/O requests are issued, the CPUs have to coalesce the requests, and the storage controller has to interpret and pass them down to the correct block device, where a proper ordering is determined. Moreover, random file operations often involve file table access, which adds additional delay.

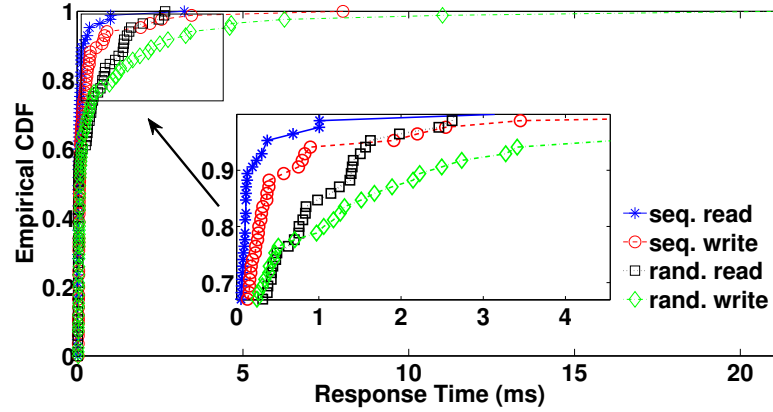


Figure 4.4: Response Time ECDF of 2611 Devices

4.2.4 Slowdown Asymmetry

The next property that may affect I/O performance (and *iowait* as a result) is slowdown asymmetry. In the following we compare the average slowdown ratio of a read and a write. The slowdown ratios are calculated as follows:

- $ReadSlowdown = \frac{\text{Response time of a read in the presence of a concurrent write}}{\text{Response time of a read when running alone}}$
- $WriteSlowdown = \frac{\text{Response time of a write in the presence of a concurrent read}}{\text{Response time of a write when running alone}}$

The Samsung S5 results for both sequential and random I/Os with standard deviations are displayed in Figure 5.2(b). For sequential I/Os, while the slowdown ratio of a read is 6.15, the slowdown ratio of a write is only 1.13. For random I/Os, while the slowdown ratio of a read is 3.18, the slowdown ratio of a write is only 1.6. This large asymmetry in the slowdowns has a following reason. Writes in the flash storage take already significantly longer than reads, hence, there is a smaller impact of the slowdown. While the response time of a sequential write running alone is on average 0.19ms, a sequential read running alone takes only 0.072ms. While the response time of a random write running alone is on average 0.41ms, a random read running alone takes only 0.187ms.

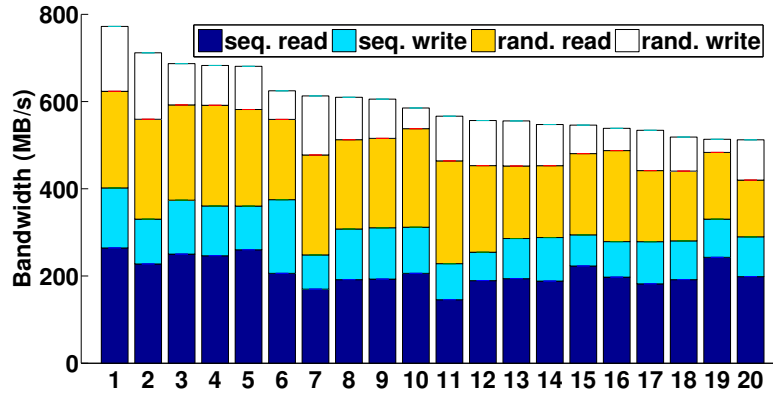


Figure 4.5: Storage Performance of Top 20 Models

To understand the trend in the large scale, we plot the response time distributions obtained via StoreBench benchmark in Figure 4.4. In general, writes take longer than reads, and random I/Os take longer than sequential ones. This is consistent with the small-scale study using Samsung S5.

We also add Figure 4.5 with storage performance ranking obtained from the devices submitted by our users. Specifically, the figure includes the total bandwidth of the top 20 devices in MB/s. If a model has more devices in the ranking, then it is represented by its top device. An interesting observation is that a more recent model does not necessary mean higher ranking. For instance, while Nexus 5 (2013) tops the whole chart, Nexus 6 (2014) only occupies the 3rd place. Nexus 5 manufactured by LG mainly dominates thanks to its strong random write performance. The devices are following. **1:**LG Nexus 5; **2:**OnePlus One (A0001); **3:**Motorola Nexus 6; **4:**Bq Aquaris E10; **5:**Motorola Moto G; **6:**Samsung Galaxy Note 2 (GT-N7100); **7:**Sony Xperia Z Ultra (XL39h); **8:**Samsung Galaxy S3 (GT-I9300); **9:**LG G2 (LG-D800); **10:**Nubia Z7 Max (NX505J); **11:**Sony Xperia Z1 (C6903); **12:**Samsung Galaxy Note 3 (SM-N9002); **13:**Asus Nexus 7; **14:**Sony Xperia Z2 (D6503); **15:**LG L70 (LG-D321); **16:**Lenovo A328; **17:**Hisilicon Hi3798CV100; **18:**LG Optimus F6 (LGMS500); **19:**HTC One M8; **20:**LG G3 (LG-D850).

4.2.5 Concurrency

The next property that may affect I/O performance (and *iowait* as a result) is concurrency. An obvious approach to speeding up the application response is to issue I/Os concurrently. However, a large number of concurrent I/Os may overwhelm the processing capacity, and thus cause performance degradation. Therefore, it is necessary to find a sweet spot in concurrency to achieve maximal speedup. The last experiment's goal is to study the speedup of the concurrent I/Os over serial ones in the Samsung S5 phone. This is done for reads and writes separately. First, we issue two serial reads, each of size 32MB, and record the total response time. Then we issue two concurrent reads, each of size 32MB, and record the response time (use the larger result of the two reads if they differ). The speedup is calculated as the ratio of the two response times (serial / concurrent). This is repeated with four reads, eight reads, 16 reads, and 32 reads, respectively. The choice of smaller workloads in this section (32MB) is because we issue up to 32 of such workloads concurrently, and do not want to overwhelm the phone's storage capacity.

To see how writes benefit from concurrency, we repeat the above with writes. First, two serial writes are issued, each of size 32MB, and the total response time is recorded. Then we issue two concurrent writes, each of size 32MB, and record the response time (use the larger result of the two writes if they differ). The speedup is calculated as the ratio of the two response times. This is again repeated with four writes, eight writes, 16 writes, and 32 writes, respectively. The speedup of concurrent I/Os over serial I/Os is illustrated in Figure 4.6.

We obtain four concurrency parameters from the figure. The number of concurrent sequential reads with maximal speedup (1.45) is 2, and the number of concurrent sequential writes with maximal speedup (1.29) is 4. The number of concurrent random reads with maximal speedup (1.55) is 4, and the number of concurrent random writes with maximal speedup (1.41) is 2. The speedup of reads is higher than the one of writes for both cases, which implies that reads benefit more from concurrency. This is expected. Intuitively,

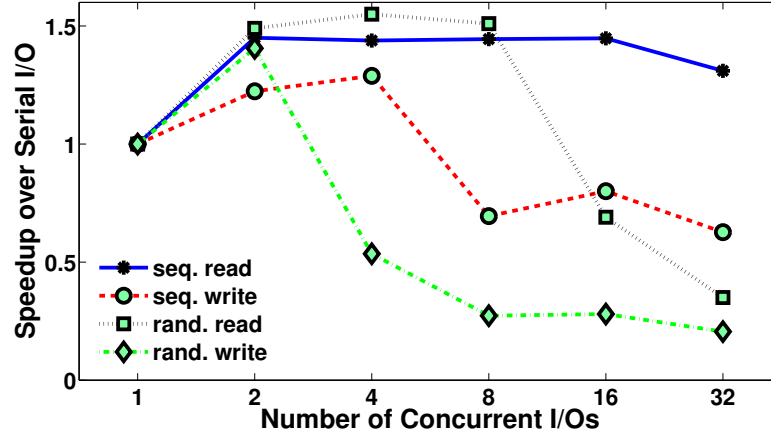


Figure 4.6: Samsung S5 Speedup over Serial I/O

with growing processing time, the wait time also increases. Moreover, if the processing needs exceed the processing capacity, then there is no well-defined average waiting time because the queue can grow without bound. Since writes take longer to process than reads, it is expected that writes would overwhelm the processing capacity sooner, and thus benefit less from increased concurrency. In addition, different devices may benefit differently from concurrency, since they may have different speedup represented by concurrency parameters. Since these concurrency parameters may differ for various devices, a solution with the maximum benefits from concurrency requires a design that is capable of adapting to each phone's concurrency characteristics.

4.2.6 Summary

The above experiments lead to several important observations that shed light on how to improve smartphone application performance, and we summarize them below.

First, Android devices spend a significant portion of their CPU active time waiting for storage I/Os to complete. Specifically, 40% of the devices have *iowait* values between 13% and 58%. This negatively affects the smartphone's overall application performance, and results in slow response time. Therefore, in order to improve the application perfor-

mance, it is essential to investigate possible causes of such waits.

One of the reasons causing such waits is I/O slowdown. Our first experiment studies slowdown of one I/O type due to presence of another, and reveals significant slowdown of reads in the presence of writes. Specifically, a sequential read experiences on average 515% slowdown and up to 626% slowdown when blocked by a concurrent write. Similarly, a random read experiences on average 218% and up to 293% slowdown when blocked by a concurrent write. This significant read slowdown may negatively impact the application performance during the life cycles when the number of reads dominates. A good example is application launch.

Next, the impact of such slowdown on the application delay may vary depending on the slowdown ratio of a read and a write. As demonstrated earlier, there is a significant asymmetry in read and write I/O slowdown. Specifically, for sequential I/Os, while the read slowdown ratio is 6.15, the write slowdown ratio is only 1.13. For random I/Os, while the read slowdown ratio is 3.18, the write slowdown ratio is only 1.6.

Finally, the last property researched is concurrency. Our experimental study reveals that different devices may benefit differently from concurrency. The above results also suggest that reads benefit more from concurrency. However, in order to optimize the application performance, we need to be able to adapt to the concurrency characteristics of each device. Such characteristics include four concurrency parameters of the maximal speedup: the number of concurrent sequential reads, the number of concurrent sequential writes, the number of concurrent random reads, and the number of concurrent random writes.

4.3 System Architecture

In order to improve the application delay performance in smartphones, we present SmartIO, a system that reduces the application response time by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized con-

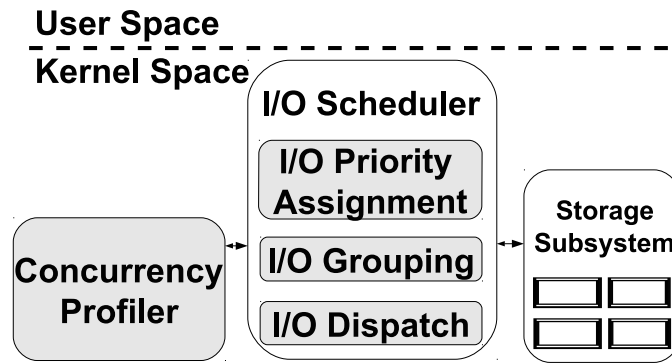


Figure 4.7: SmartIO

currency parameters. The architecture of SmartIO is illustrated in Figure 4.7. It is fully located in the kernel space, and consists of two main modules: the I/O Scheduler and the Concurrency Profiler. The I/O Scheduler encapsulates 3 submodules: I/O Priority Assignment, I/O Grouping, and I/O Dispatch. We elaborate each module and its functionalities below.

I/O Priority Assignment. Our system prototype follows the implications from the previous experimental study. First, since a read suffers a large slowdown in the presence of a concurrent write, the goal is to allow reads to be completed before writes, and delay writes as long as there are reads, while avoiding write starvation. In order to achieve this, a third level of I/O priority is added into the current block layer, assigning higher priority to reads and lower to writes. This third priority level has a lower priority than the first two priority levels (class priority, and priority within each class) from the block layer explained earlier in the Background section. Write starvation is avoided by applying a time slice, which is a maximal period of time assigned to a process, and is by default 100ms as used in the Linux scheduler time slice concept.

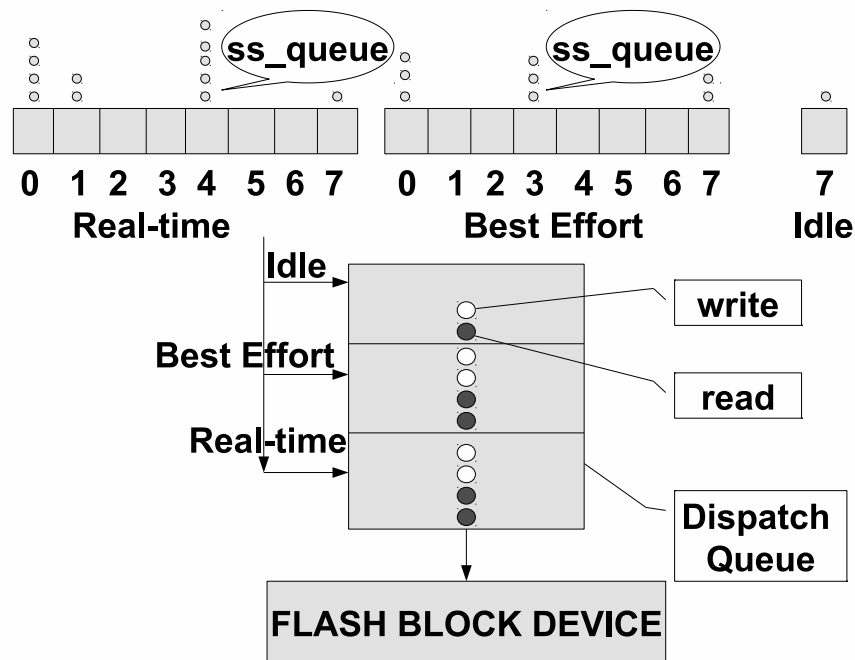


Figure 4.8: Dispatch Example

I/O Grouping. The dispatch queue further groups reads and groups writes based on the three levels of priority. Reads are ordered in front of writes, and reads are then dispatched before writes. Due to the read/write discrepancy nature of the flash storage (reads take much faster to complete), the read-preference reordering does not introduce a major delay to write I/Os.

This reordering enforced by SmartIO does not affect correctness and semantics of write barriers. It is common knowledge that write barriers are essential for consistency of many file systems. That is, however, maintained at the file system layer, which is above the I/O scheduler. Therefore, requests issued to an I/O scheduler can be reordered without affecting correctness. In fact, reordering is a common practice to minimize the seek costs in mechanical disks.

I/O Dispatch. A sample dispatch is illustrated in Figure 4.8. In the current CFQ imple-

mentation, each block device has 17 queues (`ss_queue`) of I/O requests (8 Real-time, 8 Best Effort, and 1 Idle). The existing system selects a queue based on the priorities, takes a request in the queue, and inserts it in the dispatch queue. The queue selection process accounts for two priority levels: the class priority (Real-time, Best Effort, Idle), and the priority within the class (0-7).

Our system does not change the above dispatch process but uses a third priority level to organize the dispatch queue in favor of the read I/Os. The dispatch queue is then divided into three sections, from the bottom up real-time, best effort, and idle requests. Each section is organized such that reads precede writes.

Concurrency Profiler. The system uses the knowledge of the phone's four concurrency parameters to issue the I/Os to the block device. The parameters include the optimal number of sequential or random reads (writes) that benefit most from concurrency, as discussed earlier in the Concurrency subsection. Based on the parameters, the system issues the appropriate number of reads (writes) concurrently from the dispatch queue. To achieve this, SmartIO measures the concurrency parameters during installation by invoking the *fio* tool to benchmark the phone. *fio* issues reads and writes, and calculates the speedup of concurrent I/Os over serial ones, as performed in the measurement study. The concurrency parameters with optimal speedup are then used to complete the I/O requests. This assures robustness of our system to different characteristics of the flash storage in the phones. With the use of *fio*, SmartIO can adapt to different devices without prior knowledge of their concurrency parameters.

4.4 Implementation

In this section, we elaborate implementation details of the SmartIO system. In particular, we explain the algorithm of the scheduler's dispatch process. Next, we highlight important implementation challenges of the SmartIO system. Specifically, we discuss the I/O

testing tool integration in the Concurrency Profiler module. The module utilizes the tool to obtain optimized concurrency parameters that allow SmartIO issue optimal number of I/Os concurrently to block devices.

SmartIO. First, we discuss implementation details of our solution. We implement the SmartIO system on the rooted Samsung S5 smartphone with Android 4.4.2 (KitKat), kernel 3.10, and Ext4 file system. The phone is equipped with a 2.5 GHz quad-core Krait 400 CPU, 2 GB of RAM, and 16 GB of internal flash storage. The implementation consists of 2 main modules, the I/O Scheduler and the Concurrency Profiler, both of which are in the kernel space.

The I/O Scheduler is implemented as a kernel patch of the default CFQ Linux scheduler. Users can switch to our scheduler with a simple shell command that changes the scheduler file. For instance, the scheduler is set on all block devices on-the-fly as follows: `echo ss > /sys/block/mmcblk0/queue/scheduler`. Similarly, the users can go back to the default scheduler by:

`echo cfq > /sys/block/mmcblk0/queue/scheduler`.

Details of the dispatch are explained below. First, the system selects a queue from the 17 priority queues, then chooses a request in the selected queue, and inserts the request into the dispatch queue. If the time slice of the current queue is not expired (default 100ms as in CFQ), and the queue is not empty, the dispatch continues with the current queue. Otherwise, it chooses a different queue based on the priorities. The time slice serves as an ultimate mechanism to avoid starvation. When a queue q is chosen, the algorithm dispatches a request from it. If it may dispatch, it picks a request from the queue in the FIFO fashion, and inserts the request into the dispatch queue. The *dispatch* is elaborated in Algorithm 4.4.1.

To find out if we can dispatch from a queue q , *may_dispatch* is invoked. First, it checks whether the queue has more I/Os in flight than allowed. If not, it allows the dispatch. If the queue has already reached the dispatch limit, the system checks how many queues

Algorithm 4.4.1: Dispatch(*queue* * *q*)

```
//choose a queue
if current queue q empty or its time slice expired
then choose another queue and assign it to q

//if queue q may dispatch
if may_dispatch(q)
then { pick a request in FIFO fashion
      insert request to dispatch queue
```

Algorithm 4.4.2: may_dispatch(*queue* * *q*)

```
//does this q already have too many I/Os in-flight?
if (q.dispatched >= max_dispatch)
    if (busy_queues > 1)
        then { //we have other queues, don't allow more
              //I/Os from this one
              return (false)
        }
    then { else if (busy_queues == 1)
          then { //sole queue user, no limit
                max_dispatch ← ∞
              }
          else { max_dispatch ← quantum
                //default init quantum is 8
              }
    }

//if we're below the current max, allow dispatch
return (q.dispatched < max_dispatch)
```

are waiting for dispatch. In case when there is another queue waiting, the dispatch is not allowed. If the queue is the only one, SmartIO sets no limit for it. The number of in-flight I/Os of a queue from the Linux default settings is 8. *may_dispatch* is elaborated in Algorithm 4.4.2.

Obtaining Concurrency Parameters. As discussed earlier, based on the concurrency parameters, SmartIO issues the appropriate number of reads (writes) concurrently from the dispatch queue. To achieve this, SmartIO measures the concurrency parameters during installation by invoking the *fio* tool to benchmark the phone. *fio* issues reads and writes, and calculates the speedup of concurrent I/Os over serial ones, as performed in the measurement study. *fio* [38] is a Linux I/O testing tool that directs different types of I/Os to block devices, and returns information on the delay performance. The first step to

get *fio* issue a desired workload is to write a job file. The typical contents of the job file is a global section defining shared parameters, and one or more job sections describing the jobs involved. For instance, the following code tests the sequential read and write performance of the /data partition on a phone:

```
[ global ]
directory=/data
bs=4k
size=32m

[ sequential-read ]
rw=read
numjobs=1
stonewall

[ sequential-write ]
rw=write
numjobs=1
stonewall
```

Stonewall allows a job to start only when a previous one has finished. Without the two *stonewalls* above, the tool issues two jobs running concurrently. The *directory* defines the destination for the workload, *bs* stands for block size, and *size* defines the size of the workload to be issued.

To integrate *fio* in SmartIO, we patch the *fio* code with Android compiling adjustment, and cross-compile it to get its binary. We make the binary and job files available at [13]. The binary then is imported into the Concurrency Profiler module, and in run-time transferred to the /data partition directory in the internal flash disk.

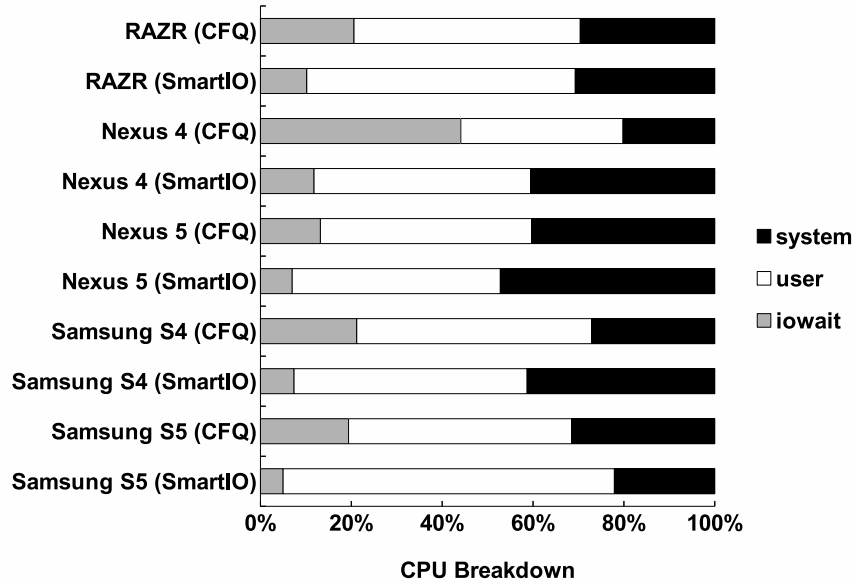


Figure 4.9: iowait Before and After

4.5 Performance Evaluation

This section evaluates SmartIO, and answers the following questions. (1) *How does SmartIO reduce iowait?* We output *iostat* values of five smartphones with SmartIO. (2) *How does SmartIO improve the benchmark performance?* We address this by investigating the I/O slowdown and asymmetry of the synthetic benchmarks. The experiments are conducted with SmartIO disabled, and enabled. Additionally, SmartIO is compared with other existing I/O schedulers. (3) *How does SmartIO improve the application performance?* This is addressed by recording the launch and run-time delay of the 40 popular apps from Google Play with and without SmartIO. In addition, we conduct an experiment on the Facebook application to determine the user-perceived performance improvement of our solution.

4.5.1 iowait

As in the measurement study, we utilize the *iostat* [14] shell command to output the I/O statistics of five devices: Samsung S5, Samsung S4, Nexus 5, Nexus 4, and Motorola RAZR Maxx. The devices are normally used daily by the authors, and are running Android 4.4, 4.3, 4.4, 4.2, and 4.0, respectively. The statistics from the use of SmartIO within 30 days and the use of CFQ within 30 days are illustrated in Figure 4.9. The results indicate a significant iowait reduction on Samsung S5 (74.2%) and Nexus 4 (73.2%). These numbers highly depend on the individual I/O traffic resulted from usage patterns of each smartphone user. In particular, Samsung S5 and Nexus 4 have both the total amount of blocks read almost an order of magnitude larger than the amount of blocks written (10,122,938 vs. 1,017,864; 250,005,743 vs. 26,042,265; each block of 4KB). This read intensive traffic benefits from our solution that favors reads over writes, which contributes to the reduction of the CPU time the devices spend waiting for I/Os to complete. The other devices also show a decent reduction in iowait: 65.1% (Samsung S4), RAZR (50.5%), and Nexus 5 (47%).

4.5.2 Benchmark Performance

To determine SmartIO's performance gain and cost, we investigate the I/O slowdown and asymmetry of benchmarks. Since the proposed system is designed to serve in favor of reads over writes, writes are expected to perform slightly worse. We run two benchmarks, first with SmartIO disabled, and the second time with SmartIO enabled. When SmartIO is disabled, the default I/O scheduler (CFQ) is utilized. The first benchmark consists of an 1-reader (128MB) and an 1-writer (128MB) process. The second benchmark consists of a 4-reader (4 x 128MB) and a 4-writer (4 x 128MB) process. We consider both sequential and random I/Os. First, the experiment is done on the Samsung S5 phone. The I/Os are issued by the *fio* tool.

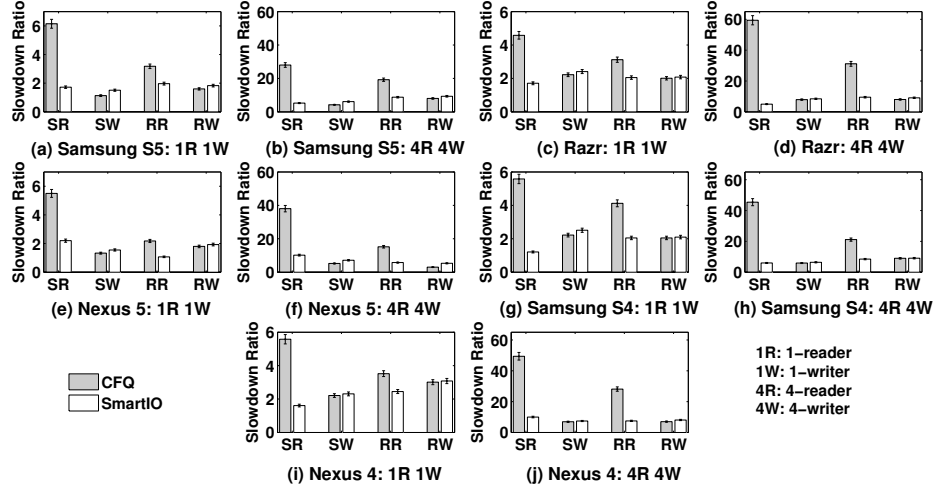


Figure 4.10: I/O Slowdown

Gain vs. Cost.

The I/O slowdown of the 1-reader and 1-writer with standard deviations is illustrated in Figure 4.10(a), where SR=sequential read, SW=sequential write, RR=random read, and RW=random write. For sequential I/Os, the read slowdown improves from 6.15 (CFQ) to 1.72 (SmartIO). Since our system delays writes in favor of reads, it is important to make sure that writes do not suffer a large performance degradation. As observed, this read performance improvement comes with only little cost due to the read/write discrepancy nature of the flash storage (reads take much faster to complete). Specifically, the write slowdown ratio worsens from 1.13 to 1.51. Similar behavior is observed for the random I/Os. While the read slowdown ratio improves significantly from 3.18 to 1.97, the write slowdown worsens slightly from 1.6 to 1.83. However, the random reads achieve smaller performance gain than the sequential ones. This is consistent with the results from the Measurement Study (Section 5.2), which show the random reads having lower slowdowns in the presence of the concurrent writes, hence, the benefit from the SmartIO read-preference scheduling is smaller.

The I/O slowdown of the 4-reader and 4-writer is illustrated in Figure 4.10(b). For

sequential I/Os, the read slowdown ratio improves dramatically from 28.03 to 5.12. This large performance gain comes from the read-preference of SmartIO, together with the speedup from improved concurrency. The write slowdown ratio worsens from 4.21 to 6.12, which is the cost of SmartIO's lower write's priority. The random read slowdown improves from 19.22 to 8.75, while the write slowdown worsens from 8.01 to 9.32. Again, the random I/Os benefit from SmartIO slightly less than the sequential I/Os, which agrees with the theory.

Adaptation to Different Phones. As for validation, we also deploy our solution on other phones. First, we look at the Motorola Razr smartphone with the Android OS 4.0 (ICS), kernel 3.0, Ext4 file system, and duo-core. The Razr's default I/O scheduler is also CFQ, and its four concurrency parameters with maximal speedup found by SmartIO are: 2 concurrent seq. reads, 2 concurrent seq. writes (different from Samsung S5), 2 concurrent random reads (different from Samsung S5), and 2 concurrent random writes. The I/O slowdown of the 1-reader and 1-writer is illustrated in Figure 4.10(c). The I/O slowdown of the 4-reader and 4-writer is illustrated in Figure 4.10(d). Both figures are plotted with standard deviations. The 1-reader and 1-writer shows a similar behavior as on the Samsung S5 phone. The 4-reader and 4-writer indicates even larger read performance improvement compared to the Samsung S5 phone. The sequential read slowdown ratio improves from 59.4 to 4.98, while its write slowdown only worsens from 7.92 to 8.41. The random I/Os also show great improvement, the read slowdown improves from 31.12 to 9.5, while the write worsens from 8.01 to 9.08. This large performance boost is due to higher gains from concurrency, and demonstrates that SmartIO with its concurrency parameters measurement can adapt to different flash characteristics. The Samsung S5's smaller performance gain is due to the fact that the phone is more recent, and its four cores already offer great baseline performance. While the Razr's duo-core architecture shows even larger read performance improvement due to the lower baseline performance of the smaller number of cores. For comparison, we also display further results on the rest

of the devices: Nexus 5 in Figure 4.10(e)(f), Samsung S4 in Figure 4.10(g)(h), and Nexus 4 in Figure 4.10(i)(j). They all demonstrate significant reductions in the read slowdown, while the write slowdown only worsens little. From these three devices, Samsung S4 has the largest read slowdown reduction (7.6 times in (h)), while Nexus 5 has the largest write slowdown increment (1.77 times in (f)). In summary, the above benchmarking experiments show different performance gains for a diverse set of devices. This is reasonable, since each device is equipped with different hardware components, and hence different results are expected. However, the experiments also confirm that SmartIO is able to adapt to different phones.

4.5.3 Scheduler Comparison

This section aims to compare SmartIO with other existing I/O schedulers: Complete Fair Queuing (CFQ), Deadline, and Noop. These are the only three schedulers available on recent Android devices. CFQ attempts to distribute available I/O bandwidth equally among all I/O requests. The requests are placed into per-process queues where each of the queues gets a time slice allocated. Further details on CFQ are explained earlier in the Background section. Deadline algorithm attempts to guarantee a start time for a process. The queues are sorted by expiration time of processes. Noop inserts incoming I/Os into a FIFO fashion queue and implements request merging.

To compare the schedulers, we utilize *fio* to issue mixed workloads of both reads and writes to the Samsung S5 phone's internal flash disk, and measure the time delay that takes to complete the workloads (response time). This is repeated on all mentioned schedulers, and the comparison is done for both sequential and random I/Os.

Sequential I/O. For each scheduler we issue a 128MB mixed workload with 10% of sequential reads (90% of sequential writes), and record the response time. Next, we issue a

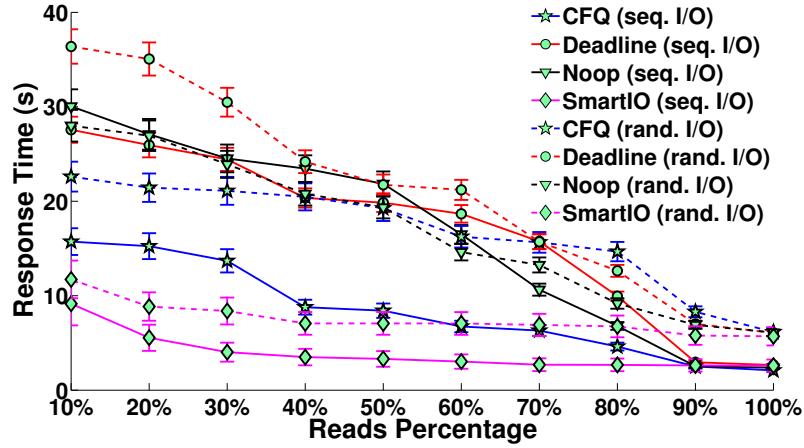


Figure 4.11: Scheduler Comparison

128MB mixed workload with 20% of reads (80% of writes), and record the response time. We continue issuing a workload with 30% reads, 40% reads, etc. Until the workload with 100% reads. The block size is set to 4KB, the queue depth to 128, and the cache is cleared after each measurement.

The resulting response times are plotted in Figure 4.11 (*solid lines*). In general, for all four schedulers, with the increased percentage of reads, the response time decreases. For instance, with a workload consisting 10% reads, the response time for SmartIO is 9 seconds, CFQ 16 seconds, Deadline 28 seconds, and Noop 30 seconds. With 50% of reads, the response time is faster, SmartIO needs 3 seconds, CFQ 8 seconds, Deadline 20 seconds, and Noop 22 seconds. This is consistent with our measurement study, since reads are faster to complete, and less writes also means smaller I/O slowdown. For most workloads, SmartIO provides the fastest response time, while the current I/O scheduler in Samsung S5 (CFQ) is second best. Deadline and Noop perform poorly, and one beats another depending on the workload. Consequently, by changing the scheduler from the default CFQ to the proposed SmartIO, we achieve on average 42% faster response times (max of 64%).

Random I/O. The above experiment is reiterated for random I/Os. The resulting response

times are plotted in Figure 4.11 (*dashed lines*). Again, it is safe to say that with the increased percentage of reads, the response time decreases for all schedulers. This is consistent with our experimental study, since reads are faster to complete, and less writes also means smaller I/O slowdown. For all random I/O workloads, SmartIO has fastest response times. As a result, by changing the scheduler from the default CFQ to the proposed SmartIO, we may achieve on average 49% faster response times (max of 66%). Compared to sequential I/Os, random I/Os take longer to complete. This is also consistent with our findings in the measurement study, which identifies that random activities generally take longer to complete.

4.5.4 Application Performance

To address the third question on how SmartIO improves the application performance, we measure the launch and run-time delay of 40 popular apps (10 games, 10 streaming, 10 miscellaneous, and 10 sensing) from Google Play, with and without SmartIO. Among others, the miscellaneous group also includes two file processing applications (File Commander and File Manager) and two write-intensive applications (ZArchiver and RAR for Android). During the experiment, our Samsung S5 has all radio communication disabled except for WiFi that is necessary to provide stable Internet connections required on most apps. The screen is set to stay-awake mode with constant brightness, and the screen auto-rotation is disabled. Only one app runs at a time, and no other app is in the background. This is to achieve a fair comparison between the two cases: with SmartIO, and without SmartIO. The cache is cleared before each measurement in order to evaluate real performance improvement caused by SmartIO.

Launch Delay. The Android Monkey tool [16] is utilized to trigger the launch process of each app. The application *launch delay* starts when the launch process is triggered, and ends when the process completes. The launch delay includes three components. We

use the *time* command [22] to output the three time components: the time taken by the app in the user mode (*user*), the time taken by the app in the kernel mode (*system*), and the time the app spends waiting for the disk and network I/Os to complete (*totalIO*). The storage I/O delay is obtained by dividing the total number of I/Os completed (*kBread* + *kBwrtn*) over the total rate of I/Os completed (*kBreadRate* + *kBwrtnRate*) in a flash block device. The network I/O delay is then calculated as the total I/O delay (*totalIO*) subtracted by the storage I/O delay (*storageIOdelay*).

Formally,

$$storageIOdelay = \frac{kBread + kBwrtn}{kBreadRate + kBwrtnRate}, \quad (4.1)$$

where *kBread* is the amount of data read from a flash block device, *kBwrtn* is the amount of data written to a flash block device, *kBreadRate* is the data rate read per second from a flash block device, and *kBwrtnRate* is the data rate written per second to a flash block device. All four variables are obtained from the output of the *iostat* Linux command.

$$networkIOdelay = totalIO - storageIOdelay, \quad (4.2)$$

where *totalIO* is the time an app spends waiting for both disk and network I/Os to complete. The variable is obtained from the *time* command during application launch.

The *cold launch delay* is a launch delay required to launch an application not currently running in the background. Such application also has its cache cleared before each measurement. The cold launch delay of the 40 apps with and without SmartIO is illustrated in Figure 4.12(a). The figure includes 10 games (1-5, 21-25), 10 streaming apps (6-10, 26-30), 10 miscellaneous apps (11-15, 31-35), and 10 sensing apps (16-20, 36-40). Applications running with SmartIO are denoted with a star (*). The figure is plotted with standard deviations. The applications are following. **1:**Angry Birds; **2:**GTA;

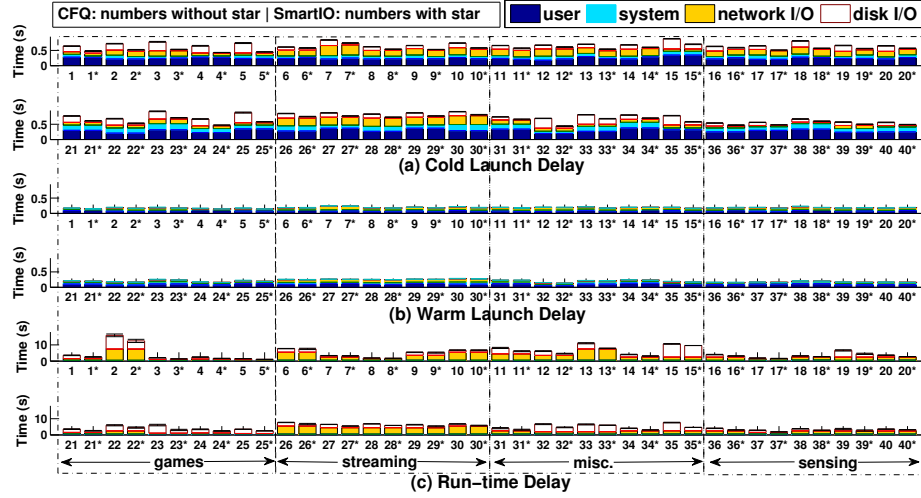


Figure 4.12: Launch and Run-time Delay

3:Need for Speed; **4:**Temple Run; **5:**The Simpsons; **6:**CNN; **7:**Nightly News; **8:**ABC News; **9:**YouTube; **10:**Pandora; **11:**Facebook; **12:**Twitter; **13:**Gmail; **14:**Google Maps; **15:**ZArchiver; **16:**Accelerometer M.; **17:**Gyroscope Log; **18:**Proximity Sensor; **19:**Compass; **20:**Barometer; **21:**2048 Puzzle; **22:**Pet Rescue Saga; **23:**Pou; **24:**Solitaire; **25:**Words; **26:**CT 24; **27:**Live Extra; **28:**VEVO; **29:**VOYO.cz; **30:**WATCH ABC; **31:**Instagram; **32:**File Commander; **33:**RAR for Android; **34:**Dropbox; **35:**File Manager; **36:**Physics Toolbox; **37:**Sensor Kinetics; **38:**Android Sensor Box; **39:**Sensor Music Player; **40:**Sensor Mouse.

The reduction in cold launch delays with SmartIO ranges from 6.3% (Accelerometer Monitor) to 37.8% (The Simpsons) as compared to delays without SmartIO. The cold launch delay with SmartIO enabled for all the 40 apps is on average 20.5% faster than with SmartIO disabled. These results are expected. The app launch is I/O intensive, and includes a lot of read activities. The average number of reads observed for the 40 apps is 5 times higher than writes. Some apps even go to the extremes, for instance, the Temple Run game has reads exceeding writes by 58 times. Therefore, the read-preference nature of SmartIO contributes to reducing disk I/O delay during the launch. Specifically, the disk I/O delay portion itself is reduced on average by 69%. Slight difference in the user and system time of several apps suggests that SmartIO also affects other time components.

We reserve further investigation for future work.

The *warm launch delay* is a launch delay required to launch an application currently running in the background. The cache of such application is not cleared before the measurement. The warm launch delay of the 40 apps with and without SmartIO is illustrated in Figure 4.12(b). The absolute values of warm launch delays are on average 65% smaller than those of cold launch delays. This is reasonable, since once an app is already in memory, its launch is much faster. In addition, since there is little I/O traffic going to the flash disk (81% less than during cold launch), the reduction in delays for all 40 apps with SmartIO is on average only 6.8%. The disk I/O delay portion itself is reduced on average by 13%.

Run-time Delay. In order to test delays of apps running on the phone with SmartIO, we utilize again the Android Monkey tool to generate streams of 500 user events such as clicks, touches, or gestures. The *run-time delay* is defined as the time needed to complete the 500 user events in a running app. We run the experiments with the same 40 Android apps mentioned previously. Each app has a predefined set of user activities triggered through the Monkey tool. The run-time delay for both cases is measured with the *time* command, once with SmartIO enabled, and once with SmartIO disabled. Monkey is a command-line tool that can send a stream of events into the phone's system in a repeatable manner. We apply a constant seed value (10) to generate the same sequence of events. The events are individually adjusted for each app to represent a typical usage, for instance, in Gmail we read and write an email, add a contact, change a label, etc.

The run-time delay of the 40 apps with and without SmartIO is illustrated in Figure 4.12(c). The figure is plotted with standard deviations. The reduction in run-time delay with SmartIO ranges from 2% (Pandora) to 29.6% (Angry Birds) as compared to run-time delay without SmartIO. The run-time delay with SmartIO enabled for all the 40 apps is on average 16.9% smaller than with SmartIO disabled. Clearly, the run-time delays do not benefit from using SmartIO as much as the application launch. This is reasonable, since

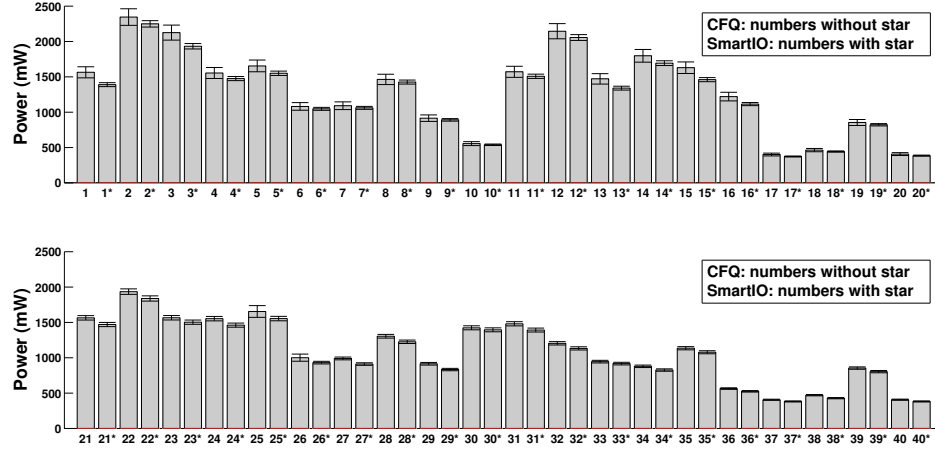


Figure 4.13: Power Consumption

the application launch is more I/O intensive than the application run-time. For the 40 apps, the average number of I/Os during launch is 2 times higher than during run-time. While the run-time delay of the games with SmartIO is on average 23% smaller, the streaming apps have on average only 4% smaller run-time delay. This is expected, since the games have decent disk I/O activity during the run-time, whereas the streaming apps are mainly network-bounded. For example, 56% of Angry Birds's run-time delay stems from disk I/Os, and the disk I/O delay portion itself is reduced by 49%. While 64.7% of CNN's run-time delay originates from network I/Os, and the disk I/O delay portion itself is only reduced by 8%. Finally, the average gains of the sensing and miscellaneous category are 18% and 20%, respectively. The improvement in the disk I/O portion of the time spent during run-time is on average by 54%.

Power Consumption.

While improving the application performance is important, having solid power efficiency is equally important. To measure power consumption, the Monsoon Power Monitor [17] is utilized. Each of the 40 apps is run with SmartIO disabled, and then enabled. The Android Monkey tool triggers the launch process of each app, and then generates the same stream of 500 user events as previously. The results with standard deviations are

presented in Figure 4.13. The average power consumption with SmartIO enabled is lower than the consumption with SmartIO disabled by 6%. Hence, our solution does not have energy overhead, and even contributes to lower power levels. We attribute this to the read-preference approach of the system that essentially allows shorter jobs to be completed first, which contributes to smaller application delay and consequently also lower power consumption.

4.5.5 User-Perceived Performance: Facebook

In this subsection we conduct an experiment on the Facebook application to determine the user-perceived performance improvement of our solution. Since the delays in Figure 4.12 are obtained in the OS layer, the values are precise but significantly smaller than if obtained in the application layer. In order to acquire measurements in the application layer, we may use a stop watch, which is however inaccurate. Instead, we choose to slightly modify the Facebook source code¹ to record timestamps of several performance parameters. Specifically, we focus on three metrics that are critical to Facebook users: cold launch, warm launch, and timeline loading. A short demo of a modified Facebook version is available at [15]. The app uses test accounts and automates 150 measurements per metric without necessity of any user interaction. The experiment is conducted on the five phones listed above.

Cold Launch. Cold launch in Facebook is defined as the time required to complete loading all components of the start activity and rendering of the News Feed. All cache data is cleared except the login information. The ultimate goal of Facebook Inc. for the following years is to have cold launch of less than 5 seconds on devices released in 2012 or newer, and less than 10 seconds on older devices. The results in Figure 4.14(a) show that cold launch on our oldest device RAZR (2012) takes 9.9 seconds with CFQ and 6.2 seconds with SmartIO. The newest phone Samsung S5 (2014) spends 3.7 seconds on cold

¹The author interned with Facebook Inc.

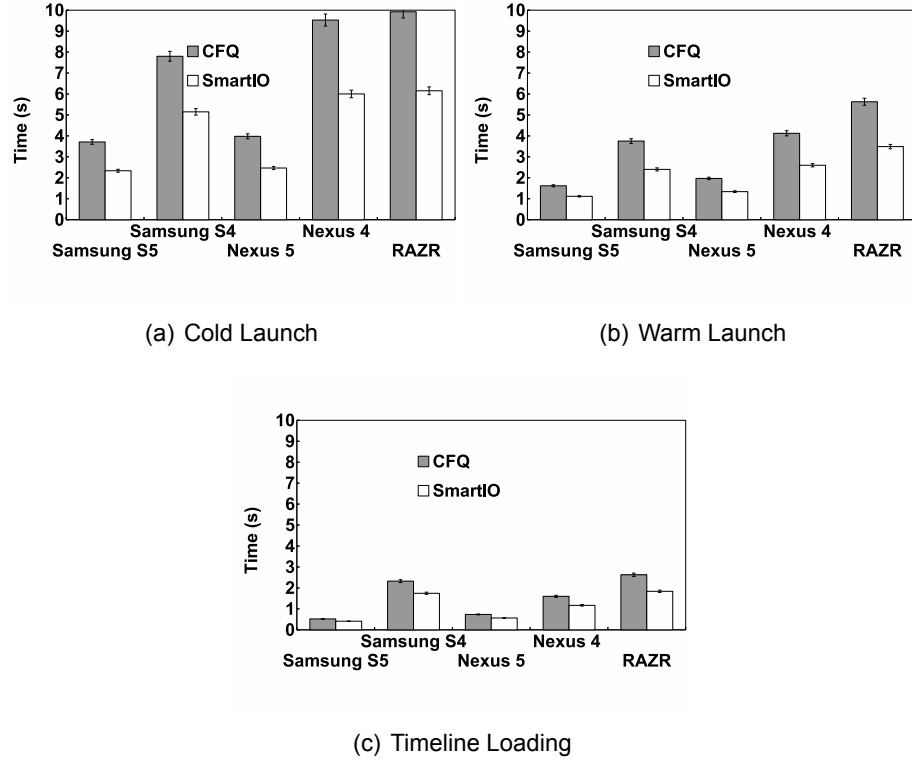


Figure 4.14: User-Perceived Performance of Facebook

launch with CFQ, and 2.3 seconds with SmartIO. Finally, cold launch with CFQ on Nexus 5 (2013), Nexus 4 (2012), and Samsung S4 (2013) requires 4 seconds, 9.5 seconds, and 7.8 seconds, respectively. While with SmartIO, the three devices need 2.5 seconds, 6 seconds, and 5.1 seconds, respectively. Since the shortest human perceivable delay is 100ms [44], we can conclude that SmartIO can contribute significantly to reducing the user-perceivable cold launch delay.

Warm Launch. Warm launch is defined similarly as cold launch, except the cache is not cleared before each measurement. Figure 4.14(b) indicates that RAZR has the most noticeable reduction in the delay. Specifically, warm launch with CFQ takes 5.6 seconds, while with SmartIO it takes 3.5 seconds. Nexus 4's warm launch delay is reduced from

4.1 seconds to 2.6 seconds. Samsung S4 shows a reduction from 3.8 seconds to 2.4 seconds. Finally, the newest devices Samsung S5 and Nexus 5 get their delays reduced from 1.6 second to 1.1 second, and from 2 seconds to 1.3 second, respectively.

Timeline Loading. Timeline is a user profile page. Its loading is defined as the time required to complete loading and rendering of all components in the profile activity, where the origin activity is the News Feed. This can be seen as switching from the News Feed to the Timeline page. The results in Figure 4.14(c) show less noticeable reductions in the delay. This is reasonable, since this timeline loading corresponds to run-time delays in Figure 4.12, where the I/O traffic is usually less intensive. RAZR and Samsung S4 show most significant delay reductions: from 2.6 seconds to 1.8 second, and from 2.3 seconds to 1.7 second, respectively.

4.6 Discussion and Future Work

Launch and run-time delays are critical to user experience, since one launches and runs apps repeatedly throughout the day. Therefore, we focus on launch and run-time delays. However, in future work we plan to evaluate the impact of other stages of the life cycle on application performance such as install, update, switch, and uninstall, and quantify their effects on everyday phone usage. We intend to extend this study by researching how other common usage patterns are impacted. For instance, taking photos, recording movies, messaging, calling, email sync (recently studied in [70]), etc.

As discussed earlier, one of the main reasons causing longer launch delay is the disk I/O performance, specifically read I/O performance. This is due to the read-intensive nature of application launch. The average number of reads observed during launch on the 40 popular apps in our experiment is five times higher than writes. Other factors may also play a role in the high variation of launch delays. In particular, the launch delay also depends on the app's physical location, i.e., whether on the internal flash or external SD

card. According to our analysis, the application size is not a big contributor to the launch delay. While the three largest apps Angry Birds (42.4MB), The Simpsons (41.7MB), and Temple Run 2 (36.7MB) have the launch delay around 0.65s, the smallest app Proximity Sensor (0.02MB) has the fifth largest launch delay (0.8s). Finally, we plan to analyze the impact of network I/O based on existing results [71, 39, 46, 47, 52, 66, 73].

Our work only focuses on reducing the application delay with respect to the internal flash storage. It may be also interesting to study how different applications use SD cards. Kim et al. [55] already performed a series of benchmarking experiments on SD cards from multiple speed classes. However, it will be useful to go beyond benchmarking and investigate I/O access patterns on these devices. This especially can benefit multimedia applications that store data on the external storage.

The major overhead of SmartIO is the additional delay in writes because it is designed to serve in favor of reads. As demonstrated in the evaluation, the write slowdown ratio worsens from 1.13 to 1.51 for sequential I/Os, while for random I/Os it worsens from 1.6 to 1.83. In another experiment, we install the 40 apps researched, and the results reveal that writes are on average 4.7% slower with SmartIO. However, at the same time, many other processes in the background may benefit from SmartIO. Based on our large-scale study, there are on average 255 processes running on each device at any point of time, from which 98 have some I/O activity and generate a workload. These processes are expected to have faster response time with SmartIO.

Our system keeps most of the dispatch process from the current Linux I/O scheduler unchanged. In particular, it only adds a third priority level to organize the dispatch queue in favor of reads. This third priority level preserves the original Linux scheduler design because it has a lower priority than the first two priority levels from the block layer. Therefore, the fairness between processes is still maintained, and a read from a process with lower priority may not incur unfair performance penalty on a service process with higher priority.

Finally, the observations made in our measurement study are based on data obtained

in the Samsung S5 phone and 2611 Android devices through StoreBench. I/O slowdown and concurrency measurements were excluded from StoreBench, since these tests take too long (around 1 hour) to complete, and would discourage users from using this benchmark tool.

Chapter 5

iRAM: Sensing Memory Needs of My Smartphone

5.1 Introduction

With smartphone penetration now reaching a total of nearly 1.3 billion units shipped in 2014 [35], the world of apps is also seeing a rise in popularity. According to a new report [33], mobile consumers download more apps than ever before, with the average number of apps owned by a smartphone user now at 41, a rise of 28 percent from the 32 apps owned on average last year. Apps are thought to make our life easier, doing things such as streamlining our calendars [29] and grocery lists [26], offering entertainment while we are stuck in line [25], and making it easy to collaborate with co-workers [27]. In a recent study [31], when consumers were asked if they had encountered a problem (app crashes, freezes, errors, or extremely slow launch) accessing a mobile app within the last six months, 56 percent said yes. Among those who have experienced a problem, 62 percent reported a crash, freeze or error; 47 percent experienced slow launch; and 40 percent reported an app that would not launch.

We believe that many such performance issues are due to little available memory and its inefficient utilization. Our study reveals that after having launched five regular

applications (Facebook, Instagram, G+, Angry Birds, and YouTube), the amount of free memory left is just around 4 percent. The low free memory level stays throughout the experiment, and does not recover to significantly larger values. This negatively affects the smartphone's overall application performance, and ultimately results in slow response times and crashes. In addition, many applications do not respect heap thresholds, and go far above their allowed heap usage, which also contributes to more erroneous behavior. Users quickly notice apps that are slow or likely to break (whether because of downtime, crashes, etc.), and this impairs both usage and brand perception. Users expect a mobile app to be fast and responsive; if it is not, it will get poor reviews, low ratings and low adoption numbers. While 79 percent of consumers would retry a mobile app only once or twice if it failed to work the first time, only 16 percent would give it more than two attempts [31]. Poor mobile app experience is likely to discourage users from using an app again.

This chapter addresses two key research questions towards achieving better application performance. (1) *How does memory affect smartphone application performance?* (2) *How can we improve application performance with memory optimization techniques?* In order to address the first research question, we study memory usage of several groups of applications. In particular, we identify the amount of memory available before and after their launch. Little available memory may result in delayed I/O operations or frequent communication with much slower flash disks, which essentially causes slow application response. Insufficient memory may even prevent an app from launching. Next, we investigate heap usage of applications. High heap usage of games and other rich multimedia apps may increase crash rates and likelihood of erroneous behaviors. To address the second research question, we design and implement a system prototype called iRAM on the Android platform. iRAM efficiently maximizes free memory levels, cleans low-priority processes, and maintains optimal heap size limits. The system learns which apps are of high priority for a particular user, and keeps them in the main memory. The launch of such apps is then much faster, since it corresponds to warm launch. iRAM also applies a prediction model to predict heap usage of a set of apps, and dynamically adjusts the

heap size based on predicted values. With this set of simple optimizations, iRAM reduces application delays and decreases likelihood of erroneous behaviors.

Little work in the research community directly relates to ours. Yan et al. [72] propose a system predicting application launch using context such as user location and temporal access patterns. Their system reduces perceived delay through application prelaunching. Our proposed solution efficiently maximizes free memory levels, cleans low-priority processes, and maintains optimal heap size limits. The system keeps high priority processes in the main memory. Hence, there is no complex prelaunching involved as in the previous work. Nguyen et al. [61] study the impact of flash storage on smartphone application performance. The authors design a system that improves the response time by prioritizing reads over writes. Our work does not study internal or external flash storage devices, but instead focuses on smartphone's memory component that is also referred to as dynamic random-access memory (DRAM).

In summary, the contributions of this chapter are as follows:

- First, through a measurement study we find that facilitating warm launch of just five applications is extremely expensive, using up to 36 percent of memory. The resulting little memory left can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Therefore, in order to improve the application performance, we investigate how each application consumes the memory. Our heap usage study of 20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries. This mainly applies to games that require up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Finally, further investigation reveals that limited heap may not only cause an app to crash, but may even prevent an app from launching. While on Samsung S4, all five games fail to launch until the heap size of 64MB, on Nexus 4, all five games fail to launch until the heap size of 128MB. Therefore, the heap

size directly affects success or failure of application launch.

- Second, we design and implement iRAM, a system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays.
- Third, our evaluation on memory hungry applications indicates that iRAM reduces application crashes by up to 14 percent. In addition, the results confirm that iRAM increases free memory levels by up to 4.8 times. The evaluation using 40 popular applications from four groups (games, streaming, miscellaneous, and sensing) also shows that iRAM reduces launch delays by up to 78.2 percent. This performance gain comes with 3.5 percent of CPU overhead and 0.9 percent of power overhead.

5.2 Measurement Study

In order to understand how memory affects smartphone application performance, we conduct a measurement study. First, we study memory usage of several regular applications. In particular, we measure the amount of memory available before and after their launch. Little available memory can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Next, we investigate heap usage of applications. High heap usage of games and other rich multimedia apps may increase crash rates and likelihood of erroneous behaviors. Then we look further into how the heap size affects success or failure of application launch. Finally, we discuss the measurement results and their implications. In our study, we utilize two devices: Samsung S4 (2GB RAM, 128MB heap size) with Android 4.3 and Nexus 4 (1GB RAM, 64MB heap size) with Android 4.2. The phones are normally used daily by the authors. During experiments, the devices have all radio communication disabled except for WiFi that is necessary to provide stable Internet connection required on most

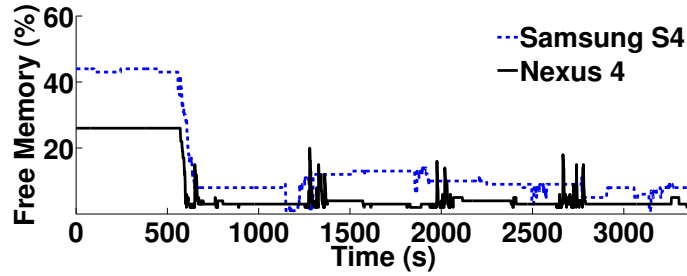


Figure 5.1: Free Memory

apps. The screen is set to stay-awake mode with constant brightness, and the screen auto-rotation is disabled. The cache is cleared before each measurement.

5.2.1 Free Memory

In this experiment, we study memory usage of five popular applications: Facebook, YouTube, CNN, Angry Birds, and Temple Run. In particular, we launch the five apps every ten minutes and issue 100 user events via Android Monkey [16], which corresponds to using an app for a few seconds. Then we close each app with the Home button keyevent, assuring that an app does not get killed and stays in the background. We record free memory levels within one hour. To facilitate the measurement, we implement a shell script with the *free* [30] command to output free memory levels of the devices.

The free memory levels during this experiment are illustrated in Figure 5.1. The results indicate that from the beginning both devices have only less than half of memory available, since the Android OS already consumes a large portion. When the apps are launched the first time, the memory level drops significantly. This is expected, since the first launch corresponds to cold launch. Specifically, Nexus 4 has afterwards only three percent of memory left, while Samsung S4 eight percent left. When the apps are relaunched after 10 minutes, the levels drop again but not dramatically, which is because the apps are already in the background (warm launch). However, facilitating this warm launch is extremely expensive, using up 36 percent of memory on Samsung S4 and 23 percent on Nexus 4. Notice also that Nexus 4 has mostly lower memory values throughout the measurement

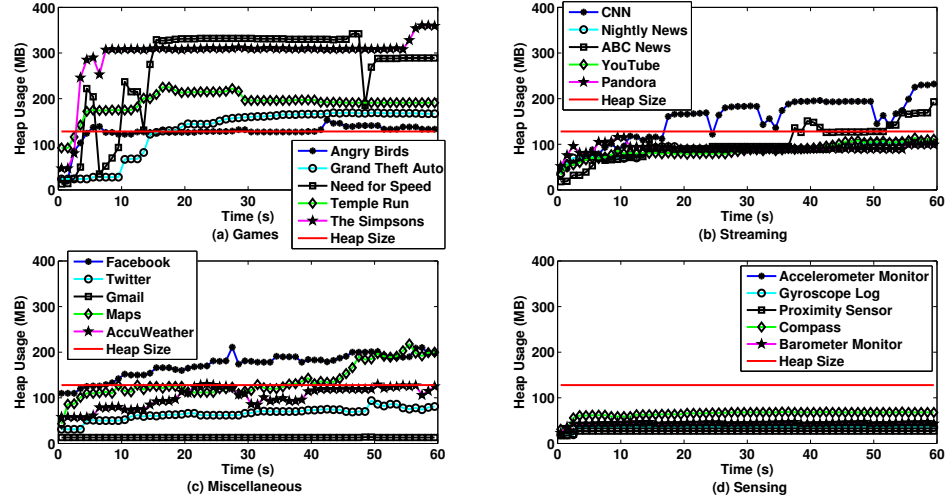


Figure 5.2: Heap Usage

due to its smaller total RAM capacity. This also causes Nexus 4 to reach a memory threshold triggering garbage collection, which is represented as spikes in the figure. The above results reveal that simple usage of only five regular applications can already use up a significant amount of memory. To understand how each application consumes the memory, we study heap usage of several groups of popular applications in the following subsection.

5.2.2 Heap Usage

This experiment investigates heap usage of applications. To obtain a deeper understanding, we look into the heap behavior of 20 applications from four diverse groups: games (Angry Birds, Grand Theft Auto, Need for Speed, Temple Run, The Simpsons), streaming (CNN, Nightly News, ABC News, YouTube, Pandora), miscellaneous (Facebook, Twitter, Gmail, Maps, AccuWeather), and sensing apps (Accelerometer Monitor, Gyroscope Log, Proximity Sensor, Compass, Barometer Monitor). Each application is launched and run for a minute and gets issued a set of predefined user events via Android Monkey. Every time an app is running, there are no other apps in the background. To facilitate the

measurement, we implement a shell script with the *dumpsys meminfo* [28] command to output the heap usage of the apps.

The results of the experiment are displayed in Figure 5.2. There are several key observations. Due to rich multimedia content, games in Figure 5.2 (a) use up much more heap than the heap size allowed by the OS. For instance, the Simpsons game reaches a maximum of 360MB, while the heap allowed is only 128MB on Samsung S4 (2.81 times difference), and 64MB on Nexus 4 (5.63 times difference), respectively. To make the figure less complex, we only display the larger heap size. The Android design allows apps to grow heap usage above the default heap size threshold, but such oversized usage does not have any performance guarantee, and apps may be killed unexpectedly by the system during run-time. This happens during the experiment with Need for Speed, and the app crashes unexpectedly at the fourth second, causing a sudden drop in heap usage. The app is subsequently relaunched at the sixth second to continue the experiment. The streaming apps in Figure 5.2 (b) overall use up less heap than games, but still go above the threshold, especially CNN and ABC News with their rich user interface. The miscellaneous apps in Figure 5.2 (c) indicate a varying heap usage. Facebook has high heap usage due to its aggressive caching policy and advanced multimedia support (five posts ahead, auto video play, images pre-loading, etc.). Maps go over the heap threshold when a route across the whole U.S. is loaded. While Gmail, mainly text oriented, is always far below the threshold. Finally, the sensing apps with their simple user interface use little heap, and are the least memory hungry. The main takeaways of this subsection are following. Rich multimedia applications may have high heap usage and go above allowed boundaries. This mainly applies to games that require up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Small amount of available memory may even prevent an application from launching. This is studied in the next subsection.

5.2.3 Application Launch

In the following experiment, we look further into how the heap size affects success or failure of application launch. We focus on games, since as seen previously they have the highest heap usage. The Android Monkey tool is utilized to trigger the launch process of each app ten times. We use `setprop dalvik.vm.heapsize` command to change the heap size from 16MB to 256MB, and record whether launch is successful or not.

Heap	App	Samsung S4	Nexus 4
16MB	AB	10x fail /10 trials	10x fail /10 trials
32MB	AB	10x fail /10 trials	10x fail /10 trials
64MB	AB	10x fail /10 trials	10x fail /10 trials
128MB	AB	10x success /10 trials	10x fail /10 trials
256MB	AB	10x success /10 trials	10x success /10 trials
16MB	GTA	10x fail /10 trials	10x fail /10 trials
32MB	GTA	10x fail /10 trials	10x fail /10 trials
64MB	GTA	10x fail /10 trials	10x fail /10 trials
128MB	GTA	10x success /10 trials	10x fail /10 trials
256MB	GTA	10x success /10 trials	10x success /10 trials
16MB	NFS	10x fail /10 trials	10x fail /10 trials
32MB	NFS	10x fail /10 trials	10x fail /10 trials
64MB	NFS	10x fail /10 trials	10x fail /10 trials
128MB	NFS	10x success /10 trials	10x fail /10 trials
256MB	NFS	10x success /10 trials	10x success /10 trials
16MB	TR	10x fail /10 trials	10x fail /10 trials
32MB	TR	10x fail /10 trials	10x fail /10 trials
64MB	TR	10x fail /10 trials	10x fail /10 trials
128MB	TR	10x success /10 trials	10x fail /10 trials
256MB	TR	10x success /10 trials	10x success /10 trials
16MB	TS	10x fail /10 trials	10x fail /10 trials
32MB	TS	10x fail /10 trials	10x fail /10 trials
64MB	TS	10x fail /10 trials	10x fail /10 trials
128MB	TS	10x success /10 trials	10x fail /10 trials
256MB	TS	10x success /10 trials	10x success /10 trials

Table 5.1: Application Launch

The results are displayed in Table 5.1. *AB: Angry Birds; GTA: Grand Theft Auto; NFS: Need for Speed; TR: Temple Run; TS: The Simpsons*. For Samsung S4, all five games fail to launch until the heap size of 64MB. This means that with limited heap, the system

implicitly does not allow application launch. Note that we launch apps via Android Monkey, which is a software approach. If a user launches an application explicitly with a touch on an icon, the launch in most cases will be triggered (and may succeed), but the application will likely crash or show an erratic behavior during run-time. Starting from 128MB heap, the application launch of all five games is successful. For Nexus 4, all five games fail to launch until the heap size of 128MB. The two devices behave differently, since they have different memory capacities and heap sizes. The above results reveal that limited heap may not only cause an app to crash, but may even prevent an app from launching. Current state-of-the-art [61, 72, 62] focuses on reducing application delays, while in this study we have observed even more critical issues that need to be addressed, and we summarize them in the subsection below.

5.2.4 Summary

The above experiments lead to several important observations that shed light on how to improve smartphone application performance, and we summarize them below. First, facilitating warm launch of just five applications is extremely expensive, using up to 36 percent of memory. The resulting little memory left can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Therefore, in order to improve the application performance, it is essential to understand how each application consumes the memory. Our heap usage study of 20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries. This mainly applies to games that require up to 5.63 times more heap than guaranteed by the system, and may cause crashes and erroneous behaviors. Finally, further investigation reveals that limited heap may not only cause an app to crash, but may even prevent an app from launching. While on Samsung S4, all five games fail to launch until the heap size of 64MB, on Nexus 4, all five games fail to launch until the heap size of 128MB. Therefore, the heap size directly affects success

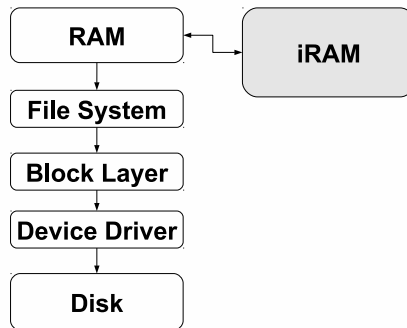


Figure 5.3: Kernel Components and iRAM

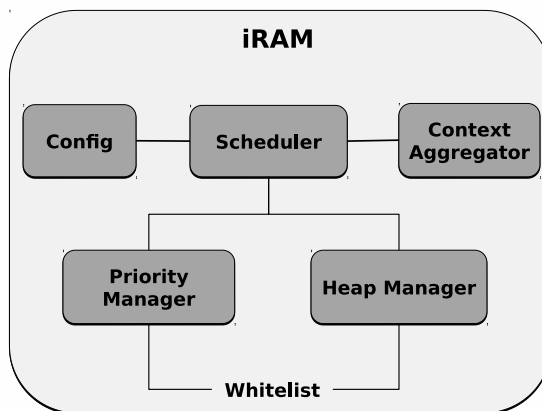


Figure 5.4: iRAM Architecture

or failure of application launch.

5.3 System Architecture Overview

In order to improve application performance in smartphones through memory optimizations, we need to address the above challenges. Of the top priority is the finding that rich multimedia applications have high heap usage and are likely to crash. Next, users cannot tolerate slow application response caused by little available memory. Therefore, we present iRAM, a system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. iRAM and main kernel components on the I/O path are displayed in Figure 5.3.

The architecture of iRAM is illustrated in Figure 5.4. It is fully located in the kernel space, and consists of several key modules: the Heap Manager, the Priority Manager, the Scheduler, the Config module, and the Context Aggregator. We elaborate each module and its functionality below. iRAM's complete source code is available for download on our GitHub page¹.

Heap Manager. Our system prototype follows the implications from the previous experimental study. Since high heap usage of games and other rich multimedia applications may cause crashes and erroneous behaviors, the Heap Manager dynamically configures global heap thresholds such that they are always higher than application heap requirements. To achieve this, we employ an autoregression model with exogenous inputs (ARX) to predict future heap usage, and then dynamically update the global heap size to avoid crashes. Specifically, we combine past data from the same run and historical data from previous runs together to predict future heap usage with the use of the ARX model. The prediction of future heap usage using past heap usage forms the autoregressive portion of the model, while the historical heap usage data serves as exogenous inputs. Finally, based on the predicted heap usage, we dynamically adjust the global heap size threshold to avoid crashes.

Priority Manager. The Priority Manager module efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. In particular, the Priority Manager finds candidate processes to be killed based on Android's importance hierarchy, from the lowest to the highest importance. Next, if there are any high priority user processes among the candidate processes, they are filtered out. Finally, the processes left in the candidate list are killed. Motivated by [62], we obtain high priority user processes through a simple prediction method. Assuming the next application to be used by the user has the highest user priority, we apply prediction by partial matching (PPM). Therefore, predicting next app based on previous usage pattern corresponds to predicting next letter based on probability distribution occurrence of previous letters. iRAM predicts

¹<https://github.com/dunguk/iRAM>

9 next applications and places them in the Whitelist. If a candidate process to be killed is also listed in the Whitelist, such process is filtered out and is not killed.

Scheduler. The Scheduler triggers Priority Manager and memory cleaning if the free memory level is below a threshold defined in the Config module. The Scheduler checks this memory level each time period, which is also configurable in the Config module. By default, we set the period to be 20 seconds when the device screen is on, and 60 seconds when the screen is off. The rationale behind this is the assumption that when it is off, there are not many user activities, and hence no frequent cleaning is required. While when the screen is on, the user is actively using the device, and likely many memory operations are going on.

Config Module. Config includes several global parameters. *AGGRESSION_LEVEL* defines how aggressively the system should proceed during memory cleaning. There are three levels, roughly 1 includes all background processes, 2 includes background processes and system caches, 3 includes background processes and foreground processes and system caches. Details are discussed in the Priority Manager Design section. Finally, *MIN_MEM* is a memory threshold below which the system should proceed with cleaning, and its default value is set to be 60 percent, implying that a relatively high free memory level is required.

Context Aggregator. Context Aggregator collects information about the user and device. Such information is, for example, whether the device is being used or in the sleep mode, how long the device is being used or how long the screen is off. The context information is utilized by the Scheduler module to manage memory cleaning process.

5.4 Heap Manager Design

In this section, we elaborate the Heap Manager's system design previously introduced in System Architecture Overview (Section 5.3). As observed before, games and other rich multimedia applications may crash during launch due to their high heap usage. To avoid

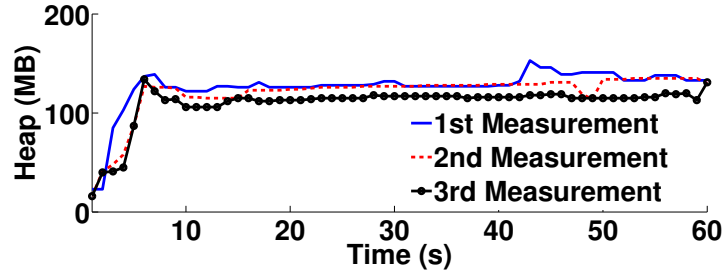


Figure 5.5: Angry Birds' Heap Usage

such erroneous behaviors, the Heap Manager dynamically configures global heap thresholds such that they are always higher than application heap requirements. To achieve this, we apply an autoregression model with exogenous inputs (ARX) to predict future heap usage, and then dynamically update the global heap size to avoid crashes.

5.4.1 Heap Usage Prediction with ARX

To predict future heap usage, a simple approach is applying past data, which refers to the heap usage data obtained from the moment an app is launched till the current time stamp. The accuracy of such method is based on high autocorrelation between the predicted heap usage and past heap usage. If predicted and past heap usage data are highly correlated, then this method can achieve satisfiable prediction result. Figure 5.5 displays three measurements of Angry Birds' application launch. The figure indicates that after 10 seconds, heap usage remains relatively stable. During this stable heap period, if we use past stable heap usage data to predict future usage, the prediction accuracy will be high. However, heap usage increases rapidly at the beginning of the application launch. Therefore, if we only predict heap usage based on past data obtained during launch, the predicted value will be lower than the real heap usage. Hence, setting the global heap size threshold based on the predicted value may lead to an application crash, since the real heap usage will exceed the threshold. Another method to predict heap usage is based on historical data, which refers to heap usage traces collected in the past. Figure 5.5 indicates that the launch curves from different measurements are similar at the beginning

(approx. first 10 seconds). The correlation coefficients between the first and second, second and third, third and first measurements in the first 10 seconds are 0.907, 0.892, and 0.987, respectively. However, once the app runs stably, heap usage from different measurements shows about 20MB difference. Therefore, after the app runs stably, prediction only based on historical data will not be accurate.

Based on the above observations, we combine past data from the same run and historical data from previous runs together to predict future heap usage with the use of the ARX model. The prediction of future heap usage using past heap usage forms the autoregressive portion of the model, while the historical heap usage data serves as exogenous inputs:

$$y(t) = \sum_{i=1}^p a_i y(t-i) + \sum_{j=0}^{q-1} b_j u(t-j) + e(t), \quad (5.1)$$

where t indexes time, $y(t)$ denotes the heap usage at time t , $u(t)$ represents historical heap usage at time t , a_i and b_j are coefficients, and $e(t)$ is a sequence of independent random variables. The objective of the model is to provide timely prediction of future heap usage. In order to do this, we have to estimate the coefficients, a_i , b_j . In addition, the model orders p and q are also unknown and have to be estimated. Parameters a_i and a_j can be estimated using the least-squares method. For orders p and q , in our experiment, we vary p from 0 to 3 and q from 0 to 4 in Equation 5.1 to obtain the optimal values of p and q for prediction. Intuitively, this answers the question of how much of past and historical heap usage data should be used to predict heap usage in the future. Within the ranges examined, $p = 0$ or $q = 0$ represent models where there is no past data or historical data. Also, if $p = 0$ and $q = 1$, we have a linear regression between current heap usage and historical heap usage. If $q = 0$, we have standard autoregression model (AR).

5.4.2 ARX Parameters

Based on the ARX model in Equation 5.1, we adopt K-fold cross-validation approach to compute the optimal combination of p and q . In a typical K-fold cross-validation scheme,

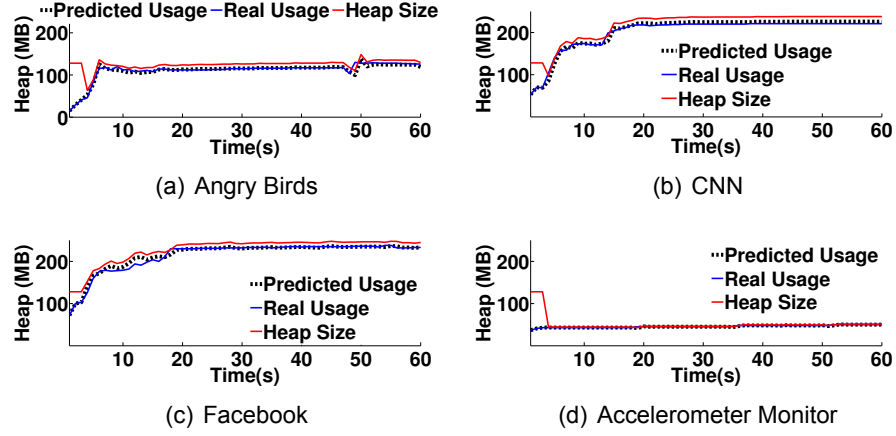


Figure 5.6: Heap Usage Prediction

one dataset is equally divided into K subsets. At each step of the scheme, one subset is selected as a test set, while other subsets function as a training set in order to estimate the model parameters. In the experiment, we collect eleven one-minute-long heap usage traces for each app as the dataset in K -fold cross-validation scheme. Heap usage prediction in the ARX model depends both on past heap usage and historical heap usage. Therefore, we randomly select one trace as a historical trace, and combine it with any other trace into one sample. This way the dataset is divided into K subsets, where each subset contains two traces, one current running and one historical trace.

By studying the launch curves of all applications, we find a common property that it takes three seconds for apps to use up the default heap size (128MB). This means the default heap size helps avoiding crashes during the first three seconds, but does not help avoiding crashes afterwards. Therefore, we can apply the default heap size for the first three seconds, and use the ARX model to predict heap usage for the time period after those three seconds. Notice that $y(t)$ is computed based on past data, $y(t-1), \dots, y(t-p)$, and historical data, $u(t), \dots, u(t-q+1)$. Therefore, the first data value that can be predicted is $y(\max(p+1, q))$. As we use the ARX model after three seconds, the first prediction time $\max(p+1, q)$ should be less than or equal to four

seconds. Therefore, the data range for p is 0, 1, 2, 3, and q is 0, 1, 2, 3, 4.

Considering the above constraints, our K-fold validation testing procedure is as follows. For each app and for each (p, q) pair from $p = 0, 1, 2, 3$ and $q = 0, 1, 2, 3, 4$, repeat the following steps:

1. Divide the dataset into K subsets $\{S_1, S_2, \dots, S_K\}$.
2. For each $S_k, k = 1, \dots, K$, compute the parameters a_i and b_j using all the other subsets with the least squares methods. Based on the estimated model parameters and associated prediction model in Equation 5.1, predict the heap usage value of each member of S_k .
3. Compare the predicted heap usage result with the real heap usage data using the root mean-squared error (RMSE):

$$\varepsilon = \sqrt{\frac{1}{T} \sum_t \left(y(t)_{predicted} - y(t)_{real} \right)^2} \quad (5.2)$$

We choose $K = 10$. Using the above 10-fold cross-validation, we compute the root mean-squared error for all the apps under different (p, q) pairs. The (p, q) pair that gives the smallest RMSE is selected as the optimal model orders. Table 5.2 shows the RMSE for Angry Birds. The table indicates that the ARX model achieves the smallest RMSE under $p = 2$ and $q = 1$. This means that to predict heap usage for Angry Birds at time t , two heap usage data values at time $(t - 1)$ and $(t - 2)$, and one historical heap usage data value at time t are the most effective data points in prediction. We also observe that when using past heap usage data alone ($q = 0$) or using historical heap usage data alone ($p = 0$), the RMSE values are relatively high (larger than 7.6MB and 8.4MB in corresponding cases). When past and historical data are combined together ($p \neq 0, q \neq 0$), we get lower RMSE values. This indicates that both past heap usage data and historical heap usage data are useful in prediction.

5.4.3 Global Heap Threshold

Based on the predicted heap usage, we update the global heap size threshold as follows:

$$y(t)_{global} = y(t)_{predicted} + \alpha \cdot \varepsilon, \quad (5.3)$$

	$q=0$	$q=1$	$q=2$	$q=3$	$q=4$
$p=0$	null	11.07	9.49	8.50	8.45
$p=1$	7.68	5.97	6.16	6.29	5.98
$p=2$	7.63	5.95	6.15	6.38	6.07
$p=3$	7.63	6.17	6.26	6.33	6.14

Table 5.2: Angry Birds' RMSE

where $y(t)_{global}$ is the global heap size we need to set, $y(t)_{predicted}$ is the predicted heap usage, α is a constant, and ε is root mean-squared error. We vary α from 0 to 5 to compute the global heap threshold, and compute the percentage of time that the threshold is larger than real heap usage under different α . When α is set to 0, 1, 2, 3, 4, we get the following numbers: 45.47%, 81.14%, 96.05%, 98.39%. The larger the α is, the larger the heap threshold, and the lower probability the app crashes. However, if the heap threshold is set to be a very large value, the system may need to distribute a large amount of memory to an app. In our system, we select α as 2. In this case, the real heap usage does not exceed the global heap threshold for 96% of time. To dynamically adjust the heap size, we utilize Android command `setprop dalvik.vm.heapsize`.

We plot heap usage prediction for Angry Birds in Figure 5.6(a) ($p = 2$ and $q = 1$). The figure confirms that the Predicted Usage curve and Real Usage curve are in proximity with the maximum error of 29.20MB and RMSE of 5.33MB. The global heap threshold (Heap Size) is set equal to the default threshold provided by the device manufacturer (128MB) for the first three seconds. Then the threshold is updated by Equation 5.3. The figure indicates that real usage data only exceeds the heap size at the time stamp of the 49th second by 18.55MB, which demonstrates the effectiveness of the Heap Manager module. We also display the results for a sample streaming app (CNN) in Figure 5.6(b), a miscellaneous app (Facebook) in Figure 5.6(c), and a sensing app (Accelerometer Monitor) in Figure 5.6(d), with RMSE values of 5.47MB, 5.34MB, and 0.48MB, respectively. This confirms that our heap prediction methodology allows efficient heap utilization by setting the heap size reasonably close to its real usage.

5.5 Priority Manager Design

In this section, we elaborate the Priority Manager's system design previously introduced in Section 5.3. Our Measurement Study (Section 5.2) leads to several important observations that shed light on how to reduce smartphone application delay. First, facilitating warm launch of just five applications is extremely memory expensive. Little free memory can be one of the main reasons causing slow application response due to delayed I/O operations or frequent communication with much slower flash disks. Moreover, insufficient memory may even prevent an app from launching. Therefore, we employ the Priority Manager module that efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. In particular, the Priority Manager finds candidate processes to be killed based on Android's importance hierarchy, from the lowest to the highest importance. Next, if there are any high priority user processes among the candidate processes, they are filtered out. Finally, the processes left in the candidate list are killed.

Processes with the lowest importance are obtained based on Android's importance hierarchy. Since its documentation [34] is incomplete, we discuss below to provide a full picture. There are eight levels, and the following list presents the different types of processes in order of importance (from the highest to lowest).

1. System: the main system process, of the top priority
2. Persistent: lower-level priority system processes supporting drivers and manufacturers' services (e.g., *android.nfc*, *qualcomm.wfd.service*)
3. Foreground: processes hosting an activity or a service that the user is interacting with (e.g., *com.facebook.katana*)
4. Visible: processes hosting an activity or a service bound to a foreground activity (e.g., *google.process.location*, *android.bluetooth*)
5. Perceptible: processes with active notifications (e.g., *android.input-method.latin*)

6. Service: processes started with the `startService()` method and do not fall into either of the two higher categories (e.g., *android.process.media*, *android.gms.wearable*)
7. Background: processes holding an activity that is not currently visible to the user; most apps running previously by the user become background processes
8. Empty: processes not holding any active application components

Based on the *AGGRESSION.LEVEL*, iRAM selects processes to kill from the lowest to highest importance. If it equals 1, iRAM populates the candidate list with Empty and Background processes. If it equals 2, iRAM populates the candidate list with Empty processes, Background processes, and system caches. If it equals 3, iRAM populates the candidate list with Empty, Background, Service, Perceptible, Visible, Foreground processes, and system caches.

Next, if there are any high priority user processes among the candidate processes, they are filtered out. We define high priority user processes as applications that have the highest probability of being used next by the user. Motivated by [62], we obtain high priority user processes through app prediction by partial matching, a variant of an existing text compression method called PPM [49]. PPM generates a probability distribution for the prediction of the next character in a sequence. Consider the alphabet of lower case English characters and the input sequence ``abracadabra". Assume that each character corresponds to an application used by the user. For each character in this string, PPM needs to create a probability distribution representing how likely the character is to occur. However, the only information it has to work with is the record of previous characters in the sequence. For the first character in the sequence, there is no prior information about what character is likely to occur, so assigning a uniform distribution is the optimal strategy. For the second character in the sequence, 'a' can be assigned a slightly higher probability because it has been observed once in the input history.

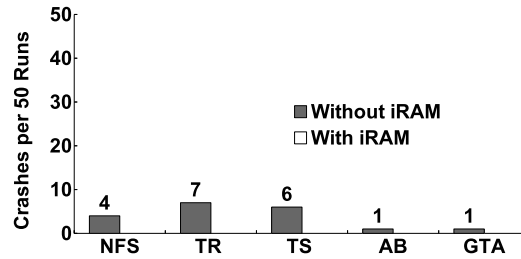
Consider the task of predicting the next character after the sequence ``abracadabra". One way to go about this prediction is to find the longest match in the input history which

matches the most recent input. The most recent input is the character furthest to the right and the oldest input is the character furthest to the left. In this case, the longest match is ``abra" which occurs in the first and eighth positions. The string ``dabra" is a longer context from the most recent input, but it does not match any other position in the input history. Based on the longest match, a good prediction for the next character in the sequence is simply the character immediately after the match in the input history. In this case, after the string ``abra" was the character 'c' in the fifth position. Hence, 'c' is a good prediction for the next character.

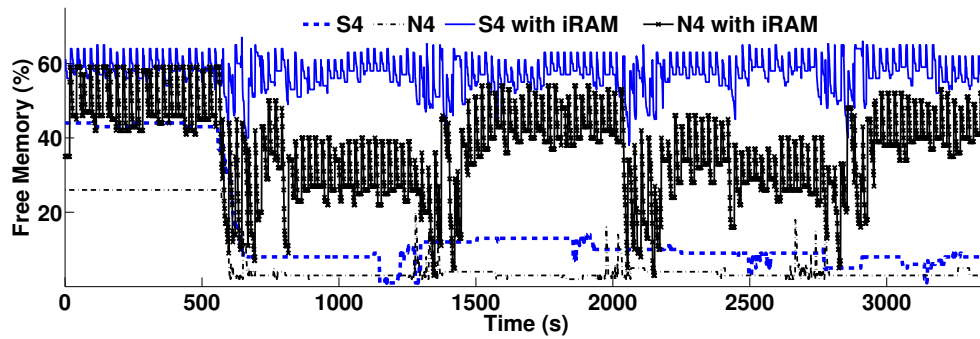
Therefore, predicting next app based on previous usage pattern corresponds to predicting next letter based on probability distribution occurrence of previous letters. Since we want to predict at each moment nine next applications to be used by the user, we want to know nine next characters instead of one next character. The choice of nine apps will be explained in the evaluation. Our modified code to [62] is also available on our previously mentioned GitHub account. As proven by the previous work, this prediction approach outperforms all previous solutions that also incorporate user context such as location and time.

5.6 Performance Evaluation

This section evaluates iRAM, and answers the following questions. *(1) How does iRAM contribute to reducing erroneous application behaviors?* We address this by performing a crash rate test on memory hungry applications. *(2) How does iRAM improve application performance?* This is addressed by evaluating how free memory levels are improved by iRAM, and how well high priority applications are predicted. We also record the launch delay of 40 popular apps from Google Play with and without iRAM. In addition, we conduct an experiment on the Facebook application to determine the user-perceived performance improvement of our solution. *(3) Does iRAM incur any performance penalties or cost?* This is determined by evaluating CPU and power overhead.



(a) Crash Rate (NFS: Need for Speed; TR: Temple Run; TS: The Simpsons; AB: Angry Birds; GTA: Grand Theft Auto)



(b) Free Memory

Figure 5.7: Crash Rate and Free Memory

The evaluation utilizes the same devices as in the measurement study: Samsung S4 (2GB RAM) with Android 4.3 and Nexus 4 (1GB RAM) with Android 4.2. When necessary, we discuss results of both devices, otherwise Samsung S4 is the main phone. As mentioned, the phones are normally used daily by the authors. During experiments, the devices have all radio communication disabled except for WiFi that is necessary to provide stable Internet connection required on most apps. The screen is set to stay-awake mode with constant brightness, and the screen auto-rotation is disabled. The cache is cleared before each measurement.

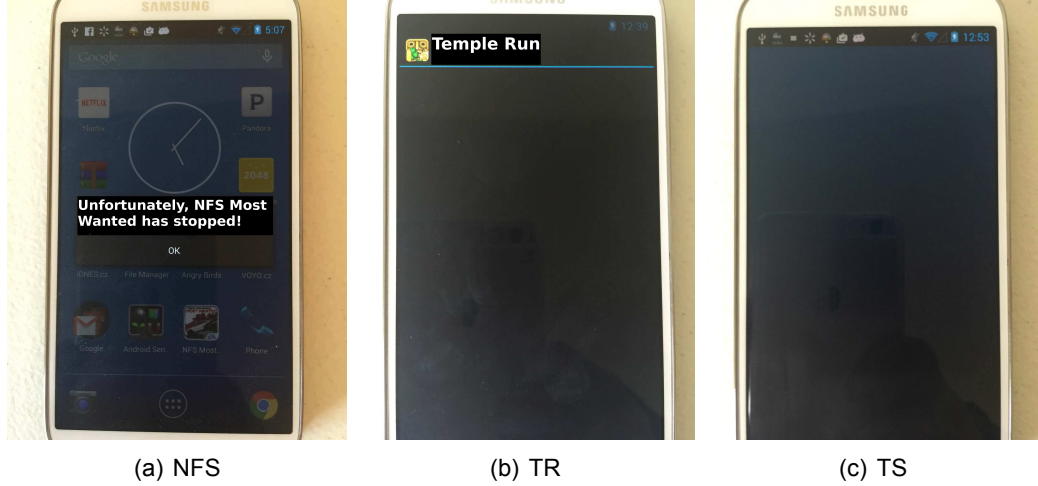


Figure 5.8: Crash Behaviors

5.6.1 Crash Rate

Our Measurement Study of 20 popular applications indicates that rich multimedia applications have high heap usage and go above allowed boundaries, which may cause crashes and erroneous behaviors. Therefore, in this section we investigate how iRAM contributes to reducing erroneous behaviors of the five games (Need for Speed, Temple Run, The Simpsons, Angry Birds, and Grand Theft Auto). Specifically, we launch each game every ten minutes, and record crash statistics, with and without iRAM. Each time we launch a game, there are four common apps in the background (Facebook, YouTube, Accelerometer Monitor, and Heads Up), while the game being launched is not currently in the background. This is to create typical memory pressure on the device. Having four apps in the background is reasonable, since current Android OS allows users to have up to 17 background apps. Each game is launched 50 times throughout 500 minutes.

The experiment has several interesting findings. First, the crashes of the applications have varying behaviors, as illustrated in Figure 5.8. (a) *Need for Speed* app displaying error message “Unfortunately, NFS Most Wanted has stopped!”; (b) *Temple Run* app displaying its logo and a black screen; (c) *The Simpsons* app displaying a black screen.

Need for Speed displays an error message indicating the app has stopped. While Temple Run crashes by showing its logo and a black screen, The Simpsons game displays a completely black screen. Finally, Angry Birds and Grand Theft Auto freeze on their launch screen. For the sake of space, we display the first three cases. All these crash behaviors are unacceptable, and impair both usage and brand perception. Users expect a mobile app to be fast and responsive, but most of all, users expect an app to work.

Next, Figure 5.7 (a) illustrates the amount of crashes over 50 runs for each application. Without iRAM, Temple Run has seven crashes over 50 runs, corresponding to 14 percent crash rate. Compared to other games, Temple Run has the highest heap usage during the first two seconds of its launch, which explains this result. The Simpsons game has six crashes, and Need for Speed four crashes. These two games have the steepest growth of heap usage, and hence the high crash rates are not surprising. Finally, Angry Birds and Grand Theft Auto have the lowest crash rate. Both games have mild heap usage at the beginning, and their maximal values are much smaller than those of others. Note that with iRAM, there are no crashes recorded. This is credited to the fact that iRAM predicts future heap usage, and then dynamically updates the global heap size thresholds such that they are always higher than application heap requirements.

Is the crash rate of up to 14 percent of the state-of-the-art significant? We think it is. Bad application performance means losing users. Users will not tolerate a problematic mobile app, and will abandon it after only one or two failed attempts. According to a recent study [31], 79 percent of users would retry a mobile app only once or twice if it failed to work the first time. In addition, dissatisfied users are driven to competitive apps and will spread unfavorable reviews in person and online. If dissatisfied with the performance of a mobile app, 48 percent of users would be less likely to use the app again.

So what are the main reasons causing app crashes? There are no statistics covering applications across the board, but based on error submissions from Facebook's user base, 62 percent of app crashes are due to OutOfMemory errors, while others are due to various system instability (CPU, GPU, network, storage, etc.). The results are based on

1.385 billion Facebook's mobile monthly active users across all mobile platforms (Android, iOS, Windows Mobile, etc.). According to our analysis, most OutOfMemory errors are caused by bad programming habits of application developers. iRAM can minimize consequences of poor memory management, but only developers can eliminate root causes of app crashes. To avoid memory leaks, developers may follow this simple guidance:

1. Strictly apply pairs based on owner lifecycle

- onResume -> onPause
- onCreate -> onDestroy
- onAttachToWindow -> onDetachFromWindow

2. Use only what is needed

- plan standard memory usage, and specify in the app's manifest if higher usage is expected
- monitor usage (use tools such as dumphsys, littleeye, Omura, etc.)

5.6.2 Free Memory

In this experiment, we study memory usage of the five popular applications used in the Measurement Study: Facebook, YouTube, CNN, Angry Birds, and Temple Run. In particular, we launch the five apps every ten minutes and issue 100 user events via Android Monkey [16], which corresponds to using an app for a few seconds. Then we close each app with the Home button keyevent, assuring that an app does not get killed and stays in the background. We record free memory levels within one hour. This is measured both on Samsung S4 and Nexus 4, with and without iRAM.

The free memory levels during this experiment are illustrated in Figure 5.7 (b). The results indicate that without iRAM, Samsung S4 has an average of 15.1 percent of free memory, and an average of 57.1 percent with iRAM (3.8 times more). Nexus 4 has an average of 7.5 percent of free memory without iRAM, and 35.9 percent of free memory

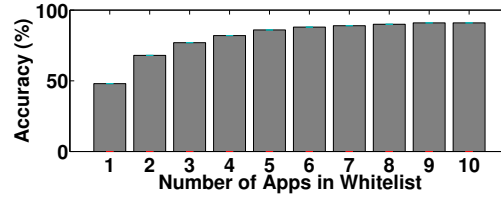


Figure 5.9: Prediction Accuracy

with iRAM (4.8 times more). This large amount of memory on both devices is made available thanks to the Priority Manager that periodically finds candidate processes to be killed based on Android's importance hierarchy, from the lowest to the highest importance. However, if there are any high priority user processes among the candidate processes, they are filtered out and hence not killed. High priority user processes are obtained through a prediction method, and we will find out in the following subsection how well such prediction works.

5.6.3 High Priority Processes Selection

Motivated by [62], we obtain high priority user processes through a simple prediction method elaborated in Section 5.5. Assuming the next application to be used by the user has the highest user priority, we apply prediction by partial matching (PPM). Therefore, predicting next app based on previous usage pattern corresponds to predicting next letter based on probability distribution occurrence of previous letters. iRAM predicts 9 next applications and places them in the Whitelist. If a candidate process to be killed is also listed in the Whitelist, such process is filtered out and is not killed.

We run the prediction on a dataset from real-world iPhone usage of 34 students [68] in the period of one year. The trace with applications run by users includes entries with user IDs, names of applications, and time stamps. The prediction accuracy results are displayed in Figure 5.9. As seen, the more applications in the Whitelist, the higher likelihood it includes an app to be used next. In other words, the more apps in the Whitelist, the better the prediction. However, the accuracy growth slows down significantly when

close to 5-6 apps, and there are almost no changes when close to 9-10 apps. Since we aim for high accuracy, but also not having too many apps to potentially pollute memory, nine applications seem reasonable. That is also what iRAM applies. Therefore, an app to be used next by the user exists in the Whitelist with the probability of 91 percent.

Since iRAM has 91 percent of prediction accuracy, in order to evaluate launch delay of an application, we can assume the app is in the Whitelist. This is discussed in the following subsection.

5.6.4 Launch Delay

To address the second question on how iRAM improves application performance, we measure launch delay of 40 popular apps (10 games, 10 streaming, 10 miscellaneous, and 10 sensing) from Google Play, with and without iRAM. In order to evaluate application launch with iRAM, we insert each tested app in the Whitelist. During the experiment, only one app runs at a time. This is to achieve a fair comparison between the two cases: with iRAM, and without iRAM.

The Android Monkey tool [16] is utilized to trigger the launch process of each app. The application *launch delay* starts when the launch process is triggered, and ends when the process completes. The launch delay includes three components. We use the *time* command [22] to output the three time components: the time taken by the app in the user mode (*user*), the time taken by the app in the kernel mode (*system*), and the time the app spends waiting for the disk and network I/Os to complete (*totalIO*). The storage I/O delay is obtained by dividing the total number of I/Os completed ($kBread + kBwrtn$) over the total rate of I/Os completed ($kBreadRate + kBwrtnRate$) in a flash block device. The network I/O delay is then calculated as the total I/O delay (*totalIO*) subtracted by the storage I/O delay (*diskIOdelay*).

Formally,

$$diskIOdelay = \frac{kBread + kBwrtn}{kBreadRate + kBwrtnRate}, \quad (5.4)$$

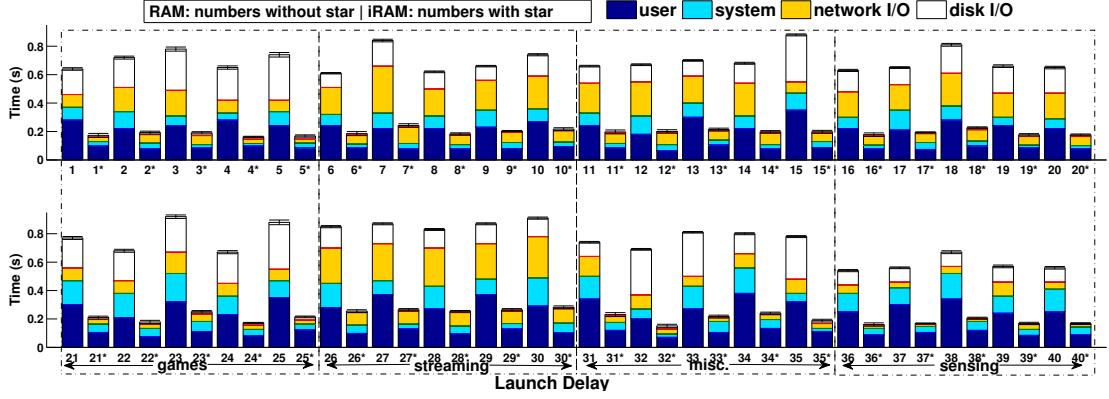


Figure 5.10: Launch Delay

where $kBread$ is the amount of data read from a flash block device, $kBwrtn$ is the amount of data written to a flash block device, $kBreadRate$ is the data rate read per second from a flash block device, and $kBwrtnRate$ is the data rate written per second to a flash block device. All four variables are obtained from the output of the *iostat* Linux command.

$$networkIOdelay = totalIO - diskIOdelay, \quad (5.5)$$

where $totalIO$ is the time an app spends waiting for both disk and network I/Os to complete. The variable is obtained from the *time* command during application launch.

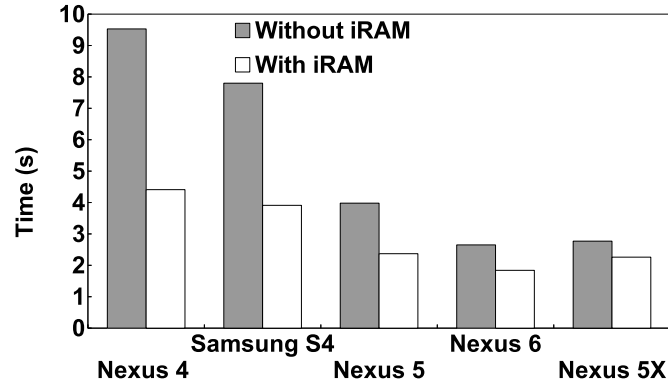
The launch delay of the 40 apps with and without iRAM is illustrated in Figure 5.10. The figure includes 10 games (1-5, 21-25), 10 streaming apps (6-10, 26-30), 10 miscellaneous apps (11-15, 31-35), and 10 sensing apps (16-20, 36-40). Applications running with iRAM are denoted with a star (*). **1:Angry Birds; 2:GTA; 3:Need for Speed; 4:Temple Run; 5:The Simpsons; 6:CNN; 7:Nightly News; 8:ABC News; 9:YouTube; 10:Pandora; 11:Facebook; 12:Twitter; 13:Gmail; 14:Google Maps; 15:ZArchiver; 16:Accelerometer M.; 17:Gyroscope Log; 18:Proximity Sensor; 19:Compass; 20:Barometer; 21:2048 Puzzle; 22:Pet Rescue Saga; 23:Pou; 24:Solitaire; 25:Words; 26:CT 24; 27:Live Extra;**

28:VEVO; 29:VOYO.cz; 30:WATCH ABC; 31:Instagram; 32:File Commander; 33:RAR for Android; 34:Dropbox; 35:File Manager; 36:Physics Toolbox; 37:Sensor Kinetics; 38:Android Sensor Box; 39:Sensor Music Player; 40:Sensor Mouse. The reduction in launch delays with iRAM ranges from 68.8 percent (Instagram) to 78.2 percent (File Commander) as compared to delays without iRAM. The launch delay with iRAM enabled for all the 40 apps is on average 71.9 percent faster than with iRAM disabled. These results are expected. The app launch is I/O intensive, and includes a lot of I/O activities involving the flash disk. However, thanks to iRAM, an application being launched is already in the background, and most I/Os only involve the main memory, which is much faster than the flash disk. The speed of main memory on our device is 400Mbps, while the speed of the flash disk is only 24Mbps, which makes the main memory 16.7 times faster than the flash disk.

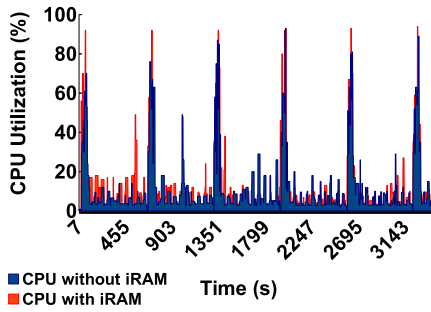
5.6.5 User-Perceived Performance: Facebook

In this subsection we conduct an experiment on the Facebook application to determine the user-perceived performance improvement of our solution. Since the delays in Figure 5.10 are obtained in the OS layer, the values are precise but significantly smaller than if obtained in the application layer. In order to acquire measurements in the application layer, we may use a stop watch, which is however inaccurate. Instead, we choose to slightly modify the Facebook source code to record timestamps of the launch delay. The app uses test accounts and automates 150 measurements without necessity of any user interaction.

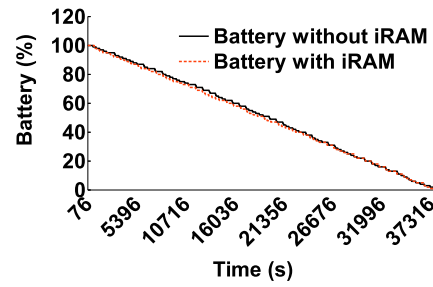
Facebook launch is defined as the time required to complete loading all components of the start activity and rendering of the News Feed. The ultimate goal of Facebook Inc. for the following years is to have the launch delay of less than 5 seconds on devices released in 2012 or newer, and less than 10 seconds on older devices. The results in Figure 5.11(a) show that launch on our older device Nexus 4 (2012) takes 9.5 seconds



(a) Facebook Launch Delay.



(b) CPU Overhead.



(c) Power Overhead.

Figure 5.11: Launch Delay and Overhead

without iRAM and 4.1 seconds with iRAM. The newer phone Samsung S4 (2013) spends 7.8 seconds on launch delay without iRAM, and 3.75 seconds with iRAM. To evaluate performance on more recent devices, we also conduct the experiments on a Nexus 5 phone running Android 4.4 with 2GB RAM (2013), Nexus 6 running Android 5.1.1 with 3GB RAM (2014), and Nexus 5X running Android 6.0 with 2GB RAM (2015). The results from the Nexus 5 and Nexus 6 are as great as expected. Specifically, the launch delay has been reduced from 3.98 seconds to 2.37 seconds, and from 2.65 seconds to 1.84 seconds, respectively. Surprisingly, the newest device (Nexus 5X) has been outperformed by its older brother (Nexus 6), and only reduces the delay from 2.77 seconds to 2.26 seconds.

We think that this is mainly due to its smaller memory capacity and the brand new Android release 6.0 that reportedly is still not without glitches. Overall, since the shortest human perceivable delay is 100ms [44], we can conclude that iRAM can contribute significantly to reducing the user-perceivable launch delay.

5.6.6 Overhead

To answer the last question whether iRAM incurs any performance penalties, we evaluate iRAM's CPU and power overhead.

CPU Overhead. In this experiment, we study the CPU overhead of the five popular applications used in the Measurement Study (Section 5.2): Facebook, YouTube, CNN, Angry Birds, and Temple Run. In particular, we launch the five apps every ten minutes and issue 100 user events via Android Monkey [16], which corresponds to using an app for a few seconds. We record CPU utilization within one hour, with and without iRAM, and the results are illustrated in Figure 5.11(b). As observed, CPU utilization with iRAM is on average 3.5 percent higher than the case without iRAM. iRAM's CPU utilization peaks during the launch time of the five apps, but since each optimization period lasts only 0.13 second, the average overhead is acceptable.

Power Overhead. While improving the application performance is important, having solid power efficiency is equally important. To evaluate power overhead, we launch the above five apps every ten minutes and issue 100 user events via Android Monkey, with and without iRAM. The battery on Samsung S4 is fully charged at the beginning. We record how long the battery lasts for each case. The results are presented in Figure 5.11(c). While without iRAM, the battery lasts 10.54 hours (10 hours 32 minutes). With iRAM, the battery lasts 10.44 hours (10 hours 26 minutes). This implies that the power overhead of iRAM is 0.9 percent, which is acceptable.

5.7 Discussion and Future Work

Launch delay is critical to user experience, since one launches apps repeatedly throughout the day. However, in future work we plan to evaluate the impact of other stages of the life cycle on application performance such as install, update, switch, and uninstall, and quantify their effects on everyday phone usage. We intend to extend this study by researching how other common usage patterns are impacted. For instance, taking photos, recording movies, messaging, calling, email sync (recently studied in [70]), etc.

Our work only focuses on improving application performance with respect to RAM. It may be also interesting to study how different applications use main memory and flash disks. Nguyen et al. [61] already researched flash storage I/O behaviors, and designed a system that improved application response times by prioritizing reads over writes. However, with the decreasing price of flash, it will be useful to investigate opportunities for performance enhancements through hybrid solutions that consider both areas.

5.8 Conclusion

This chapter presents iRAM, a system that maintains optimal heap size limits to avoid crashes, efficiently maximizes free memory levels, and cleans low-priority processes to reduce application delays. The evaluation indicates that iRAM reduces application crashes by up to 14 percent, reduces launch delays by up to 78.2 percent, and increases free memory levels by up to 4.8 times.

Chapter 6

Conclusion and Future Work

This dissertation presented three system prototypes contributing to improving smartphone energy saving and reducing application delay.

First, we proposed a system called SmartStorage that dynamically tuned storage parameters to reduce energy consumption by matching the current I/O pattern to a known pattern that we recorded from eight selected benchmarks. We validated our dynamic tuning technique by showing that SmartStorage saved 23 to 52 percent of the energy consumption by running SmartStorage in the background with selected applications from the top 20 most popular apps in the foreground.

Next, we presented a measurement study on the behavior of reads and writes in smartphones. Among others, we observed that reads experienced up to a 626 percent slowdown in the presence of concurrent writes. The obtained insights were used to design and implement a system that reduced the application delay by prioritizing reads over writes, and grouping them based on assigned priorities. The evaluation on 40 apps demonstrated that SmartIO reduced launch delays by up to 37.8 percent.

Finally, we presented iRAM, a system that maintained optimal heap size limits to avoid crashes, efficiently maximized free memory levels, and cleaned low-priority processes to reduce application delays. The evaluation indicated that iRAM reduced application crashes by up to 14 percent, reduced launch delays by up to 78.2 percent, and increased

free memory levels by up to 4.8 times.

For future work, we are considering three following research directions:

- **An Integrated Storage and Performance-Aware Smartphone Energy Saving System**

Rather than separately answering the research questions on smartphone energy efficiency and performance impact of our SmartStorage system, we will extract out principles from the research results and integrate them into a prototype system. This system will consist of three layers: (1) the application classification layer that will classify applications based on performance tolerance; (2) the storage I/O matching layer that will fingerprint applications' storage I/O activities for locating the most energy efficient storage policies; and (3) the storage configuration layer that will use optimal storage policies for different applications so as to save smartphone energy while providing high performance. The system will be thoroughly tested with existing and future smartphones. User study will also be conducted to evaluate and assist in improving user experience and acceptance of our prototype system.

- **Advanced iRAM**

We plan to extend our iRAM work with several significant improvements. 1) *Not static memory thresholds but dynamic.* iRAM provides a set of configurations that allows choosing the best policy. This includes the aggression level, minimal memory threshold, heap size, and other parameters. These thresholds should not be static but rather dynamic. 2) *Not manual white list management but automatic.* iRAM gives the highest priority to a set of selected processes in a white list that is managed manually. This should be managed automatically by a learning algorithm that would rate the importance of the processes based on the frequency it is being launched or other performance parameters. 3) *Not periodic heap usage tracking but adaptive.* iRAM maintains the global limit of the heap size available to each application/process. The system tracks the run-time heap usage of selected

apps with the highest priority. If the heap usage of any high priority app reaches the limit, the limit is increased incrementally by 10 percent. Heap size tracking should not be performed periodically but rather adaptively. For instance, when the device is in the sleep mode, tracking does not have to be often.

- **Long-term Direction: Beyond Performance, Impact on Human's Everyday Life**

In this dissertation, we studied the impact of various layers (cache, block layer, device driver, and RAM) on smartphone energy and application performance. In the long-term, we plan to investigate other layers and subsystems such as filesystem, CPU, and network subsystem. Next, this dissertation only focuses on launch and run-time delay performance that is critical to user experience, since one launches and runs apps repeatedly throughout the day. However, in future work we intend to go beyond performance, and plan to evaluate the impact of our technology on human's everyday life and seek opportunities to improve the overall phone usage experience. Specifically, we plan to extend this study by researching how other common usage patterns are impacted and how they can be improved. For instance, taking photos, recording movies, messaging, calling, and emailing.

Bibliography

- [1] Changewave research. <http://www.changewaveresearch.com>, 2011.
- [2] Androbench benchmark. <http://www.androbench.org/wiki/AndroBench>, 2012.
- [3] Antutu benchmark. <http://www.antutu.com>, 2012.
- [4] Block i/o layer tracing: blktrace. <http://linux.die.net/man/8/blktrace>, 2012.
- [5] Browsermark benchmark. <http://browsermark.rightware.com/>, 2012.
- [6] Cf-bench benchmark. <http://bench.chainfire.eu/>, 2012.
- [7] Glbenchmark. <http://www.glbenchmark.com/>, 2012.
- [8] Quadrant benchmark. <http://www.aurorasoftworks.com/products/quadrant>, 2012.
- [9] Smartbench benchmark. <http://www.1mobile.com/smartbench-2012-327800.html>, 2012.
- [10] Vellamo benchmark. <http://www.quicinc.com/vellamo/>, 2012.
- [11] Busybox. <http://goo.gl/CF6vJ>, 2014.
- [12] Deadline io scheduler tunables. <http://goo.gl/mB9a1K>, 2014.
- [13] fio: Flexible io tester ported for android. <http://storebench.com/fio.html>, 2014.
- [14] iostat. <http://goo.gl/0tZ33>, 2014.

- [15] Modified facebook application demo. <http://goo.gl/b1AxQ2>, 2014.
- [16] Monkey. <http://goo.gl/F14hW>, 2014.
- [17] Monsoon monitor. <http://www.msoon.com>, 2014.
- [18] One quarter of work devices are smartphones and tablets, forrester finds. <http://goo.gl/K23yGu>, 2014.
- [19] Storebench download. <http://goo.gl/ava9eV>, 2014.
- [20] Storebench web. <http://StoreBench.com>, 2014.
- [21] Storebench's list of devices. <http://StoreBench.com/list.html>, 2014.
- [22] Time man page. <http://goo.gl/dEKuxs>, 2014.
- [23] Worldwide smartphone 2013-2017 forecast and analysis. <http://goo.gl/v5vg2b>, 2014.
- [24] Worldwide smartphone users cross 1 billion mark: Report. <http://www.ibtimes.com>, 2014.
- [25] Angry birds. <http://goo.gl/GVnfxL>, 2015.
- [26] Any do task list todo list. <http://goo.gl/ciEA8h>, 2015.
- [27] Dropbox. <http://goo.gl/P3VjMV>, 2015.
- [28] Dumpsys meminfo. <http://goo.gl/i6qCvt>, 2015.
- [29] Google calendar. <http://goo.gl/Nswmvx>, 2015.
- [30] Linux free command. <http://linux.die.net/man/1/free>, 2015.
- [31] Mobile apps: What consumers really need and want. <http://goo.gl/VcE59W>, 2015.
- [32] Mobile experience benchmark. <http://goo.gl/0ebwYp>, 2015.

- [33] Nielsen: U.s. consumers avg app downloads up 28<http://goo.gl/3ySv0m>, 2015.
- [34] Processes and threads. <http://goo.gl/KraoUH>, 2015.
- [35] Worldwide smartphone growth forecast to slow from a boil to a simmer as prices drop and markets mature, according to idc. <http://goo.gl/bmpbCQ>, 2015.
- [36] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX ATC 2008*.
- [37] Jens Axboe. Linux block io-present and future. In *Ottawa Linux Symp 2004*.
- [38] Jens Axboe. fio: Flexible io tester. <http://linux.die.net/man/1/fio>, 2014.
- [39] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3g using wifi. In *ACM MobiSys 2010*.
- [40] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc. IMC 2009*. ACM Press, 2009.
- [41] Simona Boboila and Peter Desnoyers. Performance models of flash-based solid-state drives for real workloads. In *IEEE MSST 2011*.
- [42] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3 edition, 2005.
- [43] Alan D. Brunelle. *blktrace User Guide*. USA, 2007.
- [44] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181--186. ACM, 1991.
- [45] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proc. ATC 2010*. USENIX Assoc., 2010.

- [46] Rajiv Chakravorty, Suman Banerjee, Pablo Rodriguez, Julian Chesterfield, and Ian Pratt. Performance optimizations for wireless wide-area networks: Comparative study and experimental evaluation. In *ACM MobiCom 2004*.
- [47] Mun Choon Chan and Ramachandran Ramjee. Tcp/ip performance over 3g wireless links with rate and delay variation. In *ACM MobiCom 2002*.
- [48] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS 2009*.
- [49] John G Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4):396--402, 1984.
- [50] Marcus Paul Dunn. *A new I/O scheduler for solid state devices*. PhD thesis, Texas A&M University, 2009.
- [51] A. Gupta and P. Mohapatra. Energy consumption and conservation in wifi based phones: A measurement-based study. In *Proc. SECON 2007*. IEEE, 2007.
- [52] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z. Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing application performance differences on smartphones. In *ACM MobiSys 2010*.
- [53] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Nonintrusive and online power analysis for smartphone hardware components. Technical report, 2012.
- [54] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. In *Proc. FAST 2012*. USENIX Assoc., 2012.
- [55] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):14, 2012.

- [56] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Disk schedulers for solid state drivers. In *ACM EMSOFT 2009*.
- [57] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware i/o management for solid state disks (ssds). In *IEEE Transactions on Computing 2012*.
- [58] Kisung Lee and Youjip Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *ACM EMSOFT 2012*.
- [59] R. Murmura, J. Medsger, A. Stavrou, and J.M. Voas. Mobile application and device power usage measurements. In *Proc. SERE 2012*. IEEE, 2012.
- [60] David T Nguyen, Gang Zhou, Xin Qi, Ge Peng, Jianing Zhao, Tommy Nguyen, and Duy Le. Storage-aware smartphone energy savings. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 677--686. ACM, 2013.
- [61] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceeding of the 13th annual international conference on Mobile systems, applications, and services*. ACM, 2015.
- [62] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 275--284. ACM, 2013.
- [63] Stan Park and Kai Shen. Fios: A fair, efficient flash i/o scheduler. In *USENIX FAST 2012*.
- [64] Andrew Pyles, Xin Qi, Gang Zhou, Matthew Keally, and Xue Liu. Sapsm: Smart adaptive 802.11 psm for smartphones. In *Proc. UbiComp 2012*. ACM Press, 2012.

- [65] A. Riska, J. Larkby-Lahet, and E. Riedel. Evaluating block-level optimization through the io path. In *Proc. ATC 2007*. USENIX Assoc., 2007.
- [66] Sayandeep Sen, Neel Kamal Madabhushi, and Suman Banerjee. Scalable wifi media delivery through adaptive broadcasts. In *USENIX NSDI 2010*.
- [67] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *USENIX ATC 2013*.
- [68] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. Livelab: measuring wireless networks and smartphone users in the field. *ACM SIG-METRICS Performance Evaluation Review*, 38(3):15--20, 2011.
- [69] D.I. Shin, Y.J. Yu, H.S. Kim, H. Eom, and H.Y. Yeom. Request bridging and interleaving: Improving the performance of small synchronous updates under seek-optimizing disk subsystems. *ACM Transactions on Storage (TOS)*, 2011.
- [70] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing background email sync on smartphones. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 55--68. ACM, 2013.
- [71] Qiang Xu, Sanjeev Mehrotra, Zhuoqing Mao, and Jin Li. Proteus: Network performance forecast for real-time, interactive mobile applications. In *ACM MobiSys 2013*.
- [72] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 113--126. ACM, 2012.
- [73] Zhenyun Zhuang, Tae-Young Chang, Raghupathy Sivakumar, and Aravind Velayutham. A 3: Application-aware acceleration for wireless data networks. In *ACM MobiCom 2006*.