INFORMED STORAGE MANAGEMENT FOR MOBILE PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Hyojun Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2012

INFORMED STORAGE MANAGEMENT FOR MOBILE PLATFORMS

Approved by:

Professor Umakishore Ramachandran,
Committee Chair
College of Computing
Georgia Institute of Technology

Umakishore Ramachandran, Advisor
College of Computing
Georgia Institute of Technology

Professor Ling Liu
College of Computing
Georgia Institute of Technology

Professor Marilyn Wolf
College of Computing / School of
Electrical and Computer Engineering
Georgia Institute of Technology

Professor Moinuddin K. Qureshi
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Professor Remzi H. Arpaci-Dusseau
Department of Computer Sciences
University Wisconsin-Madison

Date Approved: August 6, 2012

TABLE OF CONTENTS

SUMMARY

Smartphones are not just ordinary phones; they are being used as mobile platforms for serving the computing needs of a significant segment of the user community. We believe that storage in smartphones is an Achilles' heel to achieving the full performance potential of smartphones for meeting the future computing needs of the user community that is starting to become more heavily dependent on mobile platforms. The poor performance of storage in smartphones can be attributed to the fact that smartphones typically employ flash storage, and the OS storage stack is not optimized to deal with the performance quirks of such storage.

We have proposed multiple storage software solutions to handle the durability issue and to improve the performance of Flash storage. There are always design tradeoffs involved in building software systems. RAM based write buffering can enhance storage performance, but sacrifices reliability. Redundancy (via replication) can improve data availability, but it has implications for data consistency. Log-structured design to combat the "small write" problem solves the random write issue in Flash storage, but introduces overhead for maintaining mapping information between logical and physical blocks.

In this thesis, entitled *Informed Storage Management (ISM)*, we aim at providing a dynamic decision-making framework for system design, specifically targeted to storage systems on mobile platforms. The goal of ISM is maximizing the performance benefits while minimizing the side effects of the design choice. As a concrete example of ISM, we provide mechanisms along three axes, namely, *type*, *temporal*, and *spatial* for selectively supporting write-back buffering, which can be used judiciously by the upper layers of the operating system. We implement and evaluate our solution on a real Android smartphone, and demonstrate significant performance gains for everyday apps on such platforms.

CHAPTER I

INTRODUCTION

Smart phones offer the potential for a spectrum of benefits to the society ranging from commerce and entertainment, to managing disasters. According to recent statistics [29, 80, 107], there are up to 4 billion cell phone users (more than half of the world's population), with accelerating penetration in regions such as Africa. The role played by smart phones in sparking the recent spate of revolutions against authoritative regimes in the Middle East underscores the increasing importance of mobile platforms in redefining the computing landscape.

A significant percentage of the population in the developed world relies on mobile devices as their primary source of information access. This trend is even more significant in the light of predictions about mobile devices dominating most personal computing landscape in the near future. As we all know, mobile platforms extend beyond smartphones. The price point for mobile platforms has dropped considerably due to technological advances. Consequently, we can see a variety of mobile platforms for dedicated uses popping up all the time including *e-books*, *tablets*, and *3D-gaming consoles*. One could see that mobile platforms are starting to penetrate the very fabric of society at least in the developed countries. Due to its very nature and due to the demographic diversity of user population using such mobile platforms ranging from children to grandmothers, low power consumption, low price, and ruggedness of the operating system have become important considerations in the design of such platforms. Interestingly, these design considerations make such platforms ideal for the deeper penetration of computing into developing countries as well. Further, well-designed mobile platforms may even become attractive server platforms in the future.

These are exciting times for advances in mobile technology. Smart phones and Tablets pack a lot of computational power. For example, Samsung Galaxy III uses Quad-core 1.4 GHz CPU and 2 GB RAM. The tablet-cum-notebook from Asus (Transformer Prime) uses Nvidia's Tegra 3 Quad-core CPU (Single core mode 1.4GHz, multi core mode 1.3GHz) and 1GB RAM.

At the same time, it is also the case that mobile platforms use low-end flash storage for reasons of power efficiency, size, weight, and ultimately cost. For example, raw NAND flash memory is the primary store in many devices including Google Nexus One, HTC desire, One-Laptop-Per-Child (OLPC-XO)-V1 (1GB), Apple iPhone and iPad series. eMMC is the primary storage technology in devices such as Google Nexus S, Kindle Fire, and OLPC XO laptop-V1.75 (4GB/8GB). MicroSDHC cards are used as secondary storage in many smart phones and Internet tablets (e.g., OLPC XO laptop-V1.5 - 4GB). The capacity of such low-end flash storage is increasing with technological advances but their performance and reliability continue to be an issue due to the very nature of the technology (e.g., poor random write performance, wear-out issue). Data centers and high-end servers routinely use high-end *solid state disks (SSDs)* built with flash memories. They are bigger, more expensive, more power hungry, and are also susceptible to sudden power failure. Such high-end SSDs are not appropriate for mobile platforms.

One would be tempted to think the primary performance bottleneck in mobile platforms is the network connectivity. We have proven that storage is a huge performance contributor even for mobile workloads [61, 62]. Using typical mobile apps such as Web browser, Maps, facebook, email, and news clipping, the studies show that the performance on an Android smart phone is storage-bound and not network-bound (with two different network connections – wired and wireless).

As we move forward to relying more on mobile devices for everyday computing activities, storage performance will become more critical with workloads that are typically considered today as desktop or server applications. It is inevitable that users will force such use cases on mobile platforms since technological advances have made the CPU, memory, and network connectivity in today's mobile platforms comparable to desktop/server systems of yesterday.

The focus of this thesis is to advance the state-of-the-art for storage technology aimed at future mobile platforms. The reality is this research space is very sparsely represented by the academic community. Device manufacturers (such as Samsung, Toshiba, Micron, and SanDisk) are focused on enhancing the device itself and are agnostic to what happens on the host side operating system, since they simply present a traditional disk-like block-device interface to the host. Enterprise storage systems (from vendors such as IBM, NetApp, and EMC) use flash memory internally but focus on high-end SSDs for use in data centers and storage servers (i.e., such research and development do not impact the client-side mobile platforms). Mobile OS vendors (such as Google, Apple, Microsoft, and RIM) are interested in this space but their efforts are usually shrouded in secrecy, and from what is emanating into the open community it is clear that their focus is in fine-tuning the performance of the *current* mobile workloads with *existing* mature storage technologies.

1.1 Problem Statement

To remove the performance bottleneck in the storage of mobile platforms, operating system level software support is critically needed, and we target three problems.

First, flash storage exhibits very different performance characteristics relative to the traditional HDD, but current operating systems (owing to their legacy of assuming HDD as the primary storage technology) are not engineered to support flash storage adequately.

Second, mobile platforms are used in much rugged environment than traditional computer systems. For instance, we carry smartphones in our pockets always, and smartphones may easily be “dropped” on the ground. In such situations, the battery can get separated from the phone, and thus, smartphones have higher chances to lose system power unexpectedly than regular computer systems do. Therefore, the storage software stack tends to be configured for high reliability and performance is sacrificed. Meanwhile, cloud computing and ubiquitous computing are very popular today, and for some applications, performance is more important than reliability in local storage because the local storage is used as just a cache store. The current reliability vs. performance controlling mechanisms are too coarse; the same system-wide configurations are used both for critical and non-critical files in each storage layer, and we lose performance unfairly.

Last, flash aware solutions and fine-grained reliability controlling mechanisms are not enough by themselves. We need to “properly” and “wisely” control them; for the purpose, system wide information can be collected and used, but right now, we do not have such methods.

1.2 Thesis Statement

There are always design tradeoffs involved in building software systems. Usually, system design chooses a “sweet spot” that optimizes the solution for meeting certain requirements and/or assumptions about the environment (application behavior, device characteristics, etc.). Unfortunately, since the real world use cases are very dynamic and the technology landscape is continually evolving, often such assumptions may turn out to be incorrect; further, statically fixing the design based on certain requirements may force the design to be conservative. This argues for the need to adjust the solution dynamically based on the use case. the technological evolution, and the workload.

With this background, we state our thesis as *“By mining and exploiting system* *wide information, we can improve the performance of storage system on mobile platforms without losing reliability."*

1.3 Organization

To support our thesis statement, we first understand the state-of-the art studies in Chapter II. Chapter III shows our motivating observation - how much the smartphone storage can influence application performance. In following two chapters, our two flash storage solutions (selective logging - Chapter IV and flash aware buffer replacement scheme - Chapter V) are explained. Then, our integrated storage solution, named Fjord, will be introduced in Chapter VI. In Chapter VII, we review the our work and provide some insights for further study. Finally, we conclude in the last Chapter VIII.

CHAPTER II

BACKGROUND AND RELATED STUDIES

2.1 Flash Storage

Flash storage has become one of the most important players in today's computer systems. Despite the fact that magnetic disk is well entrenched in the storage market, Flash storage is attractive for a number of reasons: it is small, lightweight, shock resistant, and energy efficient. These characteristics make Flash storage attractive for enterprise storage systems, which demand extremely high performance, and also for mobile platforms, which have special requirements.

Flash storage is based on the semiconductor technology, and hence shows very different performance characteristics as compared to traditional magnetic, rotating storage devices. Flash memories, including NAND and NOR types, have a common physical restriction, namely, they must be erased before writing [76]. In flash memory, the existence of an electric charge in a transistor represents 1 or 0. The charges can be moved both into a transistor by an erase operation and out by a write operation. By design, the erase operation, which sets a storage cell to 1, works on a bigger number of storage cells at a time than the write operation. Thus, flash memory can be written or read a single page at a time, but it has to be erased at a time in units of an erasable-block. An erasable-block consists of a certain number of pages. The size of a page ranges from a word (NOR flash memory) to 4 KB depending on the type of the device. In NAND flash memory, a page is similar to a hard disk sector and is usually 2 or 4 KB. Flash memory also suffers from a limitation on the number of erase operations possible for each block. The insulation layer that prevents electric charges from dispersing may be damaged after a certain number of erase operations. In single

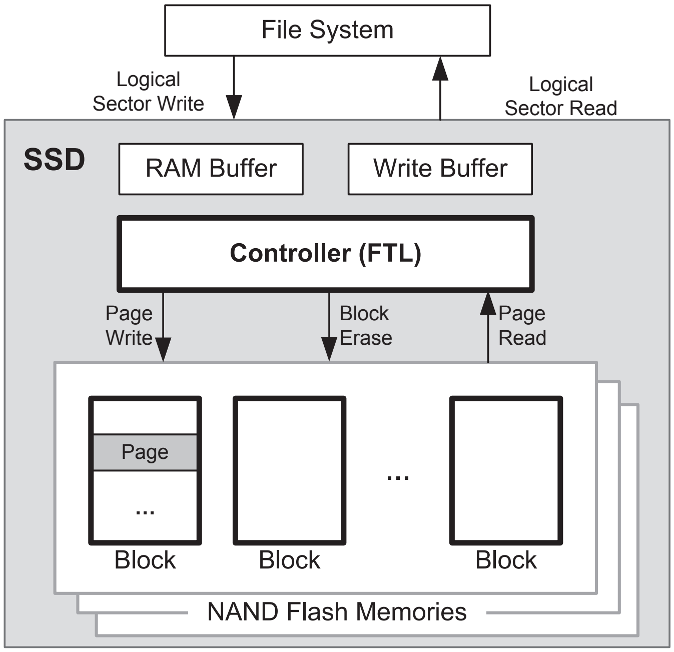


Figure 1: SSD, FTL and NAND flash memory: *FTL emulates sector read and write functionalities of a hard disk allowing conventional disk file systems to be implemented on NAND flash memory*

level cell (SLC) NAND flash memory, the expected number of erasures per block is 100,000 and this is reduced to 1,000 in triple bits multilevel cell (TLC) NAND flash memory. If some blocks that contain critical information are worn out, the whole memory becomes useless even though many serviceable blocks still exist. Therefore, many flash memory-based devices use wear-leveling techniques to ensure that blocks wear out evenly [30].

An SSD (see Figure 1) is simply a set of flash memory chips packaged together with additional circuitry and a special piece of software called flash translation layer (FTL) [22, 49]. The additional circuitry may include a RAM buffer for storing meta-data associated with the internal organization of the SSD, and a write buffer for optimizing the performance of the SSD. The FTL provides an external logical interface to the file system. A sector[1] is the unit of logical access to the flash memory provided

---

[1] Even though the term *sector* represents a physical block of data on a hard disk, it is commonly used as an access unit for the FTL because it emulates a hard disk.

by this interface. A page inside the flash memory may contain several such logical sectors. The FTL maps this logical sector to physical locations within individual pages [22]. This interface allows FTL to emulate a hard disk so far as the file system is concerned (Figure 1).

By embedding FTL software inside, Flash storage devices can provide the same functionalities of an HDD, but their performance characteristics are very different compared to HDDs. A number of studies have been conducted to enumerate the special characteristics of Flash storage [22, 31, 92], and perhaps, the most important point of these studies is that Flash storage is different.

2.2 Flash Based Storage Solutions

2.2.1 Key-Value Store

Flash memory has brought about a drastic change in storage technology recently. Some studies propose totally new storage systems using flash memory. In FAWN (a Fast Array of Wimpy Nodes) [24], a new distributed storage architecture has been proposed to provide an efficient, fast, and cost-effective key-value store with low-end CPUs and wimpy flash devices. FAWN pairs low-power embedded nodes with flash storage, and it is especially designed to save power consumption.

Another key-value store, named FlashStore [36] has been proposed as a high throughput persistent key-value store. Both FAWN and FlashStore are based on the log-structured architecture, but FlashStore uses Flash storage as a cache for hard disk drives while FAWN applies log-structured architecture for the whole storage.

Small Index Large Table (SILT) [73] is a new flash-based key-value storage system that significantly reduces per-key memory consumption with predictable system performance and lifetime. SILT requires approximately 0.7 bytes of DRAM per key-value entry and uses on average only 1.01 flash read operations to handle lookup tasks. Consequently, SILT can saturate the random read I/O on author's experimental system, performing 46,000 lookups per second for 1024-byte key-value entries, and it can potentially scale to billions of key-value items on a single host.

Key-value store has become very important today in large scale data intensive applications, and may not have much of a relevance at this point of time for mobile platforms.

2.2.2 Flash-HDD Hybrid Storage

Griffin [104] system has been proposed as a Flash - Hard disk drive hybrid storage solution like FlashStore, but the approach is very unique. In general, Flash memory is used as a cache for a HDD because Flash is faster while HDDs are cheaper and bigger. However, in Griffin, the roles of HDD and Flash are reversed; an HDD is used as a write buffer for Flash storage to extend the lifetime of flash storage. Even though their approach is interesting and evaluation results are promising, this approach will not be suitable for mobile platforms for multiple reasons; HDDs are big and heavy, consume much power, and expensive. In Griffin, the authors explicitly mention that they ruled out the RAM buffering approach due to the reliability concern. However, we believe that RAM buffering has many attractive merits compared to disk buffering and proper design of RAM buffering can overcome the reliability issue.

FlashCache [105] is a write back block cache module developed and used by Facebook. It accelerates reads and writes from slower rotational media by caching data in SSDs, and is based on the Linux device mapper mechanism [86].

Similarly, Solaris ZFS [79] can use SSDs as a cache store for HDDs. Unlike traditional volume based file systems, ZFS is designed based on Pool, and users do not have to manage partitions.

Due to the cost and size benefits of Flash SSDs, many enterprise storage systems are actively adopting Flash storage in their systems today as a cache [79, 41, 105] or a faster tier storage (EMC FAST [40], IBM Easy Tier [48], Compellent Data Progression systems [35]). Again, the approach is not applicable for mobile platforms considering the size, cost, and power limitations.

2.3 Storage Software Stack Modification for Flash

2.3.1 Flash Aware File Systems

As is evident, most random writes stem from the well-known "small write" problem in file systems. Log-structured file system [90] was proposed as a solution to the small write problem, and it is a promising approach for SSD-based file systems as well since it can change random writes to sequential writes effectively. JFFS2 [87] and YAFFS [74] are well-known log-structured file systems working on Memory Technology Devices (MTDs), and NILFS [70] is for regular disks including HDDs and SSDs. However, due to the log-structured nature, such file systems have expensive garbage collection and scalability issues.

A few years ago, raw NAND flash memories were popularly used in mobile platforms with YAFFS2 file system. Today, eMMC devices are more popular in mobile platforms, and EXT4 file system is being used instead of YAFFS2.

2.3.2 I/O Scheduler

New I/O scheduling algorithms have been proposed for flash storage. Kim et al. proposed the Individual Read Bundled Write (IRBW) algorithm which separates read scheduling from write requests and arranges write requests into bundles [68]. The algorithm is based on the observation of SSD performance characteristics; read request service time is almost constant while write request service time is not. Moreover, appropriate grouping of write requests eliminates any ordering-related restrictions and also maximizes write performances. The proposed I/O scheduler arranges write requests into bundles of an appropriate size while read requests are independently scheduled.

Dunn and Reddy also proposed a new Block Preferential I/O scheduler for flash storage [39] based on similar observations. In their study, a new framework has been proposed to find out the FTL mapping size (i.e. block size) within an SSD, and the new I/O scheduler gives higher priority to the requests that are in the same block as the previous request.

A Fair, Efficient Flash I/O Scheduler (FIOS) [84] is another one for Flash storage. While the previous two I/O schedulers are focusing only on I/O throughput, FIOS tries to improve Flash I/O fairness and efficiency. When there is a concurrent workload with a mixture of readers and synchronous writers running on Flash, readers may be blocked by writes with substantial slowdown. This means unfair resource utilization between readers and writes, and FIOS employs read preference to minimize read-blocked-by-write in concurrent workloads.

Even though these I/O scheduling schemes reflect the characteristics of flash devices quite well, the queuing mechanism of I/O schedulers limits the performance gain. In other words, the number of requests in the queue limits the capability of the I/O scheduler, and this number is not big in general; the maximum queue size is only 128 by default in most Linux distributions.

2.3.3 Page and Buffer Cache Management

Ever since the appearance of NAND flash memory based solid state storage devices, multiple flash-aware buffer cache replacement schemes have been proposed. One of the earliest schemes is CFLRU [83](Figure 2). Due to its very nature, flash memory incurs less time for servicing a read operation in comparison to a write operation. In other words, to reduce the total I/O operation time of flash, it is desirable to reduce the total number of write operations. To this end, CFLRU tries to evict a clean page rather than a dirty page because a dirty page must be written back to the storage during the eviction process. However, such a biased policy could fill the cache with mostly dirty pages at the expense of not holding frequently accessed clean pages. This is clearly undesirable as it will bring down the cache hit-ratio overall. To mitigate this problem, CFLRU divides the LRU cache list into parts (as done in the 2Q algorithm [57]), and applies the clean-first policy only to the lower part of LRU list. The authors claim that choosing the partition size intelligently will result in shorter total operation time for flash storage.

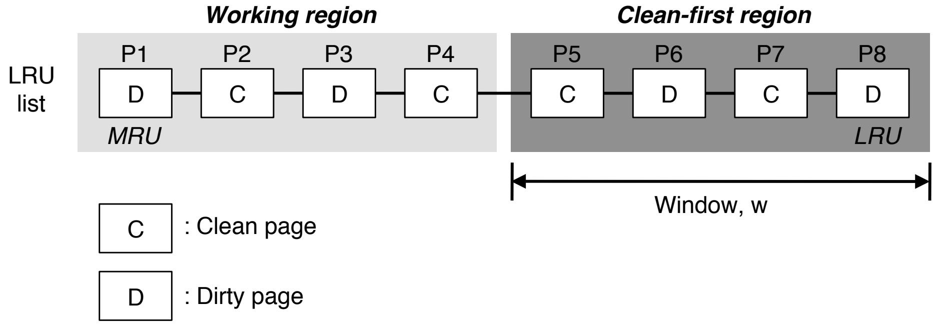


Figure 2: Example of CFLRU algorithm [83]

LRUWSR [58] shares the same motivation with CFLRU. It also tries to give higher priority to dirty pages, but it uses a different method. Instead of partitioning the LRU list into two parts, LRUWSR adds a *cold bit* to each cache frame, and gives a second chance to a dirty page frame to remain in the cache. When a page is selected as a potential victim, its dirty and cold bits are checked first. If the page is dirty and its cold bit is zero, then the algorithm decides not to choose it as a victim. Instead it sets the cold bit for this page to indicate that the page has gotten its second chance to stay in the cache. The authors argue that LRUWSR is better than CFLRU both in terms of usability (because it does not require any workload dependent parameters) and performance.

One of the latest flash-aware cache replacement schemes is FOR [75]. It also focuses on the asymmetric read and write operation time of flash storage. In addition, it combines *inter operation distance (IOD)* (the core idea of the Low Inter Reference Recency [54]) together with the recency attribute of the LRU algorithm. Further, FOR calculates the IOD and recency values separately for read and write operations. By considering all these factors, FOR calculates the weight of each page, and it evicts the page having the minimal weight value. The results reported by the authors show 20% improvement for database workloads for FOR over other schemes.

2.3.4 New Interface

Even though NAND flash memory has very different characteristics to conventional magnetic storage, flash based SSDs export the same block-level read and write APIS as hard disks do for compatibility with current systems. Some studies argue that there is a lost opportunity for proposing new abstractions that better match the nature of the new medium. In Transactional Flash (TxFlash) study [85], authors proposed to extend SSD interface to support atomic writes, which will be useful for building file systems as well as database systems.

As a similar approach, Nameless Write study [110] has been proposed to allow flash device to choose the location of a write. This approach can eliminate the need for indirection in modern SSDs.

These approaches are promising and desirable from a long-term view. However, changing the interface is not easy, and it requires broad industry consensus and massive restructuring of the existing infrastructure. Besides, the changed interface should be general enough to support all kinds of storage.

2.4 Flash Device Level Studies

At the device level, more complicated FTL mapping algorithms have been proposed to attain better write performance [82, 81]. However, due to the increased resource usage of these approaches, they are generally used only for high-end SSDs.

Incorporating a write-buffer inside a flash storage device is a slightly higher-level approach than FTL. For example, we previously proposed Block Padding Least Recently Used (BPLRU) [63] as a buffer management scheme for flash storage and showed that even a small amount of RAM-based write buffer could enhance the random-write performance of flash storage devices significantly.

There is a clear difference between a device-level solution and an OS-level solution in terms of generality. OS-level approach is useful especially for low-end flash storage devices, which suffer from limited resources.

2.5 Informed Storage

Semantically smart Disk systems study [102, 25] shows how the high level information can be used to improve the low level storage performance. File system level information can be inferred within a semantically smart storage device, and can be used in multiple cases: track-aligned extents, structural caching, secure deletion, etc.

The smart disk studies focused on the "information gap" problem between file system and storage device, and tried to solve the problem within the constraints of the real world; in other words, without changing the block level interface.

In our Informed Storage Management (ISM) study, we also claim that system wide information can be useful for the storage sub system. In addition to the information flow between file system and storage device, we include other types of information in our framework such as application's characteristics, user's preference, and sensor information.

CHAPTER III

REVISITING STORAGE FOR SMARTPHONES

In this chapter, we briefly explain our study [62] [1] showing the important role of storage sub system in mobile platforms. Contrary to conventional wisdom, we find evidence that storage is a significant contributor to application performance on mobile devices.

3.1 Introduction

Storage has traditionally not been viewed as a critical component of phones, tablets, and PDAs – at least in terms of the expected performance. Despite the impetus to provide faster mobile access to content locally [46] and through cloud services [98], performance of the underlying storage subsystem on mobile devices is not well understood. Our work started with a simple motivating question: does storage affect the performance of popular mobile applications? Conventional wisdom suggests the answer to be *no*, as long as storage performance exceeds that of the network subsystem. We find evidence to the contrary – even interactive applications like web browsing slow down with slower storage.

Storage performance on mobile devices is important for end-user experience today, and its impact is expected to grow due to several reasons. First, emerging wireless technologies such as 802.11n (600 Mbps peak throughput) [109] and 802.11ad (or "60 GHz", 7 Gbps peak throughput) offer the potential for significantly higher network throughput to mobile devices [47].

Figure 3.1 presents the trends for network performance over the last several

---

[1] This work was presented at the USENIX FAST'12 conference.

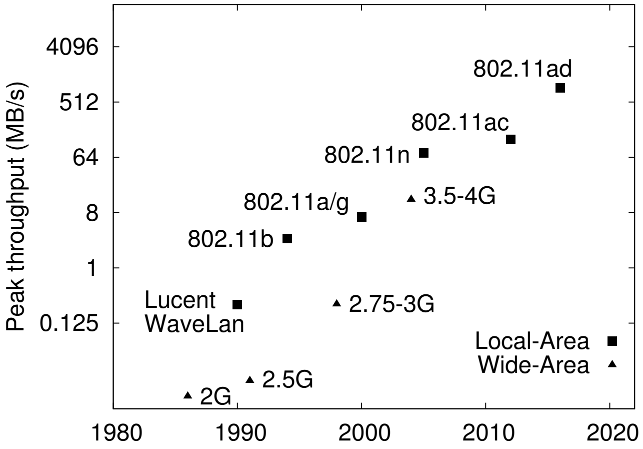


Figure 3: Peak throughput of wireless networks, Trends for local and wide-area wireless networks over past three decades; y-axis is log base 2.

decades; local-area networks are not necessarily the de-facto bottleneck on modern mobile devices. Second, while network throughput is increasing phenomenally, latency is not [99]. As a result, access to several cloud services benefits from a *split* of functionality between the cloud and the device [33], placing a greater burden on local resources including storage [71]. Third, mobile devices are increasingly being used as the primary computing device, running more performance intensive tasks than previously imagined. Smartphone usage is on the rise; smartphones and tablet computers are becoming a popular replacement for laptops [20]. In developing economies, a mobile/enhanced phone is often the only computing device available to a user for a variety of needs.

In this study, we present a detailed analysis of the I/O behavior of mobile applications on Android-based smartphones and flash storage drives. We particularly focus on popular applications used by the majority of mobile users, such as, web browsing, App install, Google Maps, Facebook, and email. Not only are these activities available on almost all smartphones, but they are done frequently enough that performance problems with them negatively impacts user experience. Further, we provide pilot solutions to overcome existing limitations.

To perform our analysis, we build a measurement infrastructure for Android consisting of generic firmware changes and a custom Linux kernel modified to provide resource usage information. We also develop novel techniques to enable detailed, automated, and repeatable measurements on the internal and external smartphone flash storage, and with different network configurations that are otherwise not possible with the stock setup; for automated testing with GUI-based applications, we develop a benchmark harness using MonkeyRunner tool.

In our initial efforts, we propose and develop a set of pilot solutions that improve the performance of the storage subsystem and consequently mobile applications. Within the context of our Android environment, we investigate the benefits of employing a small amount of phase-change memory to store performance critical data, a RAID driver encompassing the internal flash and external SD card, using a log-structured file system for storing the SQLite databases, and changes to the SQLite fsync code-path. We find that changes to the storage subsystem can significantly improve user experience; our pilot solutions demonstrate possible benefits and serve as references for deployable solutions in the future.

As the popularity of Android-based devices surges, the setup we have examined reflects an increasingly relevant software and hardware stack used by hundreds of millions of users worldwide; understanding and improving the experience of mobile users is thus a relevant research thrust for the storage community. Through our analysis and design we make several observations:

Storage affects application performance: often in unanticipated ways, storage affects performance of applications that are traditionally thought of as CPU or network bound. For example, we found web browsing to be severely affected by the choice of the underlying storage; just by varying the underlying flash storage, performance of web browsing over WiFi varied by 187% and over a faster network (setup over USB) by 220%. In the case of a particularly poor flash device, the variation exceeded 2000% for WiFi and 2450% for USB.

Speed class considered irrelevant: our benchmarking reveals that the "speed class" marking on SD cards is not necessarily indicative of application performance; although the class rating is meant for sequential performance, we find several cases in which higher-grade SD cards performed worse than lower-grade ones overall.

Slower storage consumes more CPU: we observe higher total CPU consumption for the same application when using slower cards; the reason can be attributed to deficiencies in either the network subsystem, the storage subsystem, or both. Unless resolved, lower performing storage not only makes the application run slower, it also increases the energy consumption of the device.

Application knowledge ensues efficient solutions: leveraging a small amount of domain or application knowledge provides efficiency, such as in the case of our pilot solutions; hardware and software solutions can both benefit from a better understanding of how applications are using the underlying storage.

Based on our experimental findings and observations we believe improvements in the mobile storage stack can be made along multiple dimensions to keep up with the increasing demands placed on mobile devices. Storage device improvements alone can account for significant improvements to application performance. Device manufacturers are actively looking to bring faster devices to the mobile market; Samsung announced the launch of a PCM-based multi-chip package for mobile handsets [96]. Mobile I/O and memory bus technology needs to evolve as well to sustain higher throughput to the devices. Limitations in the systems software stack can however prevent applications from realizing the full potential of hardware improvements; we believe changes are also warranted in the mobile software stack to complement the hardware.

3.2 Android Measurement

Since setting up smartphones for systems analysis and development is non-trivial, we describe our process here in detail; we believe this setup can be useful for someone conducting storage research on Android devices.

Mobile Device Setup

In this study we present results for experiments on the Google Nexus One phone [11]. We also performed the same or a subset of experiments on the HTC Desire [12], LG G2X [14], and HTC EVO [13]; the results were similar and are omitted to save space.

The Nexus One is a GSM phone with a 1 GHz Qualcomm QSD8250 Snapdragon processor, 512 MB RAM, and 512 MB internal flash storage; the phone is running Android Gingerbread 2.3.4, the CyanogenMod 7.1.0 firmware [9] or the Android Open Source Project (AOSP) [3] distribution (as needed), and a Linux kernel 2.6.35.7 modified to provide resource usage information. We present a brief description of the generic OS customizations, which are fairly typical, and then explain the storage-specific customization later in this section.

In order to prepare the phones for our experiments, we setup the Android Debug Bridge (ADB) [1] on a Linux machine running Ubuntu 10.10. ADB is a command-line tool provided as part of Android developer platform tools that lets a host computer communicate with an Android device; the target device needs to be connected to the host via USB (in the USB debugging mode) or via TCP/IP. We subsequently *root* the device with unrevoked3 [18] to flash a custom recovery image (ClockworkMod [6]).

For our experiments we needed to bypass some of the constraints of the stock firmware; in particular, we needed support for *reverse tethering* the mobile device via USB, the ability to custom partition the storage, and access to a wider range of system tools and Linux utilities for development. For example, BusyBox [5] is a software application that provides many of the standard Linux tools within a single executable, ideal for an embedded device. CyanogenMod [9] is a custom firmware that provides these capabilities and is supported on a variety of smartphones. The Android Open Source Project (AOSP) [3] distribution provides capabilities similar to CyanogenMod but is supported only on a handful of Google-smartphones, including the Google Nexus One.

We used the CyanogenMod distribution for all experiments on non-Nexus phones, and for experiments that require comparison between a non-Nexus and the Nexus One phone. All Google Nexus One results presented in this paper exclusively use AOSP; we equipped both CyanogenMod and AOSP distributions with our measurement-centric customizations.

An important requirement, specific to our storage experiments, is to be able to compare and contrast application performance on different storage devices. Some of these applications heavily use the internal non-removable storage. In order to observe and measure all I/O activity, we change Android's `init` process to mount the different internal partitions on the external storage. Our approach is similar to the one taken by Data2SD [17]; in addition, we were able to also migrate to the SD card the `/system` and `/cache` partitions.

In order to adhere to Android's boot-time compatibility tests, we provided a 256 MB FAT32 partition at the beginning of the SD card, mounted as `/sdcard`. The `/system`, `/cache`, and `/data` partitions were formatted as Ext3; at the time we conducted our experiments, YAFFS2 and Ext3 were the pre-installed file systems on our test phones. We performed a preliminary comparison between Ext3 and Ext4 since Android announced the switch to Ext4 [106], but found the performance differences to be minor; a detailed comparison across several file systems can provide more useful data in the future.

Note that this setup is not normally used by end-users but allows us to run what-if scenarios with storage devices of different performance characteristics; the internal flash represents only a single data point in this set.

As part of our experiments, we want to understand the impact of storage on application performance under current WiFi networks, as well as under faster network connectivity (likely to be available in the future). For WiFi, we set up a dedicated wireless access point (IEEE 802.11 b/g) on a Dell laptop having 2GB RAM and an Intel Core2 processor. Since we do not have a faster wireless network on the phone, we emulate one by reverse tethering [19] it over the miniUSB cable connection with the same laptop (allowing the device to access the Internet connection of the host); Table 1 shows the measured performance of our WiFi and USB RT link using *iperf* [51].

Table 1: Network Performance: Transfer rates for WiFi and USB reverse tether link with iperf (MB/s).

Network Connection	Receive Rate	Transmit Rate
USB	8.04	7.14
WiFi	1.10	0.53

To minimize variability due to network connections and dynamic content, we setup a local web server running Apache on the laptop. The web server downloads the web pages that are to be visited during an experiment and caches them in memory; where available, we download the *mobile friendly* version of a web site.

We conducted all experiments on the internal non-removable flash storage and eight removable microSDHC cards, two each from the different SD speed classes [16]. Table 2 lists the SD cards along with their specifications and a baseline performance

Table 2: Raw device performance and cost: Measurements on Desktop with card reader (left) and on actual phone (right). “Sq” is sequential and “Rn” is random performance.

SD Card (16 GB)	Speed Class	Cost US$	Performance on desktop (MB/s)				Performance on phone (MB/s)			
			Sq W	Sq R	Rn W	Rn R	Sq W	Sq R	Rn W	Rn R
Transcend	2	26	4.16	18.03	1.18	2.57	4.35	13.52	1.38	2.92
RiData	2	27	**7.93**	16.29	**0.02**	2.15	5.86	11.51	0.03	2.76
Sandisk	4	23	5.48	12.94	0.68	1.06	4.93	8.44	0.67	0.73
Kingston	4	25	**4.92**	16.93	**0.01**	1.68	4.56	9.84	0.01	1.94
Wintec	6	25	15.05	16.34	0.01	3.15	9.91	13.38	0.01	3.82
A-Data	6	30	10.78	17.77	0.01	2.97	8.93	13.49	0.01	3.64
Patriot	10	29	10.54	17.67	0.01	2.96	8.83	13.38	0.01	3.72
PNY	10	29	15.31	17.90	0.01	3.56	10.28	14.02	0.01	3.95

measurement done on a Transcend TS-RDP8K card reader[2] using the CrystalDiskMark benchmark V3.0.1 [8] (shown on the left side). The total amount of data written is 100 MB, random I/O size is 4KB, and we report average performance over 3 runs; observed standard deviation is low and we omit it from the table. Prices shown are as ordered from Amazon.com and its resellers, and Buy.com (to be treated as approximate). We also performed similar benchmarking experiments for the eight cards on the Nexus One phone itself, using our own benchmark program. Testing configuration is as before with 4KB random I/O size and 128 MB of sequential I/O; results in Table 2 (shown on the right side) exhibit a similar trend albeit lower performance than for desktop.

To summarize, read performance of the different cards is not a crucial differentiating factor and much better overall than the write performance. Sequential reads clearly show little or no correlation with the speed class; sequential write performance roughly improves with speed class, but with enough exceptions to not qualify as monotonic. Random read performance is not significantly different across the cards. The most surprising finding is for random writes: most if not all exhibit abysmal performance (0.02 MB/s or less!); even when sequential write performance quadruples (e.g., Transcend versus Wintec), random writes perform several orders of magnitude worse.

In terms of overall write performance including random and sequential, Kingston

---

[2] Note that internal flash could not be measured this way.

consistently performs the worst and tends to considerably skew the results (as shown in Figure 4 and Figure 5); we try not to rely on Kingston results alone when making a claim about storage performance. In practice, we find that application performance varies even with the other better cards. Transcend performs the *best* for random writes, by as much as a factor of 100 compared to many cards, but performs the *worst* for sequential writes; Sandisk shows a similar trend. A-Data, Patriot, Wintec, and PNY perform poorly for random, but give very good sequential performance. Kingston and RiData suffer on both counts as they not only have poor random write performance, but also mediocre sequential write performance (shown in bold in Table 2); application-level measurements reflect the consequences of the poor micro-benchmark results.

3.2.1 Measurement Software

We first explain our measurement environment and the changes introduced to collect performance statistics: (1) We made small changes to the microSD card driver to allow us to check "busyness" of the storage device by polling the status of the `/proc/storage_usage` file. (2) We wrote a background monitoring tool (*Monitor*) to periodically read the proc file system and store summary information to a log file; the log file is written to the internal `/cache` partition to avoid influencing the SD card performance. CPU, memory, storage, and network utilization information is obtained from `/proc/stat`, `/proc/meminfo`, `/proc/storage_usage` (busyness) and `/proc/diskstats`, and `/proc/net/dev` respectively. (3) We use `blktrace` [4] to collect block-level traces for device I/O.

In order to ascertain the overheads of our instrumentation, we conducted experiments with and without the measurement environment; we found that our changes introduce an overhead of less than 2% in total runtime.

Since many popular mobile applications are interactive, we needed a technique to execute these applications in a representative and reproducible manner; for this purpose we used the MonkeyRunner [15] tool to automate the execution of interactive applications. Our MonkeyRunner setup consists of a number of small programs put together to facilitate benchmarking with the necessary application; we illustrate the methodology next.

First, we start the Monitor tool to collect resource utilization information and note its PID. Second, we start the application under test using MonkeyRunner which defines "button actions" to emulate pressing of various keys on the device's touch screen, for example, browsing forward and backward, zooming in and out with the touch screen pinch, and clicking on screen to change display options. Third, while the various button actions are being performed, CPU usage is tracked in order to automatically determine the end of an interactive action. A class function `UntilIdle()` that we wrote is called from the MonkeyRunner script to detect the execution status of an app; it determines idle status using a specified *low CPU threshold* and the minimum time the app needs to stay below the threshold to qualify as idle. Fourth, once the sequence of actions is completed, we perform necessary cleanup actions and return to the default home screen. Fifth, the Monitor tool is stopped and the resource usage data is dumped to the host computer. Similar scripts are used to reset the phone to a known state in order to repeat the experiment (to compute mean and deviation).

3.2.2 Application Benchmarks

We now describe the Android apps that we use to assess the impact of storage on application performance; we automate a variety of popular and frequently used mobile apps to serve as benchmarks.

WebBench: is a custom benchmark program we wrote to measure web browsing performance in a non-interactive manner; it is based on the standard WebView Java Class provided by Android. WebBench visits a pre-configured set of web sites one

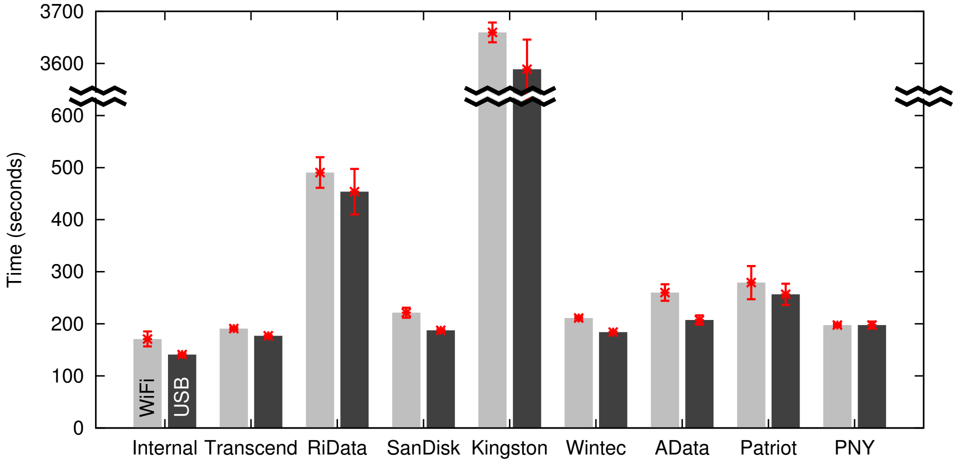


Figure 4: Runtimes for WebBench on Google Nexus One: Runtime for WebBench for SD cards and internal flash; each bar represents average over three trials with standard deviation; lighter bar is over WiFi, darker one for USB RT.

after the other and reports the total elapsed time for loading the web pages. In order to accurately measure the completion time, we made use of the public method of WebView class named `onProgressChanged()`; when a web page is fully loaded, WebBench starts loading the next web page on the list. We ran WebBench to visit the top 50 web sites according to a recent ranking [7].

AppInstall: installs a set of top 10 Android apps on Google Android Market (listed in Table 3 on the left), successively, using the

`adb install` command. App installation is an important and frequently performed activity on smartphones; each application on the phone once installed is typically updated several times during subsequent usage. In addition, often times a user needs to perform the install "on the go" based on location or situational requirements; for example, installing the IKEA app while shopping for furniture, or the GasBuddy app, when looking to refuel.

AppLaunch: launches a set of 10 Android apps using MonkeyRunner listed in Table 3 on the right; the apps are chosen to cover a variety of usage scenarios: games (AngryBird and SnowBoard) take relatively longer to load, read traffic to storage

Table 3: Apps for Install and Launch from Android Market Install: top Apps in Aug 2011, total size 55.58 MB, average size 5.56 MB; Launch: 10 apps launched individually.

App Name (Install)	Size (MB)	App Name (Launch)	Size (MB)
YouTube	1.95	AngryBird	18.65
Google Maps	6.65	SnowBoard	23.54
Facebook	2.96	Weather	2.60
Pandora	1.22	Imdb	1.38
Google Sky Map	2.16	Books	1.05
Angry Birds	18.65	Gallery	0.58
Music Download	0.70	Gmail	2.14
Angry Birds Rio	17.44	GasBuddy	1.88
Words With Friends	3.75	Twitter	1.36
Advanced Task Killer	0.10	YouTube	0.80

dominates. Weather and GasBuddy apps download and show real-time information from remote servers, i.e., network traffic is high. Gmail and Twitter apps download and store data to local database, i.e., both network and storage traffic is high. Books and gallery apps scan the local storage and display the list of contents, i.e., read to storage dominates. Imdb has no storage or network traffic due to web cache hits, while YouTube launch is network intensive.

Facebook: uses the Facebook for Android application; each run constitutes the following steps: (a) sign into the author's Facebook account (b) load the news feed displayed initially on the phone screen (c) "drag" the screen five times to load more feed data (d) sign out.

Google Maps: uses the Google Maps for Android application; each run constitutes the following steps: (a) open the Maps application (b) enter origin and destination addresses, and get directions (c) zoom into the map nine times successively (d) switch from "map" mode to "satellite " mode (e) close application.

Email: uses the native email app in Android; each run constitutes the following steps: (a) open the app, (b) input account information, (c) wait until a list of received emails

Table 4: I/O Activity Breakdown: Aggregate sequential. and random, writes and reads during benchmark; note moderate to high rand:seq write ratios for WebBench, Email, Maps, Facebook, and low for AppInstall. Zero value means no activity during run.

Activity	Write (MB)		Read (MB)	
	Sq	Rn	Sq	Rn
WebBench	41.3	32.2	6.8	0.5
AppInstall	123.1	5.6	0.7	0.1
Email	1.0	2.2	1.1	0.1
Maps	0.2	0.3	0	0
Facebook	2.0	3.1	0	0
RLBench	25.6	16.8	0	0
Pulse	2.6	1.0	0	0

appears, and (d) close the application.

RLBench [88]: a synthetic benchmark app that generates a pre-defined number of various SQL queries to test SQLite performance on Android.

Pulse News [23]: a popular reader app that fetches news articles from a number of websites and stores them locally. Our benchmark consists of the following steps: (a) open Pulse app, (b) wait until news fetching process completes, and (c) close the app.

Background: another popular usage scenario is concurrent execution of two or more applications (Android and iOS are both multi-threaded); several apps run in the background to periodically "sync" data with a remote service or to provide proactive notifications. Our benchmark consists of the following set of apps in auto sync mode: Twitter, books, contacts, Gmail, Picasa, and calendar, and a set of active widgets: Pulse, news, weather, YouTube, calendar, Facebook, Market, and Twitter.

For many of the above benchmarks (e.g., Facebook, Email, Pulse, Background), the actual contents and amount of data can vary across runs; we measure the total amount of data transferred and normalize the results per Megabyte. We also repeat the experiment several times to measure variations; for multiple iterations, the local

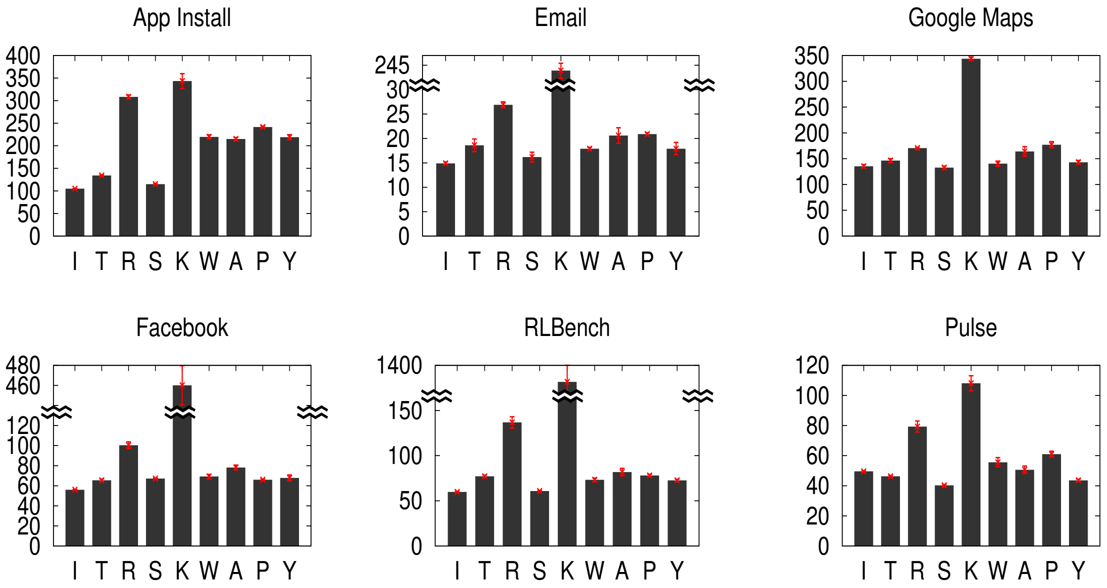


Figure 5: Runtimes for popular applications: Similar to Figure 4 but for several other apps on WiFi only; I: Internal, T: Transcend, R: RiData, S: Sandisk, K: Kingston, W: Wintec, A: AData, P: Patriot, Y: PNY. Some graphs are plotted with a discontinuous y-axis to preserve clarity of the figure in presence of outliers like Kingston.

application cache is deleted following each run.

3.3 Summary

Contrary to conventional wisdom, we find evidence that storage is a significant contributor to application performance on mobile devices; our experiments provide insight into the Android storage stack and reveal its correlation with application performance. Surprisingly, we find that even for an interactive application such as web browsing, storage can affect the performance in non-trivial ways; for I/O intensive applications, the effects can get much more pronounced. With the advent of faster networks and I/O interconnects on the one hand, and a more diverse, powerful set of mobile apps on the other, the performance required from storage is going to increase in the future. We believe the storage system on mobile devices needs a fresh look and we have taken the first steps in this direction.

CHAPTER IV

FLASHLITE: SELECTIVE LOGGING

In this chapter, we explain our selective logging solution named FlashLite [64] [1] as a case study of ISM. The main idea of FlashLite solution is selectively applying the logging solution only to the chosen applications, and it supports the ISM approach.

4.1 Introduction

Peer-to-peer (P2P) file sharing programs, such as *BitTorrent* [34] and *eMule* [56], have become popular today. A significant portion of the Internet traffic is generated by P2P programs now [59], and it is easy and efficient to download a huge Linux distribution with P2P method.

The possible reasons for the success of P2P file sharing are scalability and robustness. It downloads a file from multiple peers simultaneously, and also uploads some parts already downloaded at the same time. In a traditional downloading method, more clients implies longer downloading time. In contrast, P2P file downloading program works more efficiently when there are a number of clients trying to download the same file because they help one another. Moreover, P2P protocol usually provides robust download because it is less dependent on a single server.

This distributed downloading mechanism, called *swarming*, causes a special file write pattern in P2P file sharing programs. Because small parts are concurrently downloaded from many peers and written to a destination file, its write pattern tends to be random, and the degree of the randomness is highly dependent on the size of the downloading chunk and the number of peers that are connected to get the file

---

[1] This work was presented at ICDCS'09 conference.

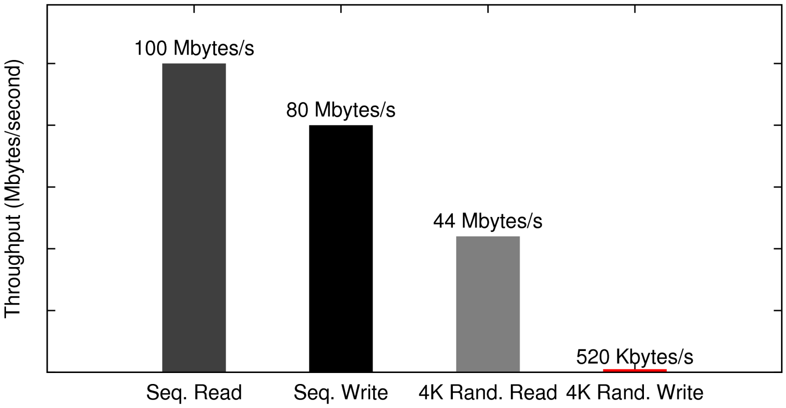


Figure 6: Sequential and 4KB random read/write performance of MTron MSD-SATA3025 SSD[22]

simultaneously.

Recently released Solid-State Drive (SSD) using NAND flash memory is getting popular due to its attractive benefits. It is energy efficient, light-weight, and absolutely silent. In addition, delay-free random reads of SSD enable a system to boot fast.

However, SSD suffers from random writes in general. Figure 6 shows the performance of MTron MSD-SATA3025 SSD [22]. The 4KB sized random write speed is only 520KB/second while the sequential write performance is 80MB/second. Random write performance is only 0.6 % of sequential performance. While the level of performance difference is specific to each SSD, they show poor performance for random writes in general [27, 38, 72].

Random writes also shorten the lifetime of SSDs. When a write request takes a long time to complete in SSD, it means that the request causes many physical operations on flash memory such as page writes and block erasures. Due to the nature of the technology, NAND flash memory can incur only a finite number of erasures for a given physical block. Therefore, increased erase operations due to random writes shortens the lifetime of an SSD. In other words, random writes make a flash storage wear out much faster than normal writes.

While random writes are very slow as well on SSDs, the durability issue is a more serious problem to solve because the performance bottleneck of P2P file sharing program is usually the network rather than storage. For example, our experiments show that P2P download could make SSD wear out over hundred times faster than normal FTP download. The reality is of course that SSDs are becoming popular and viable to use in place of hard disk on notebook and tablet PCs. The user community on such gadgets will necessarily use P2P file sharing. Therefore, solving the random write problem on SSD is critical to the lifetime of such gadgets.

In this study, we analyze the write patterns of P2P file sharing programs, and explain the basics of flash storage to show how harmful P2P program could be for SSD. We also propose, a light weight library called *FlashLite* for P2P file sharing programs. *FlashLite* changes random writes of an application to sequential writes with logging technique similar to log-structured file systems [90].

For evaluation, we have implemented *FlashLite* and applied it to a well known P2P file sharing program, *emule 0.49b*. We have collected write traces while downloading a 3.3Gbyte sized *Fedora 9* DVD ISO image using this *modified eMule*, and we have verified that the writes are effectively changed to be sequential. We have also performed trace-driven simulation to find out the number of block erasures inside an SSD. The results show that *FlashLite* effectively eliminates about 94% of the physical erase operations compared to the original for the test of *Fedora 9* image downloading.

This study makes three main contributions. First, we show that the workload of P2P file sharing program is very unique and could be harmful for flash storages. The second contribution, perhaps the most important, is the new library *FlashLite* to deal with the random write problem of *P2P swarming*. Thirdly, we propose a novel method for evaluating the lifetime of an SSD, using a combination of trace-driven simulation and emulation of the SSD hardware.

4.2 Write Patterns of P2P Downloading

We collected disk accesses on Windows XP with *DiskMon*[91] while downloading a large enough test file with various P2P file sharing programs.

Our test machine[2] has 8Gbyte sized SLC SSD for `C` drive and 30Gbyte sized MLC SSD for `D` drive. We use an empty `D` drive while Windows XP was installed on `C` drive to filter out unrelated disk accesses to our test. Before every download, we format the `D` drive with FAT32 to get rid of disk aging effect. We use a 3.3Gbyte sized *Fedora 9* i386 DVD ISO image as a test file for downloads because the file is large and popular enough for our test. Popular file can be downloaded fast by P2P file sharing program.

Figure 7 presents the collected write traces for eight downloads: One is from *ftp (Windows XP)*, four are from BitTorrent clients, and the remaining three are produced by eDonkey2000 clients. In the graph, Y-axis represents the logical sector number of the write requests and X-axis represents write sequence (i.e., temporal order of write requests).

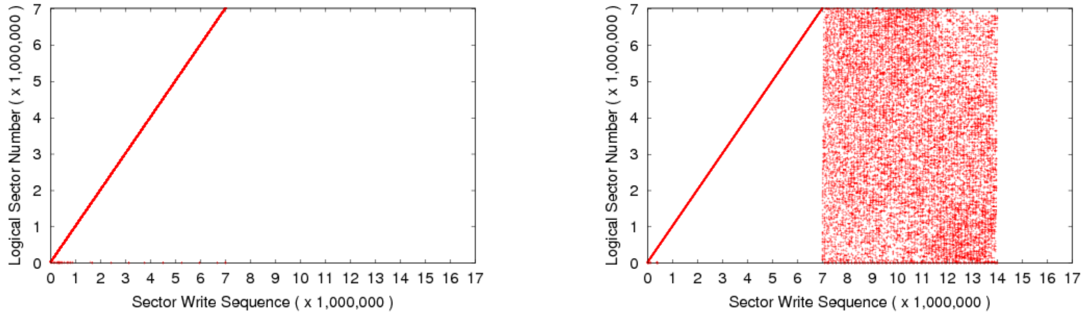
Figure 7 (a) presents perfectly sequential write pattern by *ftp*. The file content is downloaded and written from its beginning to the end in a fully sequential manner.

Unfortunately, the results are quite different when we use P2P file downloading programs. The remaining graphs in Figure 7 show these results. Figure 7 (b) shows the write traces of *BitTorrent*. The sequential writes at the beginning are due to the creation of a destination file. *BitTorrent* first creates an empty destination file with final download size, and then overwrites the blocks thus reserved almost randomly. *Vuze* (Figure 7 (c)) and *μTorrent* (Figure 7 (d)) are also BitTorrent network clients, and the write patterns are almost the same as *BitTorrent*.

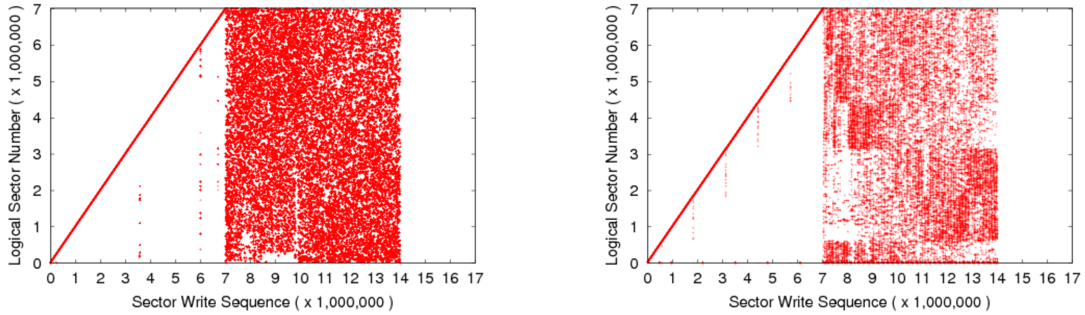
Figure 7 (e) of *BitTornado*, another BitTorrent client, presents a very unique write pattern. Instead of creating a destination file with the final download size at

---

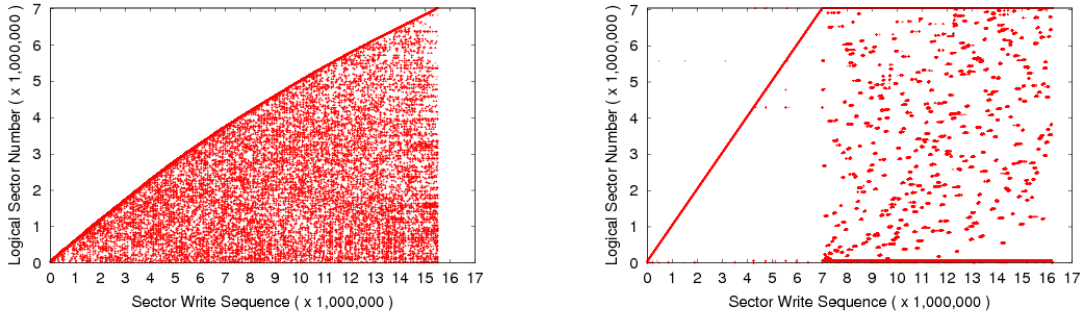
[2] Asus EeePC 1000 Netbook



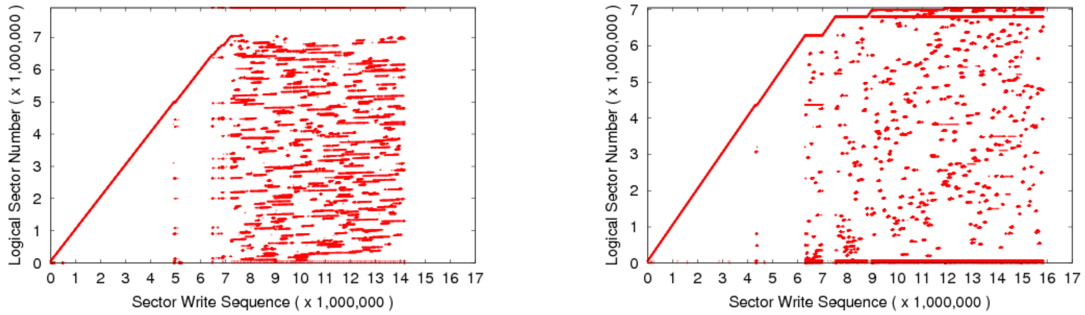
(a) ftp (b) BitTorrent 6.1.1



(c) Vuze 3.1.1.0 (d) μTorrent 1.8.1



(e) BitTornado 0.3.17 (f) NeoMule 4.50



(g) aMule 2.2.2 (h) eMule 0.49b

Figure 7: Write Traces of P2P file sharing programs: *Downloading 3.3Gbyte sized Fedora 9 Image*

the beginning like other BitTorrent clients, it increases the file size gradually. This means that *BitTornado* gradually enlarges the size of the downloading window.

Three eDonkey2000 (ED2K) network clients show almost the same write tendencies as seen in Figures 7 (f), (g), and (h). However, the writes of ED2K clients seem to be less scattered than BitTorrent clients. It is possibly because BitTorrent clients are more aggressive than ED2K clients, and ED2K network is less popular than BitTorrent network at the present time.

Even though there are some differences in the write patterns among P2P file sharing programs, all the tested P2P file sharing programs show extensive random write patterns, thus establishing the premise of our work.

4.3 FlashLite

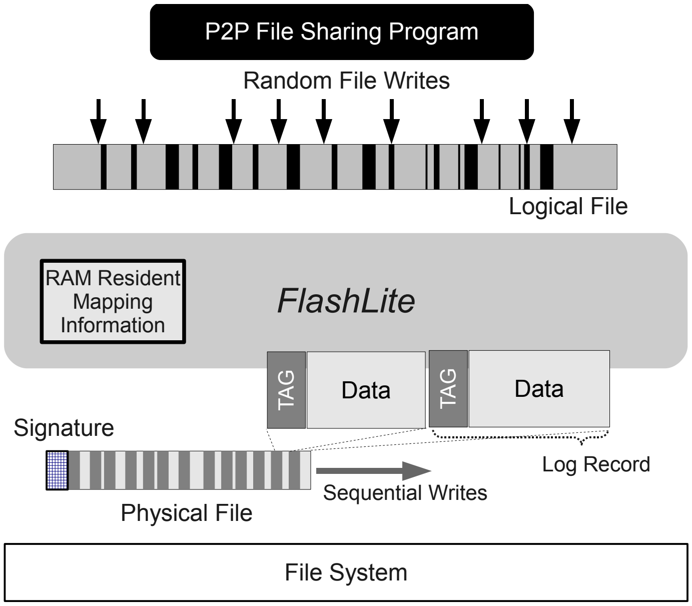


Figure 8: Concept of FlashLite

The basic idea of *FlashLite* is almost the same as log-structured file system [90]. Log-structured file system was originally proposed to avoid the small write problem in UNIX development environments. Such small writes translate to creating *log records* that are written sequentially to the same large log file. In a similar manner, *FlashLite* creates a log file and the incoming (random) writes are written as *log records* sequentially to the same physical log file. Each *log record* in *FlashLite* consists of a *tag* and *data*; the *tag* contains the information about the position of the *data* in the file that is being downloaded. Figure 8 captures the concept of *FlashLite*.

There are two important data structures in *FlashLite*. The first data structure contains information about the *tag* which describes the *log record*, and has three fields. The first field indicates the type of *log record*, and the remaining two fields are interpreted differently based on the type. For a file write operation, these fields give the logical file offset and size for the data being written. For `SetFileLength()` operation, which is called for creating a new file, only one of these two fields is meaningful and that field gives the size of the new file being created.

The second data structure is used for RAM resident mapping information. *FlashLite* writes data sequentially regardless of the logical offset. Therefore, we need to maintain a logical to physical mapping in memory for reading the file that has just been written. This is a doubly-linked data structure (see Figure 9) that contains three fields: logical offset, physical offset, and length of data.

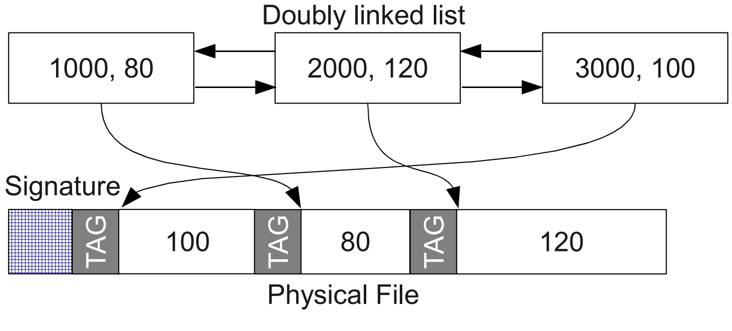


Figure 9: Linked list after three writes: 100 bytes at offset 3000, 80 bytes at offset 1000, and 120 bytes at offset 2000

File Writing

For a write request, a *tag* structure is filled with proper information (logical offset and size) and written with *data* to the physical file sequentially. *FlashLite* maintains RAM resident mapping information for logical to physical offset translation, and it is updated for the data that is being written. A new node structure is allocated, filled with logical offset, data size, and the actual physical file offset, and inserted into the doubly linked list. Currently, *FlashLite* uses a doubly linked list for simplicity; it may be changed to a more sophisticated data structure such as radix tree for better performance in the future. Figure 9 shows an example of a linked list generated after three consecutive write requests.

File Reading

To read data, we need to translate logical file offset to physical offset because data is written sequentially regardless of its logical offset in *FlashLite*. To minimize CPU overhead, *FlashLite* remembers the last accessed node structure in the mapping list, and searches the list from that point. If a node having the required data is found, the data is read using the physical offset in the node structure. The search may fail because a user may attempt to read data that has not been written yet. In that case, *FlashLite* fills the read buffer with zero. One read request on *FlashLite* can cause multiple discrete reads of the log file since there may be multiple log records on the log file that contain all the requested data.

File Opening

FlashLite writes a signature at the beginning of a log file to distinguish it from a normal file. When a log file is re-opened, RAM resident mapping information has to be reconstructed. All *tags* in a log file are read sequentially, and the doubly linked list is rebuilt with the information in *tags*. This process is time consuming because the whole file should be read. Fortunately, *FlashLite* does this process only for the certain

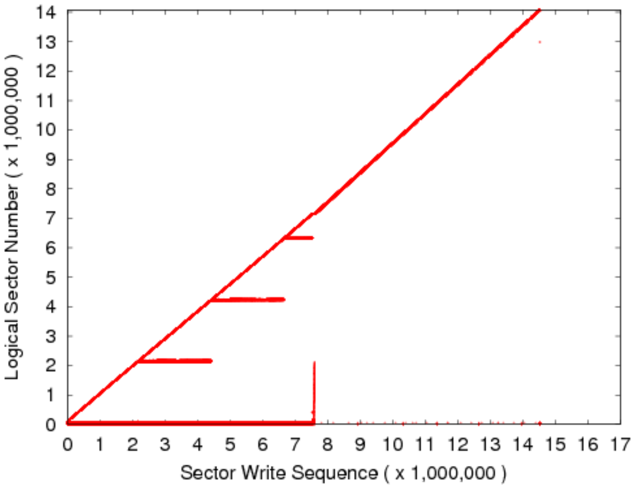


Figure 10: Write Traces of eMule with FlashLite

downloading files of P2P file sharing program while a log-structured file system has to do that for the whole storage.

File Closing

When a file is closed, *FlashLite* destroys the RAM resident mapping information for the file.

File Rearranging

When we download a file with a P2P file sharing program using *FlashLite*, the file is written as a log file as we just described. Further, this file can be read only by using the file read operation provided by *FlashLite*. However, *FlashLite* provides a simple operation as an API call to convert this log-structured file into a normal file so that normal file operations can be used by other programs that simply want to use the downloaded file. The API call, `RearrangeTo()` reads the log file with *FlashLite* and writes the destination file as a normal file from beginning to end.

4.4 Evaluation

Our evaluation is set out to serve two purposes: 1. To verify that *FlashLite* does result in changing the write pattern of an application to sequential writes from random writes. 2. To verify that *FlashLite* does reduce the erase count considerably compared to the original P2P downloading program.

4.4.1 Write Pattern Study with FlashLite

We collected disk accesses while downloading a test file with *modified eMule* to verify the write pattern, and Figure 10 shows the write traces. Compared to the write pattern of *the original eMule* (Figure 7 (h)), it can be seen that the write pattern is effectively changed to be sequential.

Figure 10 shows that the *modified eMule* has almost doubled the number of sector accesses (Y-axis) compared to the other write traces of P2P file sharing programs (Figure 7). This is because we have to make a call to `RearrangeTo()` after the file download by the P2P program is complete.

Referring to Figure 10, the first half of the writes are generated due to the log writes of *FlashLite* during the file download. During this phase, the horizontal lines in the graph are from the Microsoft FAT file system updates and some other meta files that the application generates on top of the temporal write sequences of the P2P file downloading. For example, *eMule* updates some information about downloading to a *.met* file, and also writes a statistics file frequently. The second half of the sector writes (starting roughly from sector write numbered 8 on the x-axis) is perfectly sequential (no more horizontal lines) and represents the work of the `RearrangeTo()` API call after the download is complete.

Comparing the graphs in Figure 7 with Figure 10, we can see that both the original P2P file downloading programs and the modified *eMule* with *FlashLite* write roughly the same number of sectors (determined by the maximum sector write sequence number on the x-axis). Since *FlashLite* does not create a dummy file with its final download size, the total number of writes including the final rearranging step for modified eMule is similar to that of P2P file sharing programs, except for a small increase for tag writing.

This write pattern study confirms that *FlashLite* effectively converts the random writes of eMule to sequential writes.

4.4.2 Erase Count with FlashLite

The lifetime of SSD can be measured indirectly with erase counts of physical blocks in SSD. However, there is no known way to find out actual erase counts of physical blocks from real SSD. As a solution, we have used a trace-driven simulation method.

Firstly, we developed an emulator for our target SSD. We had to guess the internal FTL algorithm of the SSD for its emulation. Even though it was not possible to find out the accurate FTL algorithm, we could get fair enough model for our emulation by some heuristic write tests. Secondly, we collected write traces on a real SSD while downloading the same test file with various P2P file sharing programs including *the original eMule* and *modified eMule* with *FlashLite*. Finally, we ran the traces on our SSD emulator and were able to get the erase counts from our emulator.

The simulation results for erase counts are shown in Figure 11. The Y-axis represents the total number of erase operations done during replaying the collected write traces, i.e., the sum of all erase counts for all the blocks as reported by the SSD emulator.

Due to the nondeterministic nature of P2P network, we repeated our test five times, and the figure shows the average results with maximum and minimum. The simulated average erase counts of *eMule*, 217,610, is significantly reduced to 13,254 by *FlashLite*. It is only 6.1% compared to the original *eMule*.

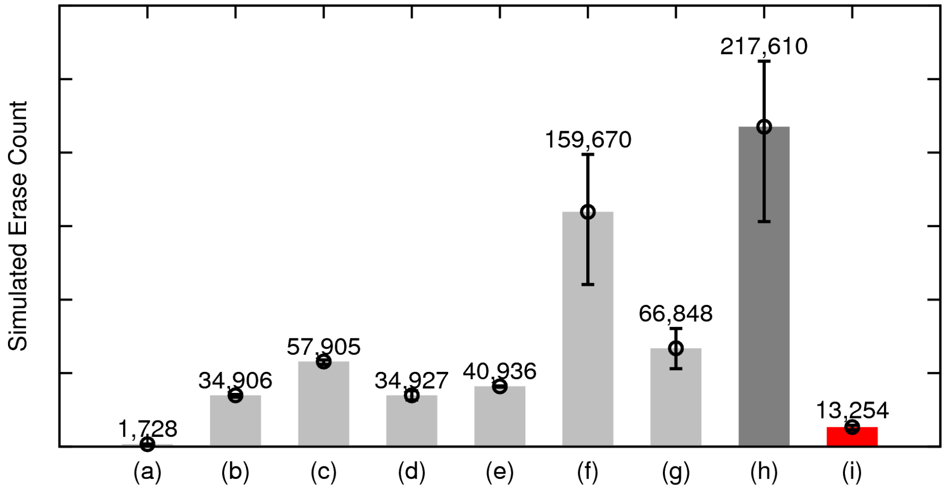


Figure 11: Simulated Erase Counts: (a) ftp, (b) BitTorrent, (c) Vuze, (d) μTorrent, (e) BitTornado, (f) NeoMule, (g) aMule, (h) eMule, (i) eMule with FlashLite

From Figure 11 note that BitTorrent clients show much smaller erase counts than ED2K clients, despite the random write patterns shown by the traces earlier (see Figure 7). This was a surprising result but can be explained due to a couple of reasons. The first reason is that the downloading chunk size in BitTorrent is 256Kbytes which is much larger than that used by ED2K.

The second reason is that BitTorrent writes only a single downloading file. On the other hand, eMule writes several files(both downloading file and meta files) during the downloading process very frequently.

4.5 Summary

SSD technology is becoming a viable replacement for hard disk at least in the end user market (laptops, tablet PC, etc.). P2P file downloading is a popular application for the community of users that use such devices. P2P file downloading employs swarming to efficiently download different parts of a large file from multiple peers. This in turn results in generating random writes to the storage device on the target platform, which is particularly detrimental to the lifetime of SSD due to the inherent nature of this technology.

We have focused on this problem and made three research contributions in this study. First, we have analyzed the downloading patterns of several popular P2P file sharing programs to show the random write patterns they generate. Second, we have proposed a simple yet powerful user-level technique called *FlashLite*, for converting the random writes to sequential writes. We have implemented this technique as a user-level library for use in applications such as P2P file sharing. We have modified a popular P2P file sharing program called *eMule* to use our library and have shown that such a modification is fairly trivial and straightforward. To evaluate the power of *FlashLite*, through actual file download using the *modified eMule*, we have shown how our technique helps in converting the random writes to sequential writes. Third, we have developed a technique for assessing the lifetime of SSD. For this part, we have faithfully emulated an SSD to account for the erasure counts. Using this emulated SSD and the traces collected from using the original and *modified eMule*, we have shown that *FlashLite* results in reducing the erasure count to 8% of the *original unmodified eMule.*

CHAPTER V

SPATIALCLOCK: FLASH AWARE CACHE REPLACEMENT

In this chapter, we explain our SpatialClock study [66] [1], which is designed for low-end flash storage in mobile platforms. From this work, we can see that low-level information (device performance characteristics) can be useful for high-level OS software.

5.1 Introduction

Over the past decade, mobile computing devices, particularly smartphones, are finding increasing use in our daily lives. According to a recent Gartner report, within the next three years, mobile platforms will surpass the PC as the most common web access device worldwide [44]. By 2013, over 40% of the enhanced phone installed-base will be equipped with advanced browsers [89].

Although mobile systems have become unbelievably popular today, only few studies have been conducted for deep understanding of mobile systems. Mobile systems are not just miniatures of personal computer systems, and thus, the previous research insights from desktop and server systems should not be simply applied to mobile systems without careful reexamination. For example, a recent study reveals that the storage subsystem has a much bigger performance effect on application performance on smartphones than it does on conventional computer systems [62]. Considering the rapid growth of mobile systems, it is the time to move our focus to the mobile system components.

---

[1] This work was presented at SIGMETRICS'12 conference.

CPU power and main memory capacity are increasing very fast; the latest smartphone has a dual core 1.2 GHz processor as well as 1 GB of main memory capacity [94]. On the other hand, the technology used for storage on mobile platforms lags significantly behind that used on regular computers. Flash storage is the norm for smartphones because of the limitations of size, cost, and power consumption. Flash storage exhibits very different performance characteristics relative to the traditional Hard Disk Drive (HDD); plus current operating systems (owing to their legacy of assuming HDD as the primary storage technology) are not engineered to support flash storage adequately. Consequently, flash storage is the Achilles' heel when it comes to performance of mobile platforms [62]. While high-end flash based Solid-State Drives (SSDs) are available and used in regular and enterprise class machines, adoption of such storage for mobile platforms is infeasible for reasons of cost, size, and energy consumption. Therefore, we argue that operating system level software support is critically needed for low-end flash storage to achieve high performance on mobile platforms, and thus, we focus our attention on inexpensive flash storage in this study. Specifically, we are concerned with the buffer cache replacement schemes for mobile platforms using inexpensive flash storage.

OS buffer cache is the focal point for actions regarding how to enhance the performance for OS generated write operations to the storage device. Specifically, we are interested in revisiting the page replacement algorithm used by the OS buffer cache. The primary goal of the OS buffer cache is ensuring a good hit-ratio for the subsystems that sit on top of it. It is well-known that Least Recently Used (LRU) or some variant thereof that preserves temporal locality is a good choice for a page replacement algorithm from the point of ensuring a good hit-ratio. However, such algorithms tend to be agnostic about the performance characteristics of the primary storage backing the buffer cache.

The first step is to take stock of the state-of-the-art in buffer cache management schemes proposed and or used in current operating systems and evaluate their efficacy for flash storages used in smartphones. LRU , Clock [26], Linux2Q [28] are three well-known *flash-agnostic* buffer cache replacement schemes. Clean First Least Recently Used (CFLRU) [83], Least Recently Used Write Sequence Reordering (LRUWSR) [58], Flash based Operation aware Replacement (FOR) [75], Flash-Aware Buffer management (FAB) [55] are previously proposed four *flash-aware* buffer cache replacement schemes. Even though most of these proposed schemes are aiming for general flash storage rather than the inexpensive ones found in mobile platforms, they are a step in the right direction. We would like to understand the performance potential of these schemes (both flash-agnostic and flash-aware ones) for mobile flash storage.

What is the best evaluation strategy for answering this question?

Analytical modeling, simulation, and real implementation are the traditional approaches to performance evaluation of computer systems. Specifically, in the context of OS buffer cache replacement schemes, two techniques have been extensively used: *real implementation* in an operating system, and *trace-driven simulation*. Clearly, real implementation inside an operating system would reveal the true performance potential of any buffer cache replacement scheme. But such an approach is fraught with a number of difficulties and downsides. A real implementation of even a single scheme would require a huge amount of effort. This is because changing the core functionality of an operating system such as the buffer cache replacement scheme is non-trivial since it affects all the subsystems that live on top of it (e.g., VM, file systems). Also, to have a side by side comparison, such an approach would require the implementation of all the competing schemes in the operating system. Besides these difficulties, there are also downsides to this approach. It may be difficult to assess the true performance benefit of the scheme being evaluated due to performance noise from other parts of the operating system. Further, it would be difficult to ask a variety of "what if" questions with a real implementation, without re-engineering the implementation to answer such questions. Perhaps, most importantly the barrier to trying out new ideas will be too high if it has to be implemented first in an operating system to get an estimate of its performance potential. It would stifle creativity.

Trace-driven simulation has been extensively used for buffer cache related studies sometimes together with real implementation [52, 53, 83], and many other times just by itself [32, 45, 54, 55, 57, 69, 75, 77, 103]. Typically, storage access traces are collected first from real applications on an existing system or synthesized from a workload model of applications. These traces are then used as inputs to a buffer cache simulator to gather metrics of interest for performance evaluation. Hit-ratio is the most popular metric, but some studies also measure the I/O operation completion time. The virtues of trace-driven simulation include time to getting useful results compared to real implementation, repeatability of results, isolation of performance benefits from other noises, and the ability to have useful "what if" knobs (additional input parameters) in addition to the traces serving as the workload for the evaluation. However, the main drawback of trace-driven simulation is that the results may not accurately capture all the metrics of interest pertinent to the real system. This is especially true with flash storage due to the complicated and often opaque mapping layer inside such devices. Device manufacturers do not publish such internal details; thus these devices have to necessarily be treated as black boxes. Therefore, any simulator can only make a best effort guess as to what is happening inside the device making the veracity of trace-driven simulation results for flash storage questionable.

In this study, we propose a novel buffer cache evaluation framework, which is a hybrid between trace-driven simulation and real implementation. Basically, our method expands the existing trace-driven simulation by adding one more step with a *real* storage device. It allows us to see the real performance effect of cache replacement schemes without actually implementing the algorithm into an operating system.

We collect *before-cache* storage access traces from a real Android smartphone while running popular applications such as Web browser and YouTube player. By using the traces and our proposed evaluation framework, we evaluate seven state-of-the-art buffer cache replacement schemes, and report very surprising results. The most previously proposed flash-aware schemes are not better (sometimes much worse) than flash-agnostic schemes at least with the smartphone workloads that we have evaluated them with. A careful analysis of the results using our new framework reveals the source of this disconnect between previous studies and our surprising new results, namely, not respecting *spatial adjacency* for write operations to inexpensive flash storage. Armed with this new insight, we propose a new buffer cache replacement scheme called *SpatialClock* for mobile flash storage. By comparing SpatialClock to the state-of-the-art buffer cache replacement schemes using our evaluation framework, we show that SpatialClock delivers superior storage performance on real mobile flash storage while not degrading the cache hit-ratio.

We make the following three contributions through this work. First, we propose a new buffer cache evaluation framework. Second, we collect *before-cache* storage access traces from an Android platform, and make them available for other researchers[2] The third and final contribution is the SpatialClock buffer cache replacement algorithm.

5.2 A novel cache evaluation framework

Flash memory is different from conventional magnetic storage. To overcome the physical limitations of flash storage, every flash storage device includes a Flash Translation Layer (FTL) [49, 60] in addition to the storage elements. FTL is a special software layer emulating sector read and write functionalities of an HDD to allow conventional disk file systems to be used with flash memory without any modifications. FTLs employ a *remapping technique* to use the storage cells more judiciously. When FTL receives a request to overwrite a sector, it redirects the new content to an empty page,

---

[2] https://wiki.cc.gatech.edu/epl/index.php/S-Clock

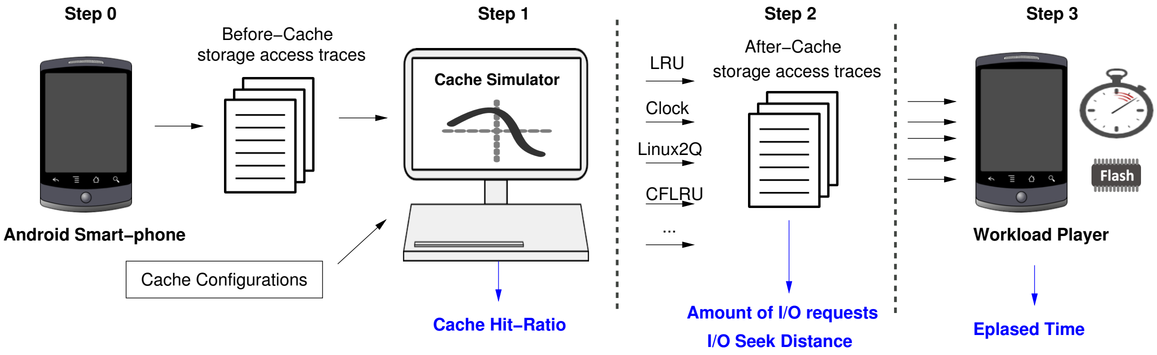


Figure 12: The evaluation framework for buffer cache replacement schemes: *Traditional trace-driven simulations stop with Step 1 or 2. Enhancing the framework with Step 3 allows us to do an accurate evaluation of the performance implications of different cache replacement schemes on real flash storage devices.*

which is already erased, and modifies the mapping table to indicate the new physical page address where the logical sector has been written.

The FTL algorithm employed by a flash storage device is usually opaque, making it difficult to simulate a flash storage device. The performance of a flash storage device critically depends on the algorithms used internally by the FTL. Thus it is pretty much impossible to accurately simulate the internal architecture of a flash device for performance evaluation purposes. While it is always possible to simulate a given FTL algorithm (assuming the details are known) for a specific flash storage, it is impossible to generalize and use it for other flash devices that we want to compare it against, since the performance characteristics of a flash device is so intimately tied to its specific FTL algorithm. Trace-driven simulation may still be good enough to understand the performance potential of a buffer cache management scheme with respect to certain metrics of interest (e.g., hit-ratio). However, we believe, it is not good enough for evaluating all the metrics of interest (e.g., completion time of I/O operations) in understanding the performance potential of a buffer cache scheme on a specific flash storage.

5.2.1 Evaluation Framework

The new buffer cache evaluation framework (Figure 12) is a hybrid between trace-driven simulation and real implementation. First, we give the big picture and then drill down to the details. We collect traces of read/write accesses to the buffer cache from an Android platform running popular Apps. We call this the *before-cache* traces. We use these traces as the workload on a simulator that implements the seven different buffer cache management policies that we alluded to in the previous section. The output of this simulator is two-fold: *hit-ratio* for the chosen cache scheme; and *after-cache* storage access trace for each cache scheme. The latter is the sequence of read/write requests that would be issued to the actual storage since the buffer cache does not have these pages. We have developed a tool called *Workload Player* that takes the *after-cache* trace as its input, sends the read/write requests in the trace to a real storage device, and reports the total elapsed time for performing the read/write operations. Since the times gathered by the Workload Player are the actual elapsed times for the requests, they account for the internal architecture of the flash storage on which the trace is being played. Thus, this hybrid evaluation framework is a faithful reproduction of the combined performance characteristic of the buffer cache algorithm and the real flash storage.

To contrast our hybrid approach to the traditional trace-driven simulator, the latter stops with reporting the observed hit-ratio for a given scheme and generating the actual storage access traces corresponding to the misses. Some studies may take the actual storage access traces to compute the expected completion of I/O requests based on published static read/write/block-erase times of a given flash storage device. By actually playing the *after-cache* traces on a storage device we are able to get the real I/O completion times.

To collect the *before-cache* traces, we have instrumented the Android OS running on a smartphone. We have used Google Nexus-One smartphone with Android version

Table 5: Evaluation results for seven cache replacement schemes: Mixed workload on Patriot 16GB microSDHC card / Nexus One with 4 / 64MB cache size

		LRU	**Clock**	**Linux2Q**	**CFLRU**	**LRUWSR**	**FOR**	**FAB**
4MB	Hit-Ratio	0.7592	0.7583	0.7665	0.7584	0.7580	0.7529	0.7497
	Generated Read Operation Count	13,735	13,917	12,709	14,067	14,125	15,418	14,549
	Generated Write Operation Count	44,600	44,650	44,096	44,413	44,450	44,261	46,131
	Measured Elapsed Time (second)	**234.69**	**234.78**	**261.69**	**241.68**	**229.78**	**236.62**	**292.48**
64MB	Hit-Ratio	0.8863	0.8860	0.8876	0.8860	0.8857	0.8829	0.8860
	Generated Read Operation Count	1,861	2,745	1,605	1,909	1,927	2,523	1,681
	Generated Write Operation Count	26,064	25,833	25,998	26,062	26,060	26,120	26,327
	Measured Elapsed Time (second)	**236.86**	**183.57**	**277.44**	**237.87**	**231.43**	**291.22**	**129.30**

2.3.7 Gingerbread. This in itself is a non-trivial piece of engineering. We modified the *init procedure* of Android for our purpose, and also patched Linux kernel page cache related sources. We developed the buffer cache simulator for the seven schemes discussed to run on a standard desktop Linux platform. The Workload Player runs both on the smartphone and the desktop, and collects performance statistics of playing the *after-cache* traces on flash storage. Our experimental setup includes multiple smartphone storages to run the *after-cache* traces.

5.2.2 A Surprising Result

We first present a representative result for the seven buffer cache management schemes using a microSD card on Google Nexus-One phone. The traces are generated from mobile Apps (such as web browsing and video streaming) running on the Android phone. The result is summarized in Table 5. This result is a good sample of the overall trend we observed for the most test cases (different flash storage devices and different cache sizes), which we elaborate in Section 5.4 (with more details about the workloads used to generate the traces). At this point our goal is to reveal a surprising result that emerged from the evaluation of these seven schemes.

All the schemes show remarkably higher hit-ratios with 64 MB cache size than with 4 MB cache size as expected. However, despite the fact that higher hit ratios implies a reduction in storage activity, the measured *elapsed times* on a real storage device tells a different story. Note for example from Table 5 that the measured *elapsed times* for LRU, Linux2Q, CFLRU, LRUWSR, and FOR are worse than those for Clock and

FAB. This is surprising because hit-ratio is the over-arching metric used to evaluate the effectiveness of buffer cache replacement schemes.

Besides, all four flash-aware schemes fail to reduce the number write operations. Recall that the main focus of flash-aware schemes (except FAB) is to reduce the number of write requests to the storage device. Given this focus, it is interesting that the amount of write operations generated by the flash-aware schemes is not that different from the flash-agnostic ones.

What is interesting from this representative result is that we cannot differentiate the relative merits of these cache replacement strategies using a conventional metric such as hit-ratio. Incidentally, this result also highlights the limitation of a pure trace-driven simulator since hit-ratio and read/write traffic to the storage device are the metrics that can be generated by such a simulator.

Our hybrid approach helps understand the performance potential of the schemes better by letting us measure the elapsed time for playing the *after-cache* traces on a real flash storage device. Surprisingly, three of the four flash-aware schemes show slower performance than the Clock scheme. This is interesting because Clock is not designed for flash storage, and the observed performance differences cannot be explained either by hit-ratios or by the number of generated read/write operations. The surprising result is the fact that the latest flash-aware schemes are not performing as well as one would have expected on a mobile flash storage (Section 5 shows this is true for a variety of flash storage devices). More importantly, this result establishes our first contribution in this study, namely, the power of the hybrid evaluation framework for performance analysis of buffer cache replacement strategies for flash storage

5.2.3 Explaining the surprising result

A more intriguing question is why the three out of the four flash-aware schemes are not performing as well as one would expect. A careful investigation reveals the source

Table 6: Comparison of an HDD (3.5" 7200 RPM HDD) vs. three flash storage devices (Patriot 16GB microSD, two eMMC devices used in Nokia N900 and Google Nexus-S smartphones (KB/sec): *write ordering is important but read ordering is not important on flash storage.*

Storage	Read(KB/sec)		Write(KB/sec)	
	Sorted	Scattered	Sorted	Scattered
HDD	6,498.4	537.6	4,836.6	1,004.0
microSD	4,852.7	4,836.6	545.2	8.3
eMMC-1	5,100.1	4,444.6	470.9	16.1
eMMC-2	3,124.5	2,551.5	566.4	259.1

of this puzzling anomaly. The short answer to this puzzle is *not respecting spatial adjacency* for the write requests that are generated to the flash storage. To fully appreciate this phenomenon, we need to understand some basics of flash storage.

Flash storage is based on semiconductor technology, and hence shows very different performance characteristics when compared to the traditional magnetic disk. A number of studies have reported on the special performance characteristics of flash storage [22, 31, 37]. It is well known that flash storage devices show a relatively low write-throughput for small, scattered (random) requests and a higher throughput for large, sequential write requests. At the same time, they are insensitive to the order of read requests, showing almost unchanging performance for sequential and random read requests. We ourselves have performed simple measurements to identify these differences in performance.

Table 6 compares the measured read and write throughput of an HDD and three flash storage devices. We use four synthetic workloads. All four workloads use the same number of requests (32,768), with request sizes of 4KB (typical page size in most virtual memory systems) within a 1 GB address space (typical process virtual address space). Two of these workloads use random read and write requests, respectively. The remaining two workloads use, respectively, read and write requests that are sorted by the sector number (thus resulting in accessing sequentially ordered sectors on the storage device).

On an HDD, both read and write throughputs are highly influenced by the sequence of the requests because it has seek-delays of the mechanically moving magnetic head. In contrast, the read throughput of flash storage is not much influenced by request ordering because there is no seek delay. For the write requests, flash devices show uniformly lower throughput than the HDD, and their scattered write throughputs are lower than the sorted write throughputs even though there is no moving parts inside flash storage devices. The reason for the lower throughput for scattered writes is due to the way data updating happens internally in a NAND flash memory chip. In other words, not respecting the spatial adjacency for consecutive write requests can result in a huge performance penalty in flash storage. This result suggests that write request ordering can make a huge performance difference, and the disparity between sorted and scattered writes demonstrates that the *ordering* is perhaps more important than the *number* of write requests.

Of the flash-aware schemes evaluated, only FAB respects spatial adjacency while the others (CFLRU, LRUWSR, and FOR) are focused on reducing the total number of write requests to the flash storage. As Table 6 shows, the penalty for ignoring spatial adjacency (i.e., sending scattered write requests to the storage) is huge. This is the reason we see elapsed time differences among the cache replacement schemes even though they all generate roughly the same amount of read/write requests (see Table 5). As is evident from Table 5, FAB has the least elapsed time compared to the other schemes (for 64 MB cache size) since it is the only scheme that explicitly cares about spatial adjacency. However, FAB, owing to its focus on supporting media player workload, is biased too much towards write performance optimization for flash storage to the detriment of overall buffer hit-ratio, which is an important figure of merit for a general-purpose OS buffer cache. This becomes apparent especially at smaller cache sizes and other diverse workloads. For example, it can be seen in

Table 5 that FAB has the worst performance (both elapsed time and hit ratio) with a 4 MB cache size compared to the other schemes. Further, it has some inherent complexities for implementation as a general-purpose OS buffer cache scheme.

5.3 SpatialClock

We have seen that even the flash-aware general-purpose OS buffer cache replacement algorithms proposed thus far do not pay attention to the spatial adjacency (or lack thereof) of the pages being evicted from the OS buffer cache (FAB is an exception but as we noted earlier it does this at the expense of cache hit-ratio and so it is not general-purpose enough for OS buffer cache). Given the discussion in Section 5.2.3, this is a missed opportunity that hurts the performance of flash storage. Therefore, we propose a new algorithm *SpatialClock* that respects the spatial adjacency of the pages being evicted from the OS buffer cache without losing cache hit-ratio remarkably.

There are two important points we want to address head on before we delve into describing SpatialClock:

1. Page replacement algorithms are an age-old topic. However, from the point of view of the storage technology that is currently being used and will be used for the foreseeable future in mobile platforms, we believe it is time to revisit this topic. From our discussion in Section 5.2.3, we can distill a couple of observations regarding flash storage that strengthen our belief: (a) respecting spatial adjacency for writes is very important, and (b) read and write operations are independent of each other due to the nature of the flash technology, in contrast to traditional storage devices such as an HDD (due to the absence of the mechanical head movement).

2. The OS buffer cache is deeply entrenched in the software stack of the operating system. Therefore it is not prudent to overburden this layer with device-specific

> optimizations (such as write reordering), which would require a significant architectural change of the entire OS software stack. What we are proposing in this section is a localized change only to the page replacement algorithm of the OS buffer cache, which does not affect the other functionalities of this layer. Further, as we will see shortly, the proposed algorithm is not storage specific; it merely respects the *logical* spatial adjacency of the pages being evicted from the buffer in addition to temporal locality

The key question is how to design a new page replacement scheme to achieve the two different objectives simultaneously: high cache hit-ratio and sequentially ordered write requests. One commonly used approach is dividing cache memory space into multiple partitions, and applying different cache management policies to the distinct cache partitions [53, 77, 83]. However, this partitioning approach introduces another difficult problem: how to adaptively adjust the partition sizes for various workloads. Further, such a partitioning approach is too major a change to this critical layer of the operating system. Therefore in designing SpatialClock, we take a different approach rather than partitioning the OS buffer.

Before we describe SpatialClock, let us briefly review LRU and Clock algorithms. In LRU, page references are kept in a sorted temporal order by the OS buffer cache. When a page frame is accessed, the frame needs to be moved to the Most Recently Used (MRU) position. The operation may require obtaining a global lock to protect the data structure from concurrent accesses. Because page references are very common, such frequent rearrangement of the data structure is expensive. Further, true LRU is difficult to implement in practice since it requires hardware assistance at individual memory reference granularity to track page frame accesses from the VM subsystem. Nevertheless, true LRU is used in memory system studies as a *standard* to compare other practical page replacement algorithms.

Clock is an approximation to the true LRU algorithm and is often referred to as *second-chance* replacement algorithm. Clock relies on a simple hardware assist common to all processor architectures supporting virtual memory, namely, a per-page *reference bit* (usually part of the page table entry for that page). The reference bit can be set by the hardware and cleared by the software. The hardware sets the associated reference bit when a page frame is accessed unbeknownst to the software (i.e., the operating system). The Clock algorithm keeps the page frames as a circular list in FIFO order of their arrival into the OS buffer cache from the storage device. The victim selection works as follows. The algorithm sweeps the circular list of page frames skipping over the frames whose reference bits are set (clearing the reference bits as it sweeps) and stops at the page frame whose reference bit is not set. This page frame is chosen as the victim for eviction from the OS buffer cache. Clock does not keep the precise reference order like LRU; hence it is simpler and does not require global locks for maintaining its data structure. Despite its impreciseness in maintaining the page reference history, the good news is that the performance of Clock approximates LRU in most cases. Therefore, Clock has been widely used especially for virtual memory systems, which require low overhead lookup of the buffer cache.

In SpatialClock, we follow the basic rules of the Clock algorithm with only one difference. Page frames are arranged by the *logical* sector number of the storage system that contains the page frame. Consequently, page frames are chosen as victims for eviction in the sequential order of the sectors that contain these frames, and thus, frames are implicitly chosen sequentially with respect to the storage device. This results in preserving/respecting spatial locality during victim selection when the chosen frames happen to be dirty as well. Figure 13 shows the victim selection algorithm of SpatialClock.

Figure 14 shows an example of the victim selection in the SpatialClock algorithm. Each row represents a page frame, and the left cell in the row represents the containing sector number for that page frame, the right cell indicates the reference bit value. The

```
/* Victim selection */
pageframe*
spatialclock_choose_victim()
{
  /* sweeping until find a victim */
  while (1)
  {
    /* circular movement */
    if (cur_pointer == NULL)
      cur_pointer =
        avltree_move_to_first();

    victim      = current_pointer;
    cur_pointer =
      avltree_move_to_next(cur_pointer);

    if (victim->referenced == 0) break;
    victim->referenced = 0;
  }
  return victim;
}
```

Figure 13: SpatialClock victim selection algorithm

page frames are pre-arranged in sorted order with respect to the containing sector numbers for the page frames. To choose a victim page, page frames are scanned from the current pointer position to find a page frame, which has a ‘0’ for the reference bit value. In the given example, the sweep stops at page frame whose containing sector number is 80, while clearing the reference bits of the page frames having 40 and 72 as the containing sector numbers, respectively.

Respecting and checking the reference bits gives SpatialClock the advantage of an approximate LRU for victim selection. Arranging the page frames in a spatially adjacent manner gives an opportunity to enforce write ordering for the evicted page frames. Giving more importance to physical adjacency than the recency of access could affect the hit-ratio. However, our evaluation results show that this is not the

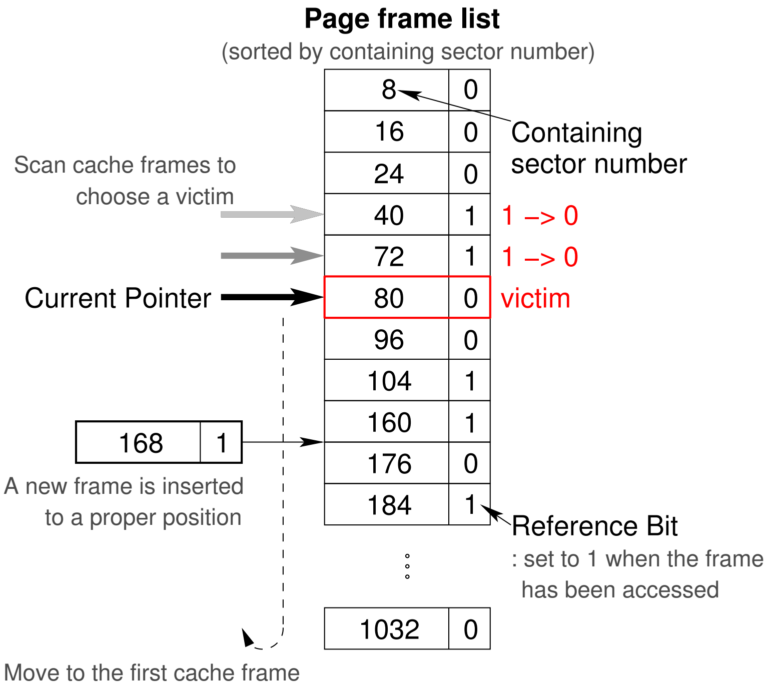

Page frame list
(sorted by containing sector number)
8
0
16
0
24
0
40
1
72
1
80
0
96
0
104
1
160
1
176
0
184
1
1032
0
Containing sector number
Scan cache frames to choose a victim
1 –> 0
1 –> 0
Current Pointer
victim
168
1
A new frame is inserted to a proper position
Reference Bit
: set to 1 when the frame has been accessed
Move to the first cache frame

Figure 14: Victim Selection in SpatialClock: *SpatialClock maintains and scans page frames in the order of the containing sector numbers to generate ordered write requests.*

case at least for the traces we studied. More importantly, we argue that paying attention to the elapsed time for storage access is crucial for achieving good performance on flash storage.

Compared to the original Clock scheme, SpatialClock requires maintaining page frames in a sorted manner, and we use an AVL tree [21] for the purpose. However, the burden is only for a page frame insertion operation, which is a relatively rare operation as long as the hit-ratio is high. The more common reference operation of the OS buffer cache remains exactly the same as in the original Clock algorithm. Besides, the AVL tree can be used for page look up purpose, which is mandatory for buffer cache maintenance.

We have implemented SpatialClock using an AVL tree to bound the page frame insertion time to be *log* N, where N is the number of page frames. We associate logical sector numbers with each frame (obtained from the storage map maintained by the OS buffer cache) to organize the circular list respecting spatial adjacency.

5.4 Evaluation

We focus our evaluation on the following two key points:

- Hit-Ratio Comparison: SpatialClock is designed to produce sequentially ordered write requests, and often disobeys the philosophy of Clock and LRU policies. Will it degrade cache hit-ratio remarkably? We will answer this question by comparing cache hit-ratios with multiple traces and cache sizes.
- Performance effect on flash storage: We will verify the performance effect of SpatialClock on real flash storage devices.

We compare SpatialClock head to head with the seven schemes we introduced already: (1) LRU, (2) Clock, (3) Linux2Q, (4) CFLRU, (5) LRUWSR, (6) FOR, and (7) FAB.

5.4.1 Experimental Setup

We collected *before-cache* storage access traces from a real Android smartphone. Even though many disk access traces are available in the public domain, most of them are *after-cache* traces, and some traces used in previous studies (for example, LIRS [54]) do not separate read and write accesses. More importantly, we want to use the traces that came from a real smartphone while running popular mobile Apps.

We used a Google's Android reference phone, Nexus-One [11] with Android Open Source Project (AOSP) [3] 2.3.7, Gingerbread version. We modified the init procedure of Android to use the partitions on an external microSD card with EXT3 file system instead of the internal NAND flash memory with YAFFS2 file system because it is not possible to collect general block level traces from YAFFS2 file system. We also had to modify the page cache related parts of Linux kernel (version 2.6.35.7) to collect the *before-cache* accesses.

We choose three typical and also popular workloads for our evaluation.

Table 7: Trace Information

	Read Operation		**Write Operation**	
	Count	Amount (MB)	Count	Amount (MB)
W1	23,666	92.4	47,350	185.0
W2	67	0.3	387,701	1,514.5
W3	134,910	527.0	105,796	413.3

- **W1: Web Browsing.** We collected storage access traces while doing web browsing for multiple hours. Web browsing may be the most common activity on today's mobile platforms such as smartphones and Internet tablets. When we visit web pages, web browser downloads web resources like image files into local storage to reduce network traffic. Therefore, while doing web browsing, small files are continually read and written, and storage performance influences user's web browsing experience. The collected amount of traces is smaller than our expectation because Android web browser is directed to the mobile web pages, which are optimized to minimize network and I/O traffic.

- **W2: Video Streaming.** We collected storage access traces while watching various YouTube video clips for multiple hours. When we watch Internet streaming video like YouTube, video data are buffered into local storage to provide stable video watching quality. This is another very popular activity on mobile platforms, and generates very different storage access pattern compared to web browsing. This workload has the highest amount of write traffic among the three workloads studied.

- **W3: Mixed App Workload.** In this workload, we collected storage access traces for several hours while running multiple Apps sometimes together and sometimes separately. Following Apps were used: Facebook, Twitter, Maps, Pandora, Angry Birds (game), Fruit Ninja (game), OfficeSuite, Camera, Internet Browser, YouTube, Gallery, Android Market, etc. We believe this workload is the best one to reflect a realistic usage of the smartphone.

Table 8: Flash Storage Devices

Smartphone	**Type**	**Chip Maker**	**Size**
Nexus One	microSDHC	Patriot(Class 10)	16 GB
N900	eMMC	Samsung	30 GB
Nexus S	eMMC	SanDisk	15 GB

Table 7 shows the number of read and write operations in the collected mobile traces. Note that there are only few read requests in the video streaming trace (W2). This could very well be due to the limitation of our trace collection method since it is not possible to collect in-memory accesses for a memory mapped file without hardware support[3]. Even though the collected traces may not be a perfectly faithful reproduction of the I/O activity in these Apps (since they are missing the accesses to memory mapped files), we note that this situation is unfortunately unavoidable and will happen even if we profile a real operating system. Thus, we believe that the traces are valid and proper for our evaluation. Besides, since all the cache replacement schemes are compared with the same set of traces, the comparison is fair.

Some of the buffer cache replacement schemes require setting some algorithm specific parameters. For Linux2Q, we set the active vs. inactive queue ratio to be 3:1 (this is similar to the setting in the Linux kernel). For CFLRU, the Clean-First window size is set to be 25% of total cache size. For FOR, we use an alpha value of 0.5 as recommended by the authors of the paper, and read and write operation cost as 100us and 800us, respectively. Lastly for FAB, we set the number of pages per block as 64, which is the same as in the author's own evaluation of their scheme.

Two different types of flash storage are popularly used in smartphones today: microSD cards and eMMC devices. Due to space limitation, we choose to present the result of three devices. Table 9 shows the list of the chosen flash devices: one microSD card (with Google Nexus-One phone) and two (slower and faster) eMMC

---

[3] ARM processor in the Nexus-One phone does not provide this functionality.

devices (internal to each of a Nokia N900 and Google Nexus-S phones, respectively).

Workload Player simply receives a trace file, performs the I/O operations specified in the trace file on a real storage device, and reports the elapsed time for performing the operations. We run Workload Player on real smartphones. Google Nexus-One is used to evaluate the microSD card, and a Nokia N900 and a Google Nexus-S are used respectively, to evaluate their internal eMMC devices. The Workload Player is written as a regular C program running on Linux, and the buffer cache is bypassed by using `O_DIRECT` option.

5.4.2 Hit-ratio Comparison

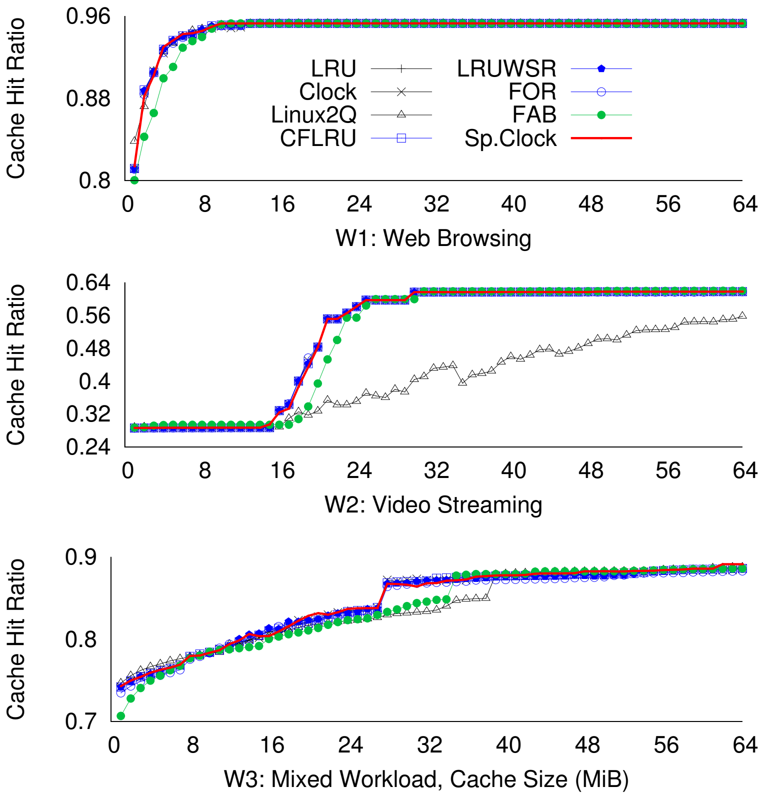


Figure 15: Hit-Ratio Comparison:*SpatialClock shows comparable hit-ratios to other schemes.*

Figure 15 shows simulated buffer cache hit-ratios for the eight cache replacement schemes using the three traces. Except Linux2Q and FAB, other six schemes are not much different from one another in terms of hit-ratio. Linux2Q shows lower hit-ratio than the others for the video streaming workload (middle graph) while showing relatively higher hit-ratios when cache size is small. Meanwhile, FAB shows remarkably lower hit-ratios when the cache size is small. It will be very interesting to analyze the reason for the poor performance of Linux2Q and FAB but it is not the focus of this study. We can clearly verify that SpatialClock and other flash-aware schemes except FAB show comparable (not remarkably low, at least) hit-ratios to non-flash-aware schemes even though they sometimes disobey LRU philosophy for the sake of accommodating the performance quirks of flash storage.

5.4.3 I/O Operation Cost Analysis

Existing flash-aware buffer replacement schemes are mainly focusing on the asymmetric read and write operation costs. Prior cache replacement performance studies have calculated the total cost of the I/O operation by applying a simple mathematical equation using the differential read/write times. We have done a similar calculation. To this end, we count the number of read and write operations for each buffer management scheme, and calculate the total cost by using a simple cost model for a flash chip as is done in the FOR paper [75] (100us and 800us for read and write I/O operations, respectively).

Figure 16 shows the calculated times. Similar to the hit-ratio comparison results, no obvious differences are seen except for the Linux2Q and FAB cases. Based on this calculated result, it would appear that none of the flash-aware algorithms (including SpatialClock) are any better in reducing the total I/O cost compared to the flash-agnostic ones. We show in the next subsection that such a conclusion would be erroneous.

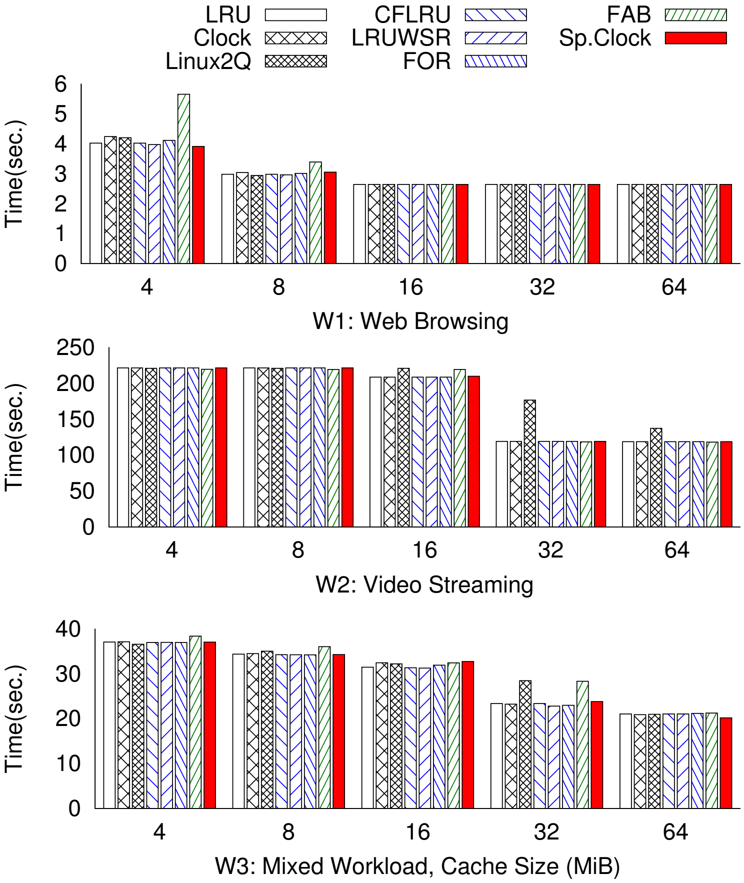


Figure 16: Calculated elapsed time based on the number of read and write operations in after-cache traces: *the results are almost indistinguishable for the different schemes.*

5.4.4 Performance Effect on Real Flash Storages

It is hard to show write ordering effect of buffer cache replacement algorithms without using real flash storage devices. Therefore, we collect *after-cache* traces generated by the cache simulator (Figure 12), and play them on real smartphones.

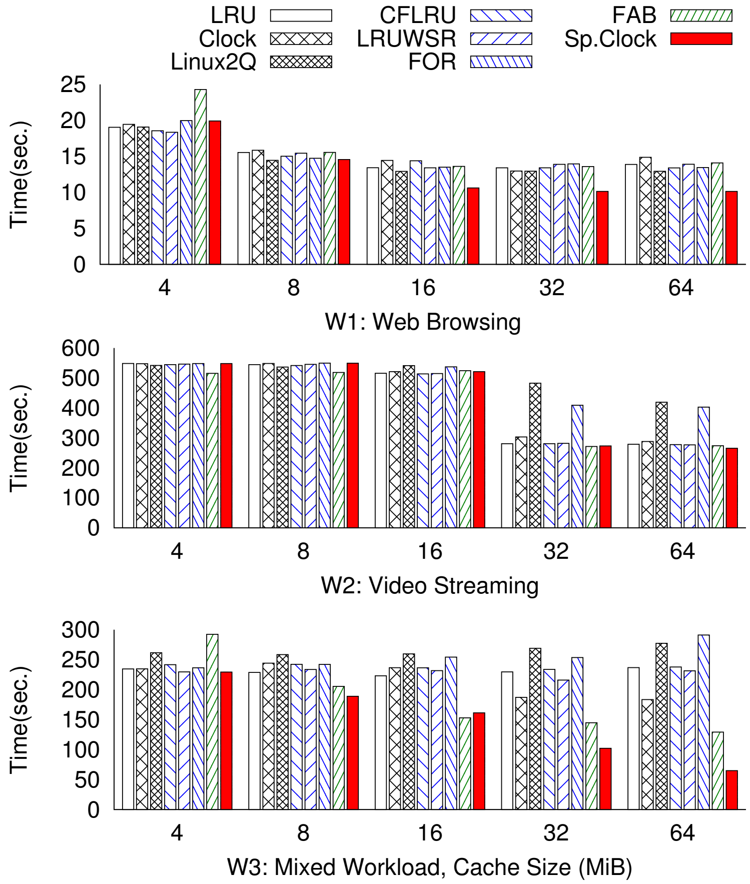


Figure 17: Patriot microSD: *SpatialClock shows -8.6-31.9%(W1), -6.4-43.4% (W2), and -5.4-77.7% (W3) elapsed time reduction compared to the other schemes.*

Figure 17-19 show the elapsed time on real flash storage devices. The measured time is represented by the bars in the graph, and shorter bars imply better performance. Unlike the mathematically calculated performance result shown in Figure 16, we can see clear differences among the seven buffer replacement schemes through this exercise of running the *after cache* traces on real flash storage devices[4].

---

[4] Please note that the absolute numbers in Figure 16 are very different from what we observed

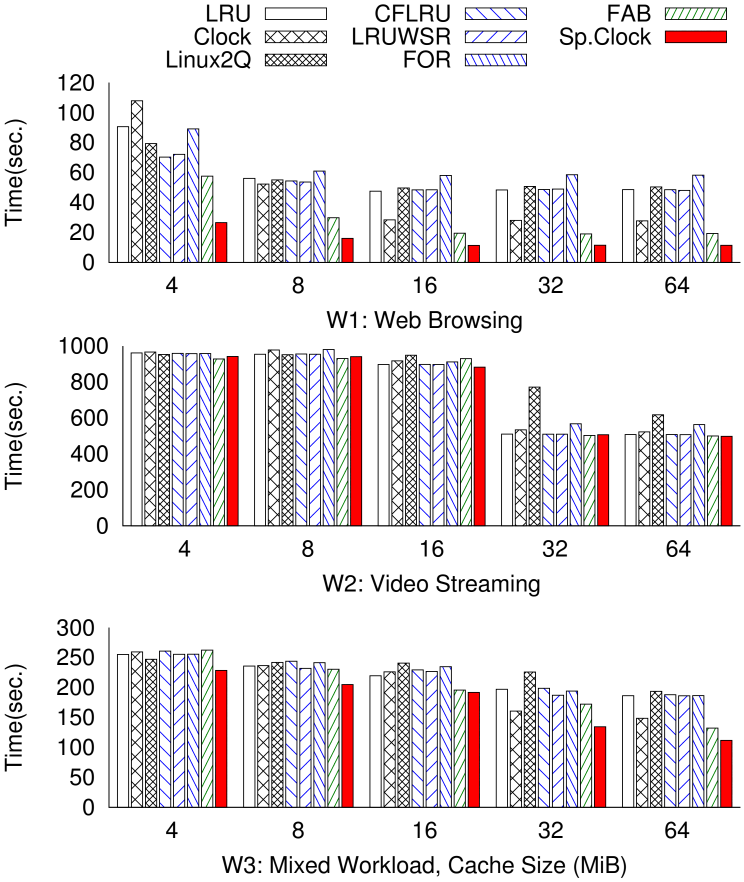


Figure 18: eMMC-1 (N900): *SpatialClock shows shows 0-80.5% (W1), -1.5-34.4% (W2), and 0-42.3% (W3) elapsed time reduction compared to the other schemes.*

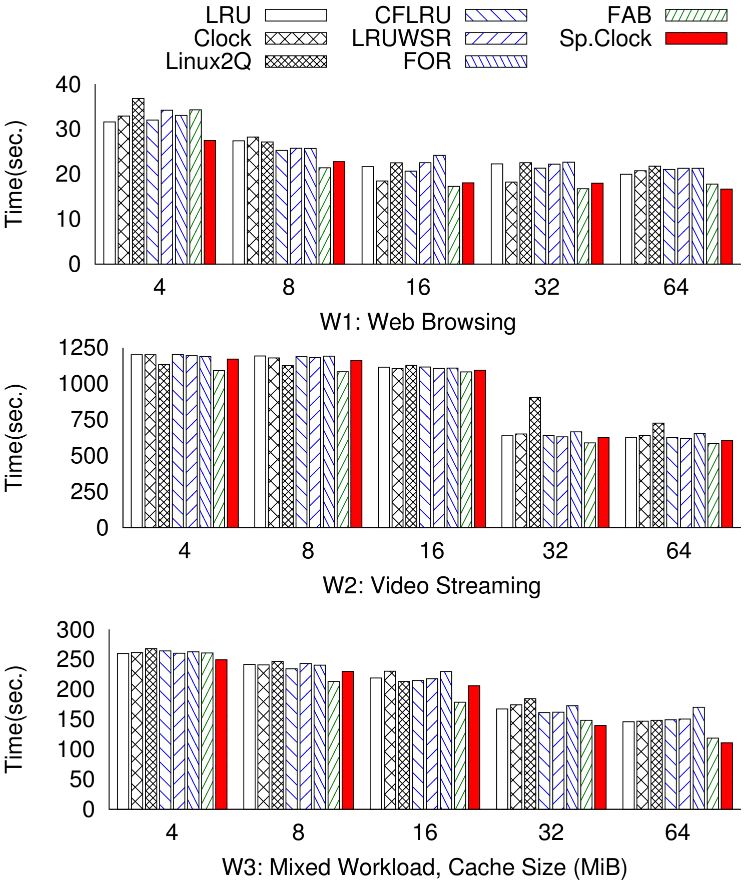


Figure 19: eMMC-2 (Nexus-S): *SpatialClock shows -7.4-25.4% (W1), -7.4-30.9% (W2), and -15.4-34.9% (W3) elapsed time reduction compared to the other schemes.*

In each cluster of the graphs, the right most red bar represents the elapsed time of SpatialClock. In general across all the mobile flash storage devices used in the evaluation, SpatialClock shows remarkable performance gains with the Web Browsing workload (W1) and Mixed workload (W3). Linux2Q and FOR algorithms show very poor performances with the Video Stream workload (W2). SpatialClock is significantly better than Linux2Q and FOR but does not show any significant performance advantage over the other cache replacement algorithms for the Video Streaming workload (W2).

As already shown in Table 6, eMMC-2 in the Nexus-S smartphone is less sensitive to write ordering, and thus, SpatialClock shows the smallest performance gains for this flash storage (Figure 19) while it shows huge performance gains for eMMC-1 (Figure 18). It is because this eMMC chip is specially designed to provide good random write performance similar to a high-end SSD, and it also implies that the interface of eMMC devices is not a differential point in the results. It is an interesting point to note that SpatialClock consistently performs better with the inexpensive microSD card than on the other the two eMMC chips, which appear to be high-end ones given their superior performance for dealing with scattered writes. One way of interpreting this result is that if the flash storage already is well positioned to handle scattered writes, then the additional performance advantage due to SpatialClock is small. However, SpatialClock does better than the other cache replacement schemes even on the eMMC-2 (24% reduction in elapsed time compared to LRU for W3 workload and 64 MB cache).

There is another very surprising and interesting insight stemming from this performance result. With 64 MB cache and W3 workload the elapsed times for LRU are:

---

from Figure 17-19. It is because the parameters for the calculation may have differences to the actual timing value.

microSD: 236.9, eMMC-1: 186.3, eMMC-2: 146.0 seconds. For the same configuration, the SpatialClock elapsed times are: microSD: 64.9, eMMC-1: 111.8, eMMC-2: 110.8 seconds. That is, the best absolute elapsed time (64.9 seconds) for this workload is achieved by using SpatialClock on the cheaper microSD card! Compare this with using the standard cache replacement scheme available in commercial operating systems running on a souped up flash storage such as eMMC and still being nearly 2.2 times slower than SpatialClock on an inexpensive flash. In other words, with the right OS support (SpatialClock), we can achieve better performance than using a hardware solution (eMMC) to circumvent the performance issues of mobile flash storage. SpatialClock is the third contribution of this study, presenting a compelling case for revisiting the replacement scheme used for the buffer cache in smartphones.

5.5 Summary

Recent studies have shown that flash storage may be the performance bottleneck for the performance of common Apps on mobile devices. Due to size, power, and cost considerations, smartphones will continue to deploy low-end flash memories as the primary storage. Therefore, it is important to consider what can be done in the OS to enhance the performance of flash based storage systems. In particular, since the buffer cache is the point of contact between the upper layers of the OS software stack and the I/O subsystem, this study re-examines the buffer cache replacement schemes with respect to their suitability for mobile flash storage. We make three contributions through this work. First, we develop a novel performance evaluation framework that is a hybrid between trace-driven simulation and real implementation. Second, we gather *before cache* storage traces for popular Apps running on an Android phone that can be used in the study of cache replacement schemes. We use this and the evaluation framework to study seven different cache replacement strategies. We made some surprising findings through this study, and the insight drawn from the study paved the way for our new buffer cache replacement scheme, SpatialClock. The key insight is the need to pay attention to spatial locality for writes to the flash storage to reduce the overall I/O time, a crucial metric to enhance the storage performance, and hence the application performance on smartphones.

CHAPTER VI

FJORD: SMART WRITE BUFFERING FOR SMARTPHONES

In this chapter, we introduce our integrated storage solution for mobile platforms.

6.1 Introduction

Smartphones have become essential parts of our daily life. In 2011, smartphone shipments overtook PCs [50], and the number of people subscribing to mobile phones is bigger than the number of people subscribing to electricity and safe drinking water [101].

Mobile systems are not just miniatures of personal computer systems, and thus, the previous research insights from desktop and server systems should not be simply applied to mobile systems without careful reexamination. For example, a recent study reveals that the storage subsystem has a much bigger performance effect on application performance on smartphones than it does on conventional computer systems [62]. Considering the rapid growth of mobile systems, it is the time to move our focus to the mobile system components. CPU power and main memory capacity are increasing very fast; the latest smartphone has a quad cored 1.4 GHz processor as well as 2 GB of main memory capacity [95]. On the other hand, the technology used for storage on mobile platforms lags significantly behind that used on regular computers. Flash storage is the norm for smartphones because of the limitations of size, cost, and power consumption. While high-end flash based Solid-State Drives (SSDs) are available and used in regular and enterprise class machines, adoption of such storage for mobile platforms is infeasible for reasons of cost, size, and energy consumption.

Therefore, we argue that operating system level software support is critically needed for low-end flash storage to achieve high performance on mobile platforms.

Architecturally, smartphones are not much different to traditional desktop and laptop computers. However, they are very different in terms of requirements and operation environments. First, smartphone is a communication device, and communication can be critically important in our daily life. We communicate each other with mobile phones everyday, some people rely on mobile phones for their businesses, and we use mobile phones in emergency situations. Therefore, users expect highest availability as well as reliability for smartphones. In addition, smartphones are being used in unstable environments than conventional computer systems. We carry smartphones in our pockets always, and smartphones may easily be "dropped" on the ground. In such situations, the battery can get separated from the phone, and thus, smartphones have higher chances to lose system power unexpectedly than regular computer systems do. To complicate matters, smartphone adoption is by a much larger community of users who are not necessarily computer savvy. Even for computer savvy users, fixing routine problems in a smartphone is not as straightforward as dealing with similar problems in a desktop or laptop. Thus problems with smartphones usually require a visit to the manufacturer's service center.

For these reasons, system reliability is always a top requirement in designing smartphones, and thus, the safer but slow design choices normally win over faster but risky ones. As a concrete example, Google Android 4.0.4 uses write barrier enabled EXT4 journaling file system instead of the faster EXT2 file system, and reduces `dirty_expire_centisecs` and `dirty_background_ratio` values from 3000 to 200 and 10 to 5, respectively. These configuration changes are to minimize the possibility of losing dirty page content due to unexpected power failures. Of course, the configurations limit the capability of Linux page cache, and it can be a limitation to the performance of smartphone storage.

Flash storage is the norm for smartphones because of the limitations of size, cost, and power consumption. Compared to Solid-State Drives (SSDs) and Hard Disk Drives (HDDs), the performance of smartphone storage is much more limited. Sequential write throughputs are about 400 MB/sec and 150 MB/sec on a latest SSD and HDD, respectively [108], but it is only 13 MB/sec on smartphone storage (SanDisk eMMC [97]). That is, smartphone storage is slower than regular storage, and furthermore smartphone OS storage software stack is configured mainly for safety rather than performance. Consequently, the low-end flash storage easily becomes the Achilles' heel when it comes to performance of mobile platforms [61, 62].

Meanwhile, there is a very different and common use case of smartphones. Communication is a critically important function of smartphones, but smartphones are being used for various other purposes, and many of them do not require the same high standard for reliability. In many cases, smartphones are terminal devices for cloud contents, and local smartphone storage is used mostly as a cache for data that already resides safely in the cloud. For such situations, loss of the cached content is not catastrophic since the original content is safely in the cloud. Facebook, web browser, Twitter, Google Maps are good examples; local storage is used to hold the copy of cloud data. By their very nature, cache store does not require high reliability, and thus, we argue that it is neither necessary nor prudent to sacrifice performance for reliability. However, there is no systematic method to selectively control the conservative storage configurations of the smartphone OS for such applications that can use relaxed semantics for reliability to gain higher performance. Therefore, overall storage performance is unfairly degraded on smartphones. In other words, if smartphone OS provides a fine-grained control mechanism to tradeoff reliability for performance, then it will be possible to get better performance for some applications without losing reliability for critical applications (e.g., bank transactions).

In this study, we explore the capabilities and limitations of storage solutions for smartphones. We first design and implement two typical approaches for flash storage, *logging* and *RAM based write buffering*, and then we evaluate the storage performance effects of the two solutions. As a further step, we propose an integrated solution with logging and write buffering, and explain how the integrated solution can make synergy effects. Finally, we expand our solution even more to obtain bigger performance gains based on system wide information. We call our system solution *Fjord*[1], which is aimed at providing fine-gained control for trading reliability for performance in existing file systems as well as for the novel features we have proposed as part of an integrated solution, namely, logging and write-back buffering.

We implement and evaluate our solution on two real Android smartphones, and demonstrate significant performance gains with SQLite benchmark application as well as multiple everyday applications. For instance, the elapsed time for running the Email test case is reduced from 34.6 seconds to 16.1 seconds on Samsung Galaxy Note phone with Fjord.

We make the following contributions through this work. First, we design and implement typical logging and write buffering solutions, and show their performance effects on real smartphones. Second, we propose a novel integrated solution with logging and write buffering, and third, we uncovered an interesting granularity problem for cache/buffer management, which we elaborate on in later sections. The final contribution is Fjord, an integrated storage solution for smartphones. Even though we are focusing on smartphone storage in this paper, we believe that the idea has potentials beyond smartphones for other types of storage and systems, and we discuss this potential in Chapter VII.

---

[1] The English word Fjord is derived from a Scandinavian word that signifies a narrow and often shallow area in a river for crossing from one side to the other on foot...an analogy for our thin system software layer that allows safely moving data from higher levels of system software to the storage device.

6.2 Background and Related Work

File system operations can be categorized as user-data operations or file system metadata operations, and in general, we need to be more careful for metadata operations because inconsistent file system metadata may result in the entire file system becoming unusable. Therefore, safe and efficient update of file system metadata has always been an important topic in file system research. Soft-update was proposed to provide stronger reliability guarantees than journaling, and it attacks the meta-data update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/Os [42, 43], and Seltzer el al. compared the file system performance of Soft-update and journaling [100].

Ensuring write ordering is an essential part of both Soft-update and journaling file systems. Most storage devices have volatile on-board write buffer to improve storage performance, and consequently write ordering is not guaranteed. In other words, the storage devices internally decide as to when the writes pending in the on-board write buffer are committed to the physical medium. To enforce write ordering under these circumstances, storage devices expose a *write barrier* interface to the OS. Whenever a storage device receives a write barrier from the upper layers of the OS, it has to ensure that the content of the on-board buffer is written to the physical storage media safely. That is, a write barrier limits the capability of on-board write buffer, and thus frequent use of write barrier can degrade the overall file system performance significantly. For this reason, EXT3 file system turns off write barrier by default even though it is a journaling file system. In EXT4 file system, the write barrier option is turned on by default for the safety of a file system.

In the latest Android version 4.0.4, EXT4 file system is used as the default file system. EXT4 provides three different data modes related with file system consistency: write-back, ordered, and journal modes [10]. With the fastest write-back mode, EXT4 does not journal user-data at all, and provides only file system metadata consistency.

That is, a crash can cause incorrect data to appear in files, which were written shortly before the crash. When the ‘ordered’ mode (which is the default) is used in EXT4, once again only the metadata is written to the journal, but the metadata update happens only after the associated data blocks have been written to the storage first. With the safest - but slowest - journal mode, both user-data and file system metadata are written to the journal first, than written to their final locations. Within the latest Android version 4.0.4, EXT4 file system is being used with the default *data=ordered, barrier=1* options, and the performance of the option is between write-back and journal options. In EXT4 file system, it is not possible to use different journaling options for different files even though each file has different level of reliability requirements.

Flash memory has brought about a drastic change in storage technology recently. Some studies propose totally new storage systems using flash memory. FAWN [24] and FlashStore [36] have been proposed as new key-value stores using flash memory, to save power consumption and achieve better performance. These ground-breaking approaches are desirable to advance the research in storage systems with a long-term view, but are far from practical usage for general smartphone users.

Griffin [104] system has been proposed to extend the lifetime of flash storage by caching data with a HDD. Griffin system is free from the reliability issue. The authors explicitly mention that they rule out the RAM buffering approach due to the reliability issue. However, we believe that RAM buffering has many attractive merits compared to disk buffering and proper design of RAM buffering can overcome the reliability issue.

New I/O scheduling algorithms have been proposed for flash storage. Kim et al. proposed the Individual Read Bundled Write (IRBW) algorithm which separates read scheduling from write requests and arranges write requests into bundles [68]. Dunn and Reddy also proposed a new Block Preferential I/O scheduler for flash storage [39]. Even though these I/O scheduling schemes reflect the characteristics of flash devices quite well, the queuing mechanism of I/O schedulers limits the performance gain. Fjord has the same goal as that of I/O schedulers, but uses a different mechanism.

CFLRU[83] and LRU-WSR[58] are new buffer management schemes for flash storage. DULO[53] is another buffer management scheme, which is aware of both temporal and spatial localities. Apart from the fact that these algorithms have been proposed for the OS buffer cache, there are some important technical differences in comparison to Fjord. CFLRU and LRU-WSR give high priority to dirty pages being kept in the cache to reduce the number of writes; but they do not worry about ordering of write requests. Similarly, the DULO algorithm uses only the size of the requests and does not take care of reordering as Fjord does.

At the device level, more complicated FTL mapping algorithms have been proposed to attain better write performance [81, 82]. However, due to the increased resource usage of these approaches, they are generally used only for high-end SSDs.

Incorporating a write-buffer inside a flash storage device is a slightly higher-level approach than FTL. For example, we previously proposed Block Padding Least Recently Used (BPLRU) [63] as a buffer management scheme for flash storage and showed that even a small amount of RAM-based write buffer could enhance the random-write performance of flash storage devices significantly.

Smartphones just a year back used to have a bare NAND flash memory chip with a flash native file system like YAFFS2 [74], but the latest ones use eMMC devices rather than bare NAND flash memories. eMMC devices are produced in small Ball Grid Array (BGA) packages, and present a Multi Media Card (MMC) interface to the host computer [78, 97]. The upshot is that each eMMC chip has FTL software internally by using System-On-Chip technology. This approach is desirable both for chip manufacturers and handset makers. The software layer within an eMMC chip hides complicated chip level details from eMMC users, and shields the users from issuing erroneous commands to the underlying NAND flash memory. The standard interface is also helpful for handset makers. Eliminating the need for FTL and/or flash native file system (such as YAFFS2) on the host side helps rapid development, and the unified interfaces (at the storage system software level) can be used by the mobile platform both for an internal eMMC chip and for an external flash memory card like microSDHC. Therefore, it is safe to assume that for the foreseeable future mobile flash storage devices will be either eMMC and/or small microSDHC cards.

Naturally, small flash memory cards (including eMMC) suffer from many limitations. Only small amount of RAM is available for internal FTL, and these flash memory cards have severe limitations when it comes to response time and power consumption. As a result, most inexpensive flash storage devices show very poor performances especially for small random write requests, and as a consequence, inexpensive flash storage devices remain as the main source of performance bottleneck on mobile platforms. Therefore, we argue that operating system level software support is critically needed for low-end flash storage to achieve high performance on mobile platforms.

6.3 Log-structured Non-Volatile Write Buffer

Log-structured design to combat the "small write" problem solves the random write issue in Flash storage, and has been popularly used in many previous studies [24, 36, 70, 74, 87]. In a log-structured architecture, sectors are written sequentially regardless of their original addresses, and the mapping information is maintained by the system. This approach is desirable for flash storage since a page write has to be preceded by an expensive block erase operation. However, a log structured architecture has its own limitations. First, maintaining the mapping information is very expensive. Because the sectors are not written to their original positions, to read sectors, such mapping information is necessary. For better performance, it is desirable to keep the information in main memory. The mapping information also has to be safely stored in non-volatile storage, and it requires additional flash I/O operations. Second, the log-structured design is good for write but bad for read performance. Sectors are not in their original positions, and thus sequential reads are not sequential any more. Even though Flash storage is less sensitive to the request ordering for read operations, it is still the case that sequential reads are much faster than random reads.

For multiple reasons, we decide to do logging, only for a non-volatile write buffer at the block device level. Block device level solution is more general and easier to apply, and the write buffer approach allows us to use logging selectively. Figure 20 shows our design for non-volatile write buffering. In this design, foreground small writes can be changed to big sequential writes, and the storage integrity is not affected. We also use SpatialClock to reclaim the logging buffer. By using SpatialClock, sequentially ordered writes are generated, and thus, we can minimize the operation cost for the garbage collection. Unlike server systems, smartphones have long idle time in general, and the logging buffer can be reclaimed when a smartphone is idle. As long as there is enough space in the logging buffer, the foreground write requests will be quickly handled.

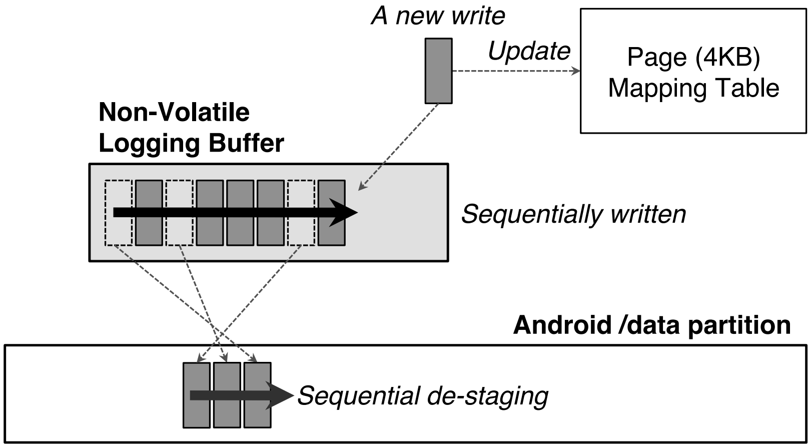


Figure 20: Non-Volatile Logging Buffer: *new pages are written to logging buffer sequentially, and de-staging also happens sequentially due to SpatialClock*

6.3.1 Mapping Information Maintenance

Linux virtual memory page size is 4Kbytes. Therefore it is convenient to use the same size as a logical to physical address mapping unit. If we maintain 32bits entry for each page, 1Gbyte of flash storage will require 1Mbyte of main memory for the mapping table. Today's Android smartphones have 8 to 64 Gbytes of flash storage in total, but only 1-2 Gbytes space is allocated for internal /data partition, which is the main storage area for Android applications. When we apply logging for the internal /data partition, only a small amount (a few Mbytes) of main memory will be required for the mapping table, and this amount is small enough considering the typical main memory size of today's smartphones (typically at least 512Mbytes). Thus, memory requirement is not a problem to implement logging. The tricky part is how to keep the content of the mapping table safely.

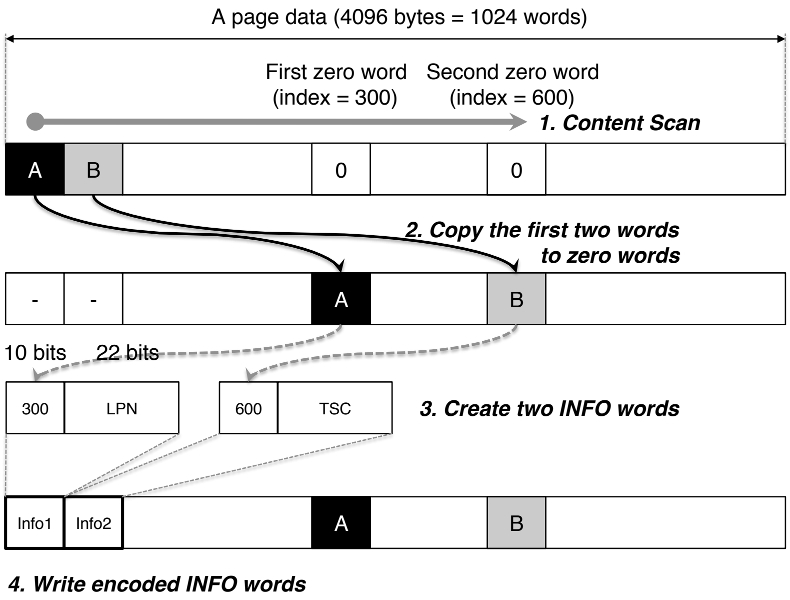
In NAND flash memory each data page has spare array (typically, 16 bytes per 512 bytes), and the space is used to store Error Correction Code (ECC) and some useful information like FTL mapping information. This addition space is very useful for logging because the mapping information can be written without any additional write operations. However, we cannot use the spare array of NAND flash memory because it is invisible from the outside. We may have to reserve small part of flash storage for the mapping information like most file systems do for file system metadata such as inode and bitmap information. However, this approach requires at least one additional write operation whenever mapping information changes, and for the block device level logging, every write request results in the update of mapping table, which means that the amount of writes will remarkably increase. We can delay mapping information updates for better performance, but it will sacrifice storage reliability. For these reasons, we use a different approach to store the mapping information safely in flash storage without additional flash write operation.

Two pieces of mapping information are critical for the logging technique: (a) the logical page number (LPN) and (b) the time stamp counter (TSC). LPN is necessary to re-construct the logical-to-physical page association, and TSC is needed to know which page is the latest one when multiple physical pages are mapped to one logical page. The mapping table contains these two pieces of information for each logical page.

To avoid separate I/O for mapping table update, we embed the mapping information in the first two 32-bit words of the data page itself. How is this possible? The trick is as follows. When a new data page is presented to Fjord, it scans it for two 32-bit words in the page that contain zero values. Most of the time, we will succeed in finding two zero value words in a 4Kbyte page. In the event we fail to find two zero words in the page, we simply give up and do not use the logging approach for this page.

Assuming we find the two zero value words in the page, then we do the following. We first write the contents of the first two words in the page to the positions where we found the two zero value words in the page. Then, we use the first two words of the page to store our mapping information as well as the locations of the first two words of the page that have been displaced from their original positions to make room for the mapping information. With 4Kbyte pages, the zero value word locations can be represented with 10-bits. The remaining 22-bits in each of the first two words of the page are available for storing the LPN and TSC, respectively. Figure 21 shows an example of the mapping information encoding trick.

For the encoding process (embedding LPN and TSC into the content of a page), the data page has to be scanned for identifying the locations of the two zero value words in the page. This is in the critical path a data write operation. The good news, however, is that the decoding process (extracting LPN and TSC, and recovering the original page content for the first two words) is much more efficient and requires at most two memory reads and four memory writes. This is a reasonable compromise since most page read operations are synchronous while most page write operations are asynchronous.



A page data (4096 bytes = 1024 words)
First zero word (index = 300)
Second zero word (index = 600)
1. Content Scan
A
B
0
0
2. Copy the first two words to zero words
-
-
A
B
10 bits
22 bits
300
LPN
600
TSC
3. Create two INFO words
Info1
Info2
A
B
4. Write encoded INFO words

Figure 21: Example of embedding the mapping information into the page itself: *in this example, zero value words are found at page offsets 300 and 600. The contents of A and B are copied into the zero word locations 300 and 600, respectively.*

6.3.2 Sequential Garbage Collection

In a log-structured architecture, one of the most important problem is how to reclaim the storage space, i.e., Garbage Collection (GC) process. In fact, the foreground logging is easy and straightforward while background garbage collection is difficult and complicated. In a typical GC process, a candidate storage segment is chosen first based on the GC cost. In general, the segment having the least number of valid data pages is chosen as a target of the GC process. Then, the valid data pages in the target segment are copied to a new segment to make the target segment perfectly clean. Our GC technique is quite different. First, we do not manage the logging space in segment units. Instead, we treat the whole logging buffer as a one big space. Second, we do not relocate data pages within the logging buffer to make a big free chunk. When there is no more room to write in the logging buffer, we sequentially scan the page-mapping table from the position where the last GC happened. Because we scan and choose the victim pages based on the logical page ordering, GC generates sequentially ordered writes. The idea comes from our flash aware buffer placement scheme, SpatialClock [66]. Unlike RAM based cache space, the logging buffer is a not faster area then the original data partition for read operations. Therefore, there is no need to keep hot pages in the logging buffer; we ignore temporal locality and only respect spatial ordering. As a result, generated writes (foreground writes as well as GC writes) may not be big and perfectly sequential. If we follow the typical GC method to make foreground writes be perfectly sequential, we have to clean a target segment completely. The valid data pages can go either to the final destination or a new clean segment. Both are detrimental to performance: the former will result in random writes; the latter will trigger GC more frequently. For the former case, random writes will happen in background, and for the later case, GC will happen more frequently. We believe our approach strikes a good balance between the foreground write performance and the background GC cost.

6.4 Flash Aware RAM Buffering

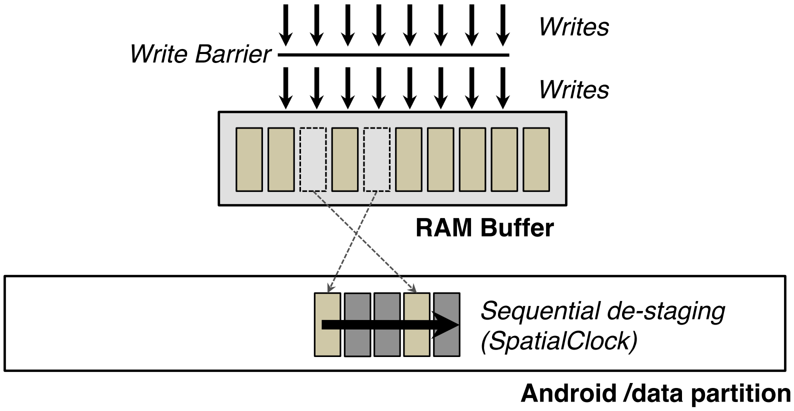


Figure 22: RAM based Flash Aware Write Buffer: *Write-back buffering is safe as long as it follows write-barrier semantics*

All HDDs internally have RAM buffer for better performance, and written data can stay in the write buffer during some amount of time even after the write request is completed. The upper layers often want to ensure all data are safely written to non-volatile storage device, and for that purpose, special commands are being used. *Write barrier* is used at the file system level, and a special SCSI command (`SYNCHRONIZE_CACHE`) is used at the device level. As long as we follow the write barrier semantics, write back buffering is safe.

In our previous work BPLRU [63], we have shown that a small write buffer can improve random write performance significantly, and as a further step, we have implemented a host-side write-back buffering solution, named FlashFire [65] for regular desktop and laptop computers. FlashFire has been moderately successful for improving flash storage performance on Windows OSes, and has a user community of over 100,000. To know the capability of host-side write buffering for mobile platforms, we re-design write buffering layer for Android smartphones.

Figure 22 shows the core design concept of our buffering layer. We allocate small sized (i.e., 16Mbytes) main memory for write buffering, and the layer sits between a file system and block device layers. We use the SpacialClock [66] algorithm for buffer replacement, and strictly follow the write barrier semantics. We also implement a mechanism, which drains write buffer during idle time to minimize the reliability concern for write-back buffering.

We will explain the performance effect later in following evaluation section in detail. As a preview to the evaluation results, we would like to observe at this point that write buffering is not very beneficial to performance with the EXT4 file system due to the frequent generation of write barriers by the file system.

6.5 Integrated Write Buffering

Since write-back buffering in of itself is not as effective as we had hoped, we integrate the RAM buffer solution with a logging buffer solution. Figure 23 shows how the

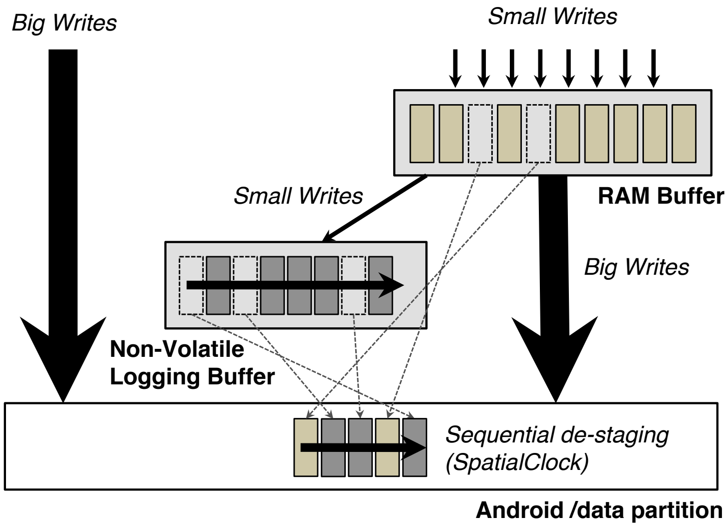


Figure 23: Integrated Write Buffering: *RAM buffer and logging buffer are integrated together*

two solutions are combined together. Our main design goal with this approach is to save logging buffer area by using the RAM buffer. RAM buffer acts as a staging area for small write requests; spatially near, but temporally separated write requests are merged to a big write request, and directly written to the final location. However, as we will later see in the evaluation section, even this combined solution does not yield much performance gains if the RAM buffer follows write barrier semantics strictly. We will discuss the results in a later evaluation section in detail.

6.6 Fjord: Fine-grained Reliability Control

Today's smartphones are sacrificing performance to protect themselves from unexpected power losses. However, not all applications require such high reliability; many of them use local storage just as a cache store, and thus, it is neither necessary nor prudent to sacrifice performance for data integrity for such applications. We propose *Fjord, a fine-grained selective write buffering layer* to improve chosen application performance without affecting the reliability of other applications.

6.6.1 Design Principles

In this study, we focus on page cache and write buffering because smartphone storage has limited performance for writing. Write-back buffering has multiple benefits. It can reduce the amount of write requests, and it can also be used to reorder write requests so that the pattern of write requests become more desirable from the point of view of the underlying storage. There are three design principles embodied in the design of Fjord: (1) The first principle is aggressive write-back buffering. Two supporting mechanisms make this possible, namely, *fsync() control* and an additional *write-back buffering layer* between the page/buffer cache and the block device layer. (2) The second principle is ensuring the consistency of the file system at all times. To cater to this principle, Fjord distinguishes between user data and file system metadata, bypassing the write-back buffering layer for all metadata writes to the storage device. (3) The third and last principle is selectivity in applying the performance enhancing write-back buffering mechanisms at the granularity of individual files even for user data. This ensures that applications are able to choose when to trade reliability (for user data) for performance at the granularity of individual files.

6.6.2 Architecture

Figure 24 shows the overall architecture of Fjord together with that of Android system. Android system is based on Linux kernel, and our storage software stack is almost the same but for the conservative configuration of the original Android system. We add a new write buffering layer between file system and block device layers, and this layer optimizes write requests considering the general performance characteristics of flash storage. To distinguish file system metadata from user-data, we slightly modify Linux page cache system. We also modify file system code to control overly used fsync() function calls. The implementation details for each component will be explained in the following sub sections.

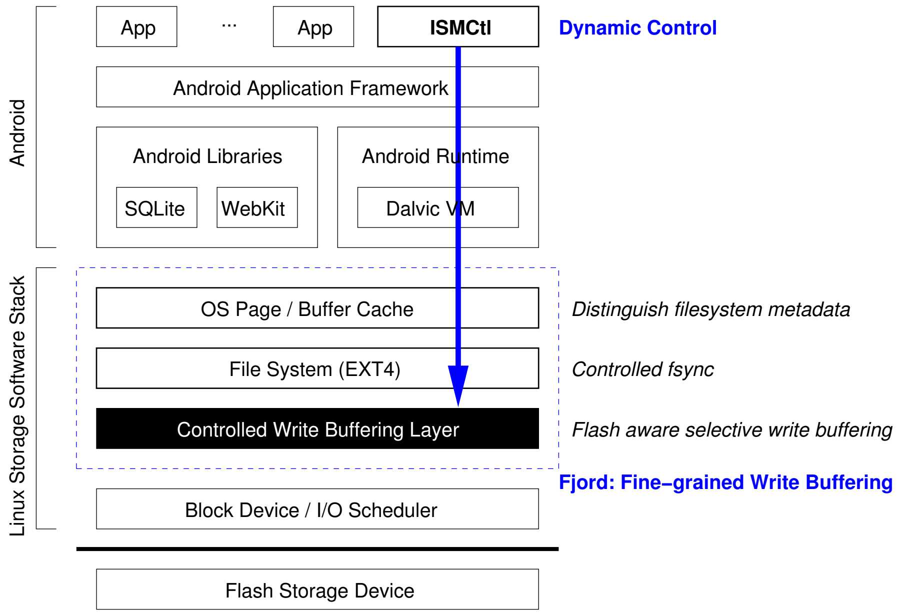

App
...
App
ISMCtl
Dynamic Control
Android
Android Application Framework
Android Libraries
SQLite
WebKit
Android Runtime
Dalvic VM
Linux Storage Software Stack
OS Page / Buffer Cache
Distinguish filesystem metadata
File System (EXT4)
Controlled fsync
Controlled Write Buffering Layer
Flash aware selective write buffering
Fjord: Fine–grained Write Buffering
Block Device / I/O Scheduler
Flash Storage Device

Figure 24: Bird eye view of fine-grained write buffering

6.6.3 File Level White List Control

To provide fine-grained control for write buffering, we modify Linux kernel to maintain a *white list* (opposite concept to blacklist). In Android system, each application writes only to well separated own storage space under `/data/data/` directory. In our prototype, a file path can be easily inserted to or deleted from the white-list by using `/proc` file system.

The entries in white-list are compared whenever a file is opened. We modify `do_sys_open()` function within `fs/open.c` file to check if the opened file is buffer-able or not. We also add a new variable to the `file` structure of Linux to denote that the content of the file does not require high reliability.

In our prototype implementation, we provide two different interfaces to control the white list. The first interface is an ioctl command, and it lets application developers to enable write buffering at file level. The other interface is based on Linux proc file system, and this interface lets users or smartphone manufacturers to control per file write buffering.

6.6.4 Bypassing File System Metadata

To protect file system from unexpected power losses, Fjord distinguishes file system metadata from user-data. It can be done of course by file systems, but we take a different approach. Instead of modifying file system code, we modify page cache related, `generic_perform_write()` function within `mm/filemap.c` file. The function is called when a file is written by file system write APIs, and it eventually copies the written data to a page frame in Linux page cache. Clearly, the page is holding user-data of a file, and we annotate that within `page` data structure. For this purpose, we add a new page bit flag (`include/linux/page-flags.h`) and set the bit within `generic_perform_write()` function. At this time, we also check newly added variable within `file` structure (`generic_perform_write()` function receives `file` structure as a argument), and mark the page as buffer-able page only when the associated file is marked as buffer-able file. We verify the correctness of this method in the following section.

6.6.5 Controlling fsync

Recent study about Android smartphone storage [62] reported that Android applications use database interface extensively for various purposes, and SQLite database engine uses fsync() to support transactional semantics. It turns out that such transactional semantics is an overkill in many situations. For example, when a web browser visits a web site, it downloads multiple image files to the local web cache (on the storage device), and uses SQLite to maintain the index for the stored items. It is just convenient for the application to use SQLite for generating the indexes. However, the application does not need strong transactional semantics for the objects it stores and indexes in the web cache since they are available in the cloud (i.e., web servers in this case). Unfortunately, SQLite has no way of knowing this, and it provides strong transactional semantics using fsync() liberally every time it stores an index into the web cache. This is the motivation behind the fsync() control in Fjord, which allows an application to inform the file system that sits below the database (see Figure 24) that it can selectively ignore fsync() calls to specific files.

The implementation is extremely simple. We already explained file level annotation mechanism and related modification for `file` structure. We modify `ext4_sync_file()` function within `fs/ext4/fsync.c` file to return when the file is marked as a bufferable file. This mechanism influences only the selected file, and it effectively reduces the amount of metadata writes. It is helpful because underlying write buffering layer is designed to bypass file system metadata.

6.6.6 Over Block Device Write Buffering

Our experience with BPLRU [63] and FlashFire [65] show that RAM buffering is a very powerful mechanism, which can effectively improve random write performance. However, in the current Android based smartphones, dirty data cannot stay in the RAM buffer for a sufficiently long time because of the too frequent write barriers from the EXT4 file system. As we have already observed, this is to make smartphone storage system robust and resilient to sudden power failures. Even though some writes are from non-critical applications, we cannot disobey the write barrier semantics, and have to flush all dirty data in RAM buffer.

In Fjord, we propose *fine-grained control of write buffer to improve performance without hurting the reliability of smartphones.* As shown in Figure 25, Fjord distinguishes non-critical data from other critical data, and allows the non-critical data to stay in the RAM buffer in spite of a write barrier. Thus, if a power failure happens, the data content of non-critical applications could be lost, but it is not a serious problem because the original copy safely exists in cloud. We want to emphasize that Fjord will never lose critical data due to write buffering since we strictly obey write barrier semantics for such data.

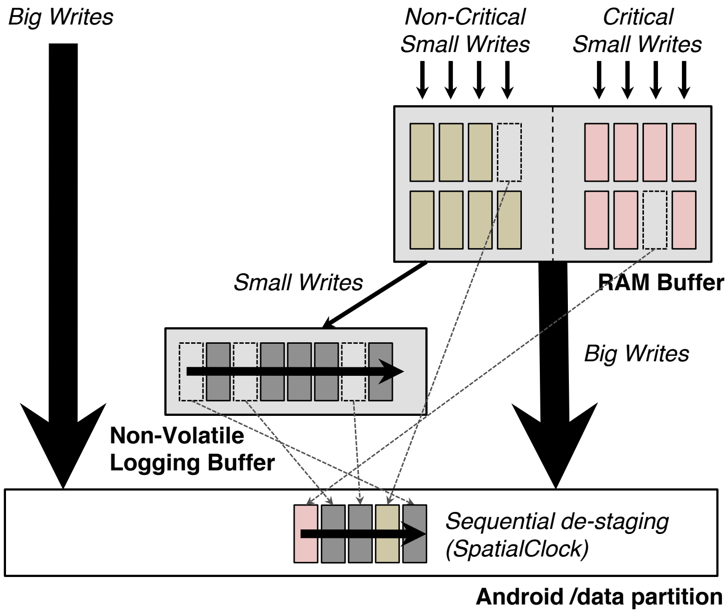

Big Writes
Non-Critical Small Writes
Critical Small Writes
Small Writes
RAM Buffer
Big Writes
Non-Volatile Logging Buffer
Sequential de-staging (SpatialClock)
Android /data partition

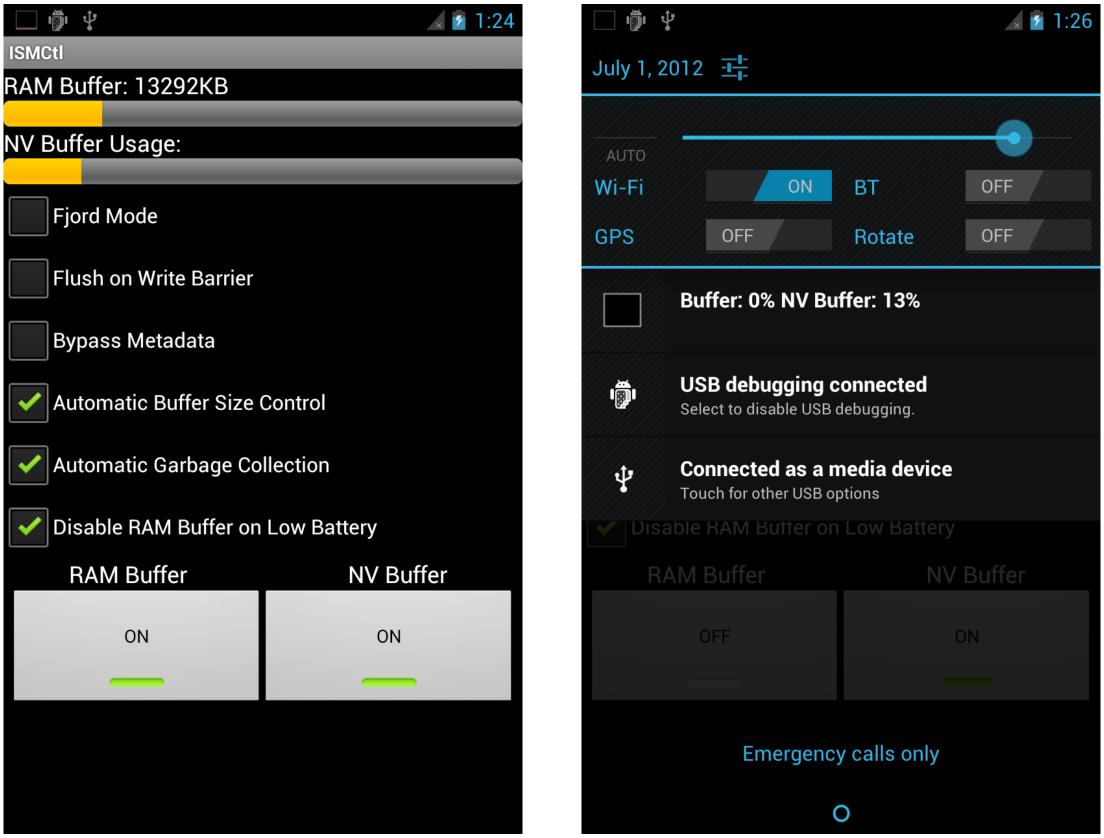
Figure 25: Fjord, logging, and RAM buffering: *Non-critical pages are allowed to stay in RAM buffer against for a write barrier*

Fjord also has the spontaneous buffer draining mechanism to trigger a partial buffer flush during periods of low I/O activity. By draining write buffer during idle time, Fjord ensures that it has enough buffer space for handling future write traffic.

6.6.7 Dynamic Write Buffering Control

User Interface

Fjord's write buffering is visible to all the layers in the software stack from application down to the buffer cache. Therefore, it can be freely enabled or disabled for different scenarios. At the application level, a user may explicitly control write buffering for more performance or for increased reliability. We design the Fjord buffering engine to be fully controllable by using the `/proc` file system, and we have also written a small Android App named *ISMCtl* (Figure 26 (a)) as a user interface. Users easily can turn on and off RAM buffer and logging buffer selectively, and also can decide whether to follow write barrier semantics or not. The buffer usage is always shown through the top Android status bar, and users can enter to ICMCtl App by dragging down the top status bar and clicking clicking the buffer usage information (Figure 26 (b)).



ISMCtl

RAM Buffer: 13292KB

NV Buffer Usage:

☐ Fjord Mode

☐ Flush on Write Barrier

☐ Bypass Metadata

☑ Automatic Buffer Size Control

☑ Automatic Garbage Collection

☑ Disable RAM Buffer on Low Battery

RAM Buffer: ON

NV Buffer: ON

Wi-Fi: ON

BT: OFF

GPS: OFF

Rotate: OFF

☐ Buffer: 0% NV Buffer: 13%

USB debugging connected
Select to disable USB debugging.

Connected as a media device
Touch for other USB options

☑ Disable RAM Buffer on Low Battery

RAM Buffer: OFF

NV Buffer: ON

(a) (b)

Figure 26: Fjord Dynamic Control App (ISMCtl): *(a) Fjord control interface, (b) buffer usage in Android notification bar*

Automatic Buffer Size Control

By default, we allocate 16Mbytes of main memory for Fjord, and the memory is used for page-mapping table and write buffer. Because it is hard to decide the proper buffer size, and we do not want to waste memory, we design the Fjord buffering engine to be able to adjust RAM buffer size at runtime. ISMCtl periodically monitors the buffer usage level in background, and increases or decreases the buffer size on predefined conditions; in our prototype implementation, buffer increases when buffer usage level is over 80% during any 3 second interval, and decreases when buffer usage level is lower than 10% during in any 5 second interval.

Automatic Garbage Collection

In a logging solution, Garbage Collection (GC) timing is very important. Lazy GC can maximize the logging effect, but eager GC will minimizes response time. By default, Fjord does lazy GC; GC happens when there is no space to write in the logging buffer. This approach is easy to implement because there is no need to decide when the buffering engine triggers GC.

Meanwhile, smartphone users are sensitive to the foreground performance while smartphones have long idle time. Therefore, it will be desirable to do GC when the smartphone is idle. Android framework provides a lot of useful information. However, Fjord buffering engine is implemented as a part of Linux kernel, and therefore it is not easy to access such information within the Fjord buffering engine. Therefore, Fjord relies on ISMCtl App to trigger GC based on higher-level information. When *Automatic Garbage Collection* option is turned on, ISMCtl App detects when the LCD screen is off for over 5 seconds, and triggers the GC function to drain the logging buffer. This is a good example for showcasing how system wide information can be used for our storage solution, and it is the key idea of Informed Storage Management.

Smart RAM Buffer Disabling

Users may want to use write buffering for better performance even though there is a risk for losing data. In that case, ISMCtl App can automatically disable RAM buffering when here is high possibility for a power failure. To show the feasibility, we implement one mechanism; when the `ACTION_BATTERY_LOW` message is given to Fjord ISMCtl App (Android framework sends the message when the battery level is very low), ISMCtl disables RAM buffer automatically. The idea can be extended with other types of information as well. For example, RAM buffering can be turned off when the user is jogging (based on sensor information) as there is a high probability of power failure.

6.7 *Evaluation*

Fjord has been implemented into real Android smartphones, and evaluated with several popular Apps on multiple flash storage devices. In this section, we first start with our evaluation environment, and explain our evaluation results focusing on the following two key questions:

- Reliability verification: How effective is Fjord in ensuring the reliability of mission critical applications?
- Performance impact: How effective is Fjord on enhancing the performance of everyday applications?

6.7.1 Evaluation Environment

We use two Android smartphones, Samsung Galaxy Note N7000 [93], and Google Nexus One [11]; Galaxy Note phone has 16Gbytes of internal eMMC flash memory; Nexus One phone has 512 Mbytes NAND flash memory; both phones have external memory card slots. We test the internal eMMC device and 8Gbyte sized class 10 microSDHC card from Samsung. Table 9 summarizes the flash devices used in this study.

Table 9: Read/Write throughputs of flash devices used in this study (Mbytes/second).

Device	**Size**	**Seq. Read**	**Seq. Write**	**Rand. Read**	**Rand. Write**
SD Card	8 GB	18.3	12.0	2.5	0.01
GNote eMMC	16GB	40.7	15.1	5.4	0.43

The Galaxy Note phone has a dual-core 1.4GHz Samsung Exynos processor, 1 Gbytes main memory, and a 5.3 inch 1280 x 800 resolution LCD screen. As a test software platform, we use a custom ROM for the Galaxy Note phone, named AOKP build 38 [2], which is based on Google's Android Open Source Project (AOSP [3]) version 4.0.4 (Icecream sandwich). AOSP only supports Google's official reference phones while AOKP is available for more smartphones including Galaxy Note. For the Linux kernel, we use version 3.0.15 from Samsung.

The Nexus One has a single-core 1 GHz Qualcomm QSD8250 Snapdragon processor, 512 MB RAM, and 512 MB internal flash storage. For this phone, AOSP version 2.3.7 (Gingerbread) and Linux kernel 2.6.35.7 are used. We use the latest available software for both devices on current point.

EXT4 file system is used for all tests, and 16 MB of RAM is assigned for write buffering layer. We apply write buffering only for `/data` partition, which is used as a writable application storage within an Android system.

6.7.2 Reliability Verification

The key idea of Fjord is improving the performance of chosen applications without hurting the reliability of other applications. To this end, we control fsync() at file system level, and adopt additional write-back buffering layer over the block device layer. Related with verification of safety, by design our fsync() control does not compromise reliability since we apply it only for chosen files. However, it is not very intuitive how the additional write-back buffering layer does not violate the reliability needs of critical applications that have chosen NOT to sacrifice reliability for performance.

An important question is whether Fjord is really able to distinguish buffer-able data from un-buffer-able data or not. To verify the correctness of our implementation, we use postmark benchmark.

Postmark is a well-known benchmark, which emulates the mail server workload. It first creates a lot of text files, modifies the contents of randomly chosen files, and deletes the files. Postmark is known as metadata intensive benchmark. Originally, it writes randomly generated text content to test files. For verification purpose, we modify postmark source code to write special fingerprint instead of random text data, and we insert a piece of verification code in our write buffering driver. By checking the content of associated buffers of write requests at the write buffering layer, we can find out whether a write request is for user-data or not. Then, we can verify whether Fjord is properly doing buffering only for chosen application user-data or not.

We compile postmark for Android and run this test on a real smartphone. We configure postmark runtime parameters as follows: Transactions = 1000, number of files = 1000, file size = 4KB to 400KB. In this configuration, postmark generates total 87,692 write requests, and among them, 1,247 requests are for metadata and rest 86,445 requests are for user-data. We have verified from the results that Fjord successfully distinguished all 1,247 write requests for file system metadata.

With this experiment, we can verify that Fjord bypasses file system metadata and does not influence file system integrity. However, the buffered file content can be lost due to sudden power failures, and we want to know the impact on the application when the content of a file is partially lost. We built a simple program, which fills zeros to a randomly chosen part of a file currently in use by a cloud-backed Android application such as a web browser. We have done this "controlled corruption" of data files used by multiple cloud-backed Android Apps such as facebook, twitter, and web browser. In most of our tests, the Apps run without any problem whatsoever, despite such data corruption (since most likely they do an internal consistency check of the data files and fetch from the cloud if the consistency check fails). In some cases, we have noticed that an App may crash (most likely because it uses the data file without an internal consistency check). Even in the event of an App crash, simply restarting the App allows it to recover where it left off (most likely the App clears any inconsistent state upon restart). According to our observation, when Android database engine finds errors within a database file(such an index file used by the web cache), it deletes the inconsistent database file automatically. Then, the application re-creates the database file by downloading information afresh from its cloud source. Therefore, we are able to verify that Fjord does not cause catastrophic problem. An interesting question, which is part of our future work (see Chapter VII), is a thorough study of cloud-backed applications and the effect of data file corruption due to sudden power loss.

6.7.3 Performance Evaluation

To show the performance effect of Fjord on Android applications, we first measure application performance without any change. Then, we switch to a new kernel enabled with Fjord, and turn on write buffering for the application's working directory path (which is under `/data/data`[2]), and measure the changed application performance. All tests have been repeated 3 times, and the average numbers are reported in our results.

We compare performance for six different configurations: (a) the original storage (Native), (b) only logging buffer is enabled (Logging), (c) only RAM buffer is enabled (RAMBuf.), (d) Both RAM and logging buffers are enabled (Both), (e) Fjord selective buffering is used with RAM buffer (FjordRAM), and (f) Fjord selective buffering is used with both RAM and logging buffers (FjordBoth). Note that the first four configurations do not sacrifice reliability while the last two configurations (FjordRAM and FjordBoth) sacrifice reliability for chosen application files.

RL Benchmark: SQLite

This benchmark measures the database performance of Android system by running synthetic database queries [88]. According to the latest study about Android application performance [62], database is a key performance contributor, and thus, this benchmark is very useful to see the performance effects of our storage solution.

Figure 27 and Figure 28 show the reported runtime for the benchmark program

---

[2] For our controlled experiments, we do this write buffering control manually in the ADB shell.

on the Galaxy Note smartphone with the internal eMMC device and the external SD card, respectively. Figure 29 shows the result from Nexus One phone with the same external SD card, and Figure 30 compares only two configurations (Native, FjordBoth) for all three phone / storage configurations.

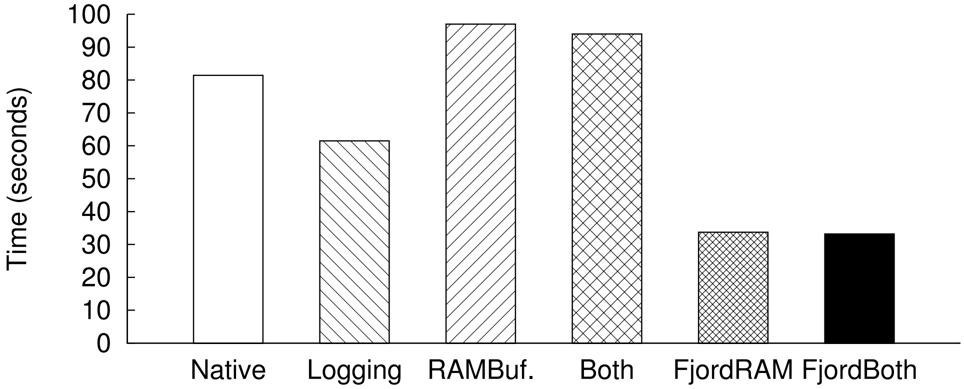


Figure 27: RL Benchmark: SQLite (Galaxy Note, eMMC, Android 4.0.4): *Fjord eliminates about 59% of execution time*

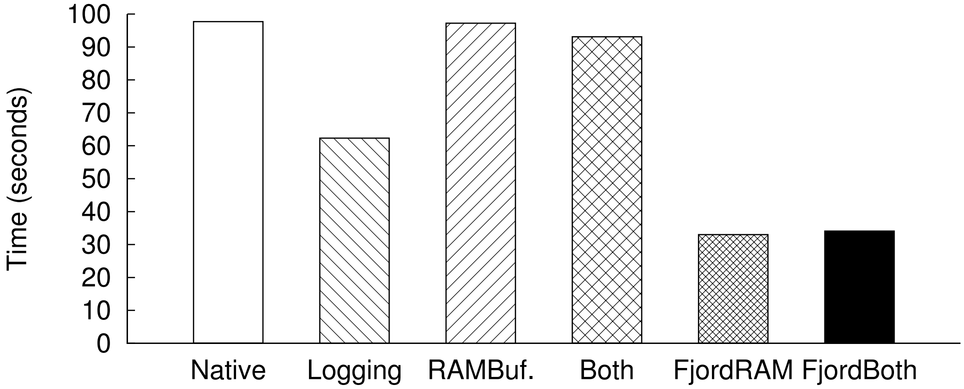


Figure 28: RL Benchmark: SQLite (Galaxy Note, SDCard, Android 4.0.4): *Fjord eliminates about 65% of execution time*

From the results, we can see that the RAM buffering with write barrier is not much helpful. It is expected because the EXT4 file system uses write barrier very frequently. Writes go to RAM buffer with some amount of overhead (buffer allocation, memory copy, etc), and almost immediately flushed to the flash storage because of the ensuing write barrier. Non-volatile logging looks very effective; it eliminates 24%,

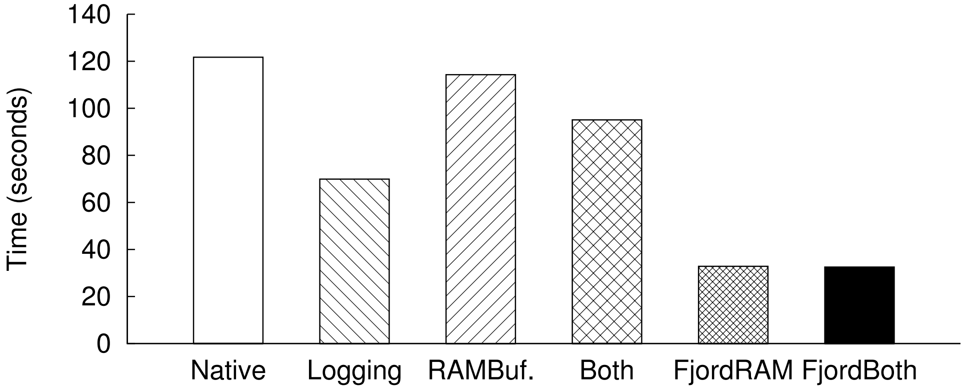


Figure 29: RL Benchmark: SQLite (Nexus One, SDCard, Android 2.3.7): *Fjord removes about 73% of execution time*

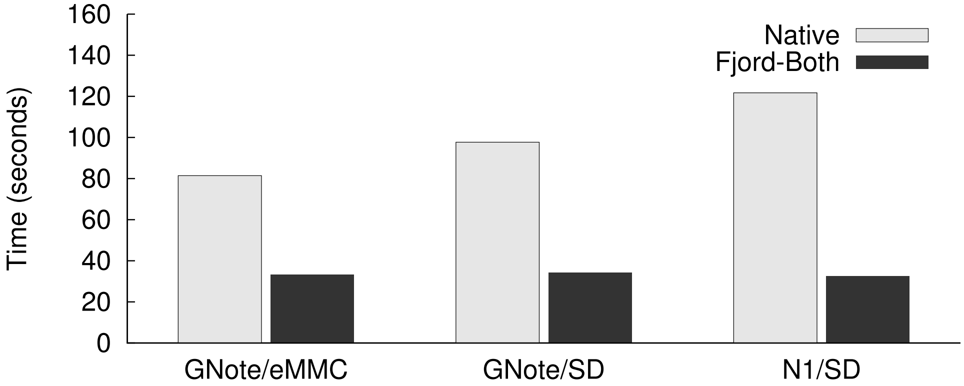


Figure 30: RL Benchmark: SQLite: *Summary for all three configurations*

36%, and 47% of execution time for the benchmark on tested three cases. However, these numbers are obtained when there is enough free space in the logging buffer, and the performance gain will be reduced as the logging buffer becomes full.

With Fjord, we can control write buffer more precisely. For cloud backed applications, the local dirty file content (which is really meta-data of the App for manipulating the real data objects in the cloud) is not that critical because the real content needed by the App (e.g., widgets displayed by a browser) are safely in the cloud, and can be easily recovered over the network. By allowing non-critical dirty data to stay in RAM buffer, it can be seen in Figures **??**, **??**, **??**, that the performance gain increases up to 59%, 65%, and 73%; it is a huge performance gain.

In addition, we see that Fjord removes the dependences of storage; Nexus One with SD card is about 50% slower than Galaxy Note with eMMC for this benchmark, and with Fjord, Nexus One becomes even slightly faster than the Galaxy Note phone.

Breakdown of Performance Contribution

Fjord consists of two major mechanisms: *fsync() control* and *write buffering layer*. Figure 30 shows the improved performance with both the mechanisms. Because we are curious about the performance contribution of each mechanism, we run the benchmark with only one mechanism separately. Figure 31 shows the measured results on Nexus One phone with the SD card because the obtained performance gain is the biggest on the phone.

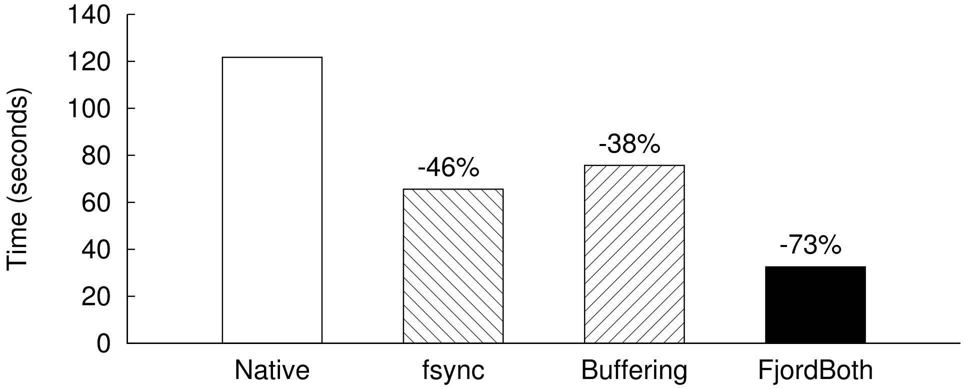


Figure 31: Reported runtime RL Benchmark: *the breakdown of the performance contribution*

Figure 31 shows the breakdown of the performance contribution. We also measured the amount of write traffic given to the block device, and Table 10 shows the measured amount write requests for each configuration. As can be seen from the table and graph, fsync() control reduces the write traffic more effectively, and thus its performance gain is more than with write buffering. Write buffering reduces about 25% of the write traffic, but its performance gain is much bigger, about 38%. This is because our write buffering layer additionally does write request reordering to be

Table 10: The amount of write requests collected while running RL Benchmark

Configuration	Amount of Write Traffic	Reduction Percent
Native	108,717 Kbytes	0
fsync control only	43,400 Kbytes	-60%
Write buffering only	81,751 Kbytes	-25%
FjordBoth	28,885 Kbytes	-73%

more favorable for flash storage internal characteristics. An important observation is that both the mechanisms of Fjord contribute to storage performance, and thus both are necessary.

Email

Email is one of the most popular applications on smartphones. In this benchmark, we measure the runtime for downloading emails after we input information of an email account.

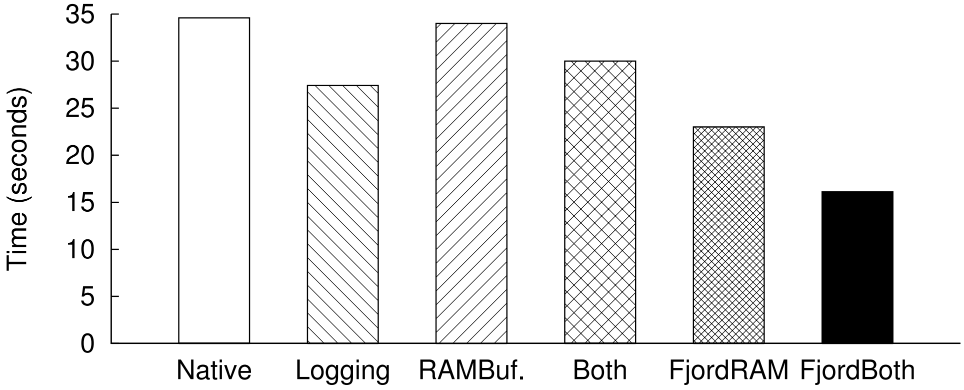


Figure 32: Email (Galaxy Note, eMMC, Android 4.0.4): *Fjord eliminates about 53% of execution time*

Figure 35 shows that Fjord significantly reduces the runtime by 88%, 47%, 58% on three configurations, respectively. The trend in Email test results is almost the same as observed for the RL Benchmark. RAM buffering with write barrier shows limitations, non-volatile logging is promising, and Fjord shows superior performance gains than other approaches. One very interesting result is observed on Galaxy Note

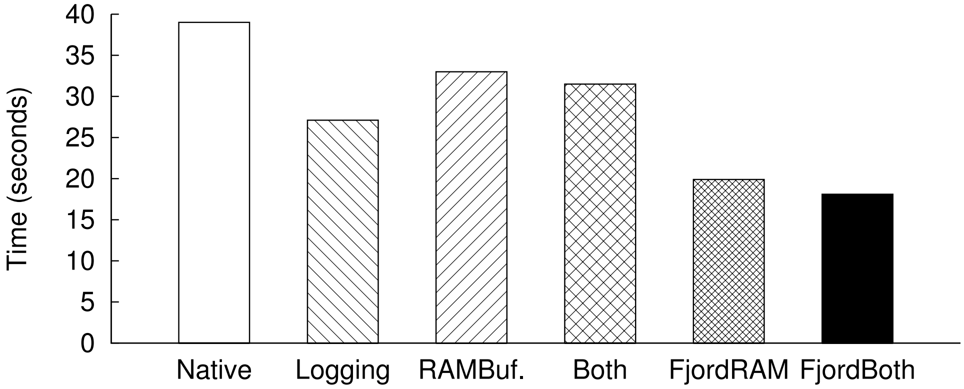


Figure 33: Email (Galaxy Note, SDCard, Android 4.0.4): *Fjord eliminates about 55% of execution time*

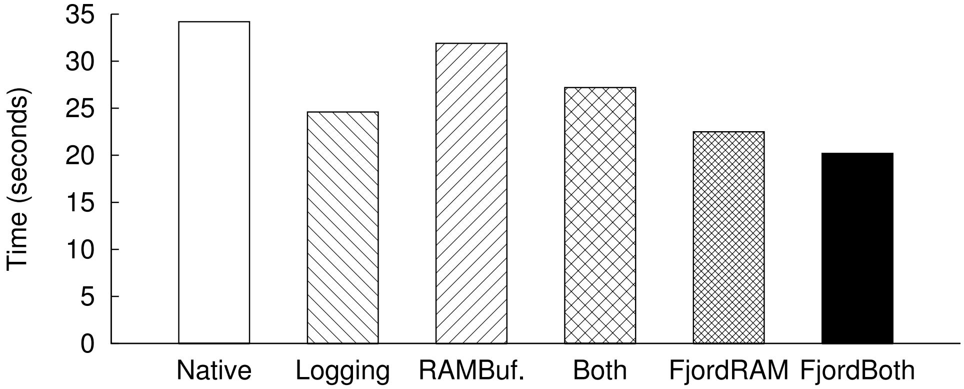


Figure 34: Email (Nexus One, SDCard, Android 2.3.7): *Fjord removes about 41% of execution time*

smartphone with the eMMC device. FjordBoth configuration shows about 20% bigger gain than FjordRAM configuration, which means all three mechanisms (logging, RAM buffering, fine-grained reliability control) are necessary and meaningful for achieving high performance.

Web Browsing To evaluate web-browsing performance, we build our custom benchmark program, which visits 20 pre-defined web sites continuously and reports elapsed time. Figure 36 - Figure 39 show the measured results, which are quite interesting; unlike other benchmark results, Fjord's performance gain is relatively small on the Android 4.0.4, Icecream sandwich phone. We measured this decrease performance

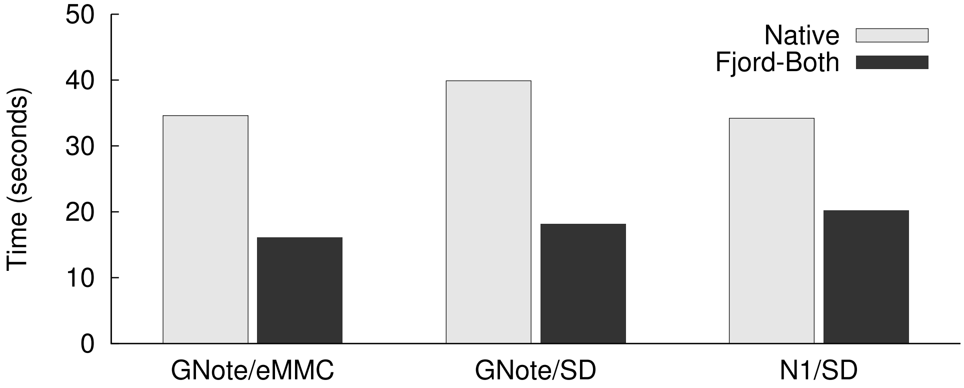


Figure 35: Email: *Summary for all three configurations*

benefit of Fjord with the newer Android Icecream Sandwich, but not with Android Gingerbread. We believe this is because the WebKit engine of Android 4.0.4 has been changed to resolve the performance bottleneck related with web caching that we reported in our earlier studies [62]. We confirmed our hypothesis by comparing the new source code of the WebKit engine of Android 4.0.4 with that of AOSP and found huge differences. However, the important message is that Fjord achieves almost the same performance gain with minimum effort and without changing one line of application source code. In other words, Fjord can enhance the performance of any legacy cloud-backed application without any change to the application source code.

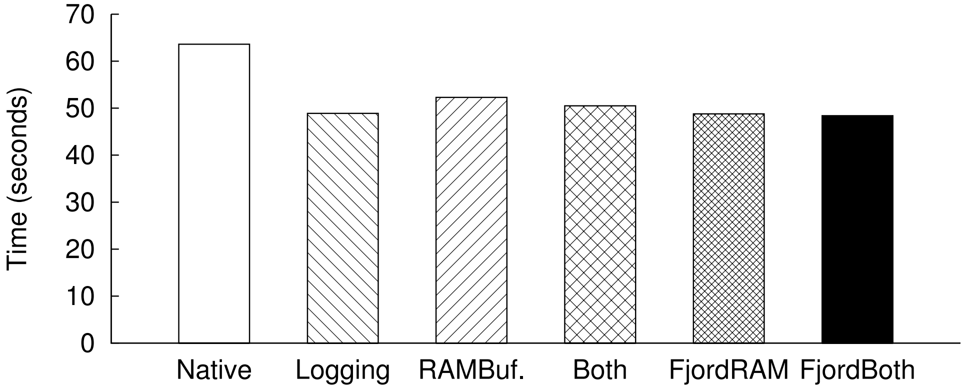


Figure 36: WebBench (Galaxy Note, eMMC, Android 4.0.4): *Fjord eliminates about 24% of execution time*

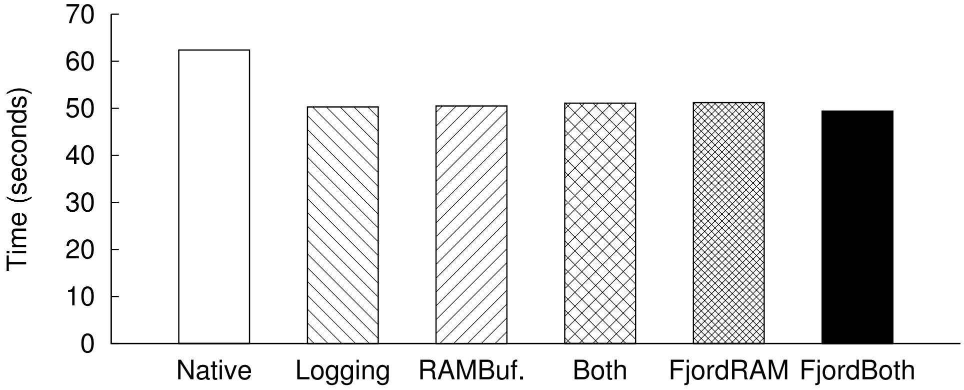


Figure 37: WebBench (Galaxy Note, SDCard, Android 4.0.4): *Fjord eliminates about 21% of execution time*

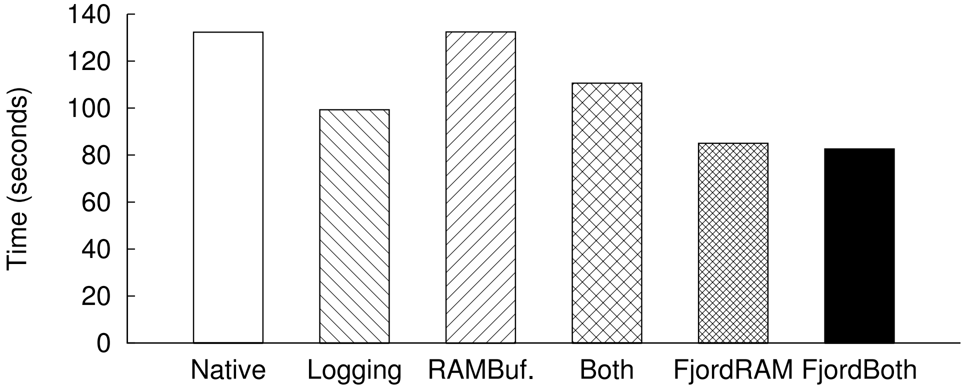


Figure 38: WebBench (Nexus One, SDCard, Android 2.3.7): *Fjord removes about 38% of execution time*

6.8 Summary

Due to size, power, and cost considerations, smartphones will continue to deploy low-end flash memories as the primary storage. Therefore, it is important to consider what can be done in the OS to enhance the performance of flash based storage systems. In this study, we propose Fjord, a fine-grained write buffering solution for mobile platforms. We show the effectiveness of our solution, and also provide explanations as to why Fjord is safe for mission critical applications. The solution is very simple, and thus practical, and we prove this fact by applying it to two real Android smartphones. Even though we are focusing on smartphone storage in this study, we believe that

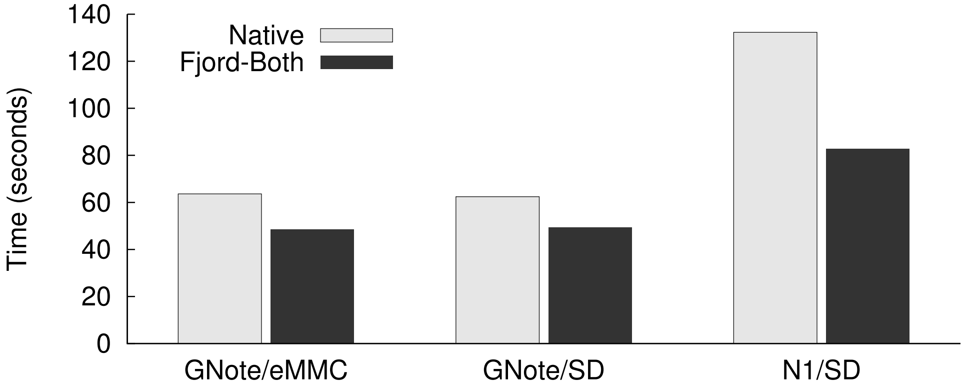


Figure 39: WebBench: *Summary for all three configurations*

this idea has potentials beyond smartphones for other types of storage and systems (see Chapter VII).

CHAPTER VII

DISCUSSION AND FUTURE DIRECTION

In this dissertation, we have presented storage solutions focusing on mobile platforms. In this chapter, we present several insights learned from our experiences.

7.1 Endurance of Research Results from this Dissertation

The thesis statement and the dissertation work were motivated by mobile platforms, but the research output from this dissertation has implications beyond mobile platforms. In this section, we distinguish which parts of the work are specific only to smartphones and which parts will endure beyond mobile platforms.

7.1.1 Smartphone Specific Findings

The performance observation study published as a research paper entitled "Revisiting Smartphone Storage", is mainly about smartphones. SpatialClock is also specific to smartphone storage because its key design features are motivated by the special performance characteristics of low-end flash storage devices, wherein write ordering is very important. In other words, such devices deliver much faster performance for sorted write requests are much fast than randomly scattered writes. Within today's regular sized SSDs, sophisticated FTL design is being used, and write performance is not that much influenced by write ordering; request size is still important, though.

The write buffering solutions of Fjord study is also tightly coupled with smartphone storage. RAM buffering and logging solutions have been designed to improve the random write performance of smartphone storage. Random writes are changed to sequential writes by using the log-structured non-volatile write buffer, and additional RAM buffering helps to efficiently utilize limited logging space.

7.1.2 Studies beyond Smartphones

Cloud computing and ubiquitous computing are popular today, and we propose Fjord as a fine-grained reliability control mechanism for the cloud-backed applications on a smartphone. Fjord approach can be useful beyond smartphones. Today, a user has multiple computing devices; smartphones, tablets, laptop, and desktop computers. Cloud storage enables users to access their data from any device conveniently. For instance, Apple's Photo Stream automatically uploads and distributes photos over all registered devices via iCloud. With the emerging spread of the cloud storage, the reliability requirement can be relaxed for some applications even in traditional laptop and desktop computers, and thus, Fjord can be useful also on regular laptop and desktop computers.

Besides, Fjord's fine-grained reliability control can be useful even without cloud storage. Applications generate many temporary files for some reasons. For example, database management systems (DBMS) create and use temporary files for join and sort operations. The contents of such files are not critically important, and losing the content for sudden power failure or system crash is not an issue; uncompleted operation will be safely rolled-back and the temporary files will be deleted anyway.

Fjord provides reliability relaxing mechanisms as a tool, and it can be useful as long as storage performance needs to be improved regardless of the system types.

Lastly, we have demonstrated how the system wide information can be used to control storage solutions. We design and implement an Android App, which is specifically based on usage pattern of smartphones. We believe that system wide information can be useful beyond smartphones. In our Beacon project [67], we have shown that application knowledge can be used for proactive data migration on a multi-tiered storage system composed of SSD and HDD. Information barriers are very common in today's systems; guest OSes on a virtualized system, application and storage servers, etc. Sharing information over the barriers holds significant promise for the future evolution of computer systems.

7.2 Boundary between Software and Hardware

One very interesting question is deciding who does what between the operating system software and flash storage device. Flash storage devices are becoming more and more powerful. A consumer level MLC SSD has 3-cored ARM 9 processor and 256MB on-board cache. Thus deciding the functional boundary between the host operating system and the flash storage is an interesting topic.

When storage devices become powerful and complicated, it is hard to predict the behavior of the device; for example, our prior work shows that cheaper low-end SSDs are more suitable for building a video server than expensive high-end SSDs are [92].

In addition, there is an information barrier between a storage device and the host system. The study by Arpaci-Dusseau et al. [25] highlights that information from the file system has the potential for significantly enhancing the storage performance. Similarly, our SpatialClock research shows that we can optimize write pattern for flash storage devices in the OS; with the right OS support (SpatialClock), we can achieve better performance than using a hardware solution (eMMC) to circumvent the performance issues of mobile flash storage.

In other words, findings from this dissertation work call to question the wisdom of increasing the hardware complexity of the flash storage when it is possible to achieve the same performance with the right OS support on the host side.

7.3 Future Work

In this section, we list the possible further studies, which could come as a natural outgrowth of this dissertation.

7.3.1 Fjord Implementation in Other Layers of the Software Stack

We have proposed and demonstrated the fine-grained reliability relaxing idea, that is the philosophy of Fjord, through this dissertation by implementing it as two mechanisms: fsync control and block device level write back buffering. We chose these approaches because we believed the approaches were easy to implement but effective. In addition, our Fjord mechanisms are applicable without any changes to the Android applications themselves. As a further study, we may apply the Fjord idea to the other layers of the storage software stack to find out which approach is the most desirable one. We may change SQLite or EXT4 file system APIs to distinguish critical data accesses from non-critical data accesses.

7.3.2 Energy Saving Considerations

Fjord distinguishes non-critical files from critical files and use RAM buffering aggressively only for non-critical files. By doing this, Fjord significantly reduces the number of I/O operations; that is, Fjord can save power by reducing I/O operations. As we all know, power consumption is critically important in mobile platforms like smartphones and tablets because these devices rely on limited battery power. Therefore, understanding the energy saving implications of the techniques proposed in this dissertation is an important direction of future research. Smartphones keep synchronizing local database files with the cloud content in background. We have observed that background sync process generates about 20 times bigger write traffic than the network receive traffic when the system is idle. Fjord reduces the amount of I/O operations, which has implications for the energy consumption for the actual storage operations as well as the wakeup times for the synchronization process between the smartphone and the cloud. An interesting avenue of future research is understanding the extent of energy savings due to Fjord.

7.3.3 Application Consistency Study

The key idea of Fjord is improving the performance of chosen applications without hurting the reliability of other applications. To this end, we control fsync() at file system level, and adopt additional write-back buffering layer at the block device layer. Fjord guarantees file system consistency by not applying Fjord mechanisms to file system metadata, but it achieves performance enhancement by sacrificing reliability guarantee for data files; some local file content may not be up to date or lost in case there is a sudden power failure. We have conducted preliminary experiments to understand the effect of such optimization on application behavior, and reported that there were no critical issues for many cloud-backed applications. More complete and elaborate study to understand the influence of the proposed mechanisms on cloud-backed application behavior is an important avenue for future research.

As a further step, a middleware approach can be taken to let application developers control file level reliability more gracefully. A pertinent research question in this context can be framed as "What is the desirable programming model for fine-grained reliability control?"

7.3.4 Emerging Storage

NAND flash memory is the only major player in mobile platforms today. In the near future, we will certainly have more diverse storage technologies such as Phase Change Memory (PCM). PCM is more scalable, and its read latencies are almost two orders of magnitude better than Flash memory. PCM does not have "big sized erase-before-write" issue, and its write endurance is about three orders of magnitude better than Flash memory. The performance, reliability, and functionality of a solid state storage system can be greatly enhanced with emerging storage.

There are three potential approaches for incorporating PCM into the storage hierarchy. The first approach is to replace all the Flash memory with PCM, and the second possibility is to use PCM together with Flash memory. We believe the second approach (adding a small amount of PCM to the Flash storage) is more practical considering the cost per gigabyte of PCM; it is highly unlikely that PCM will have comparable price to Flash memory in the near future. Lastly, PCM can be used in a more radical way, namely, to build a new *universal memory*, which plays the role of both the main memory and the storage.

In addition, we need to think about how to use system wide information for new storage devices as we have done in this dissertation. We first need to understand the characteristics of the new storage device, and then, study how the current storage software stack can be changed considering the special characteristics of the emerging storage. As a further step, the idea of Fjord and Informed Storage Management can be applied for the emerging storage devices.

7.3.5 ISM for Desktop and Server System

Even though the main focus of this thesis work is given to mobile platforms, the research outputs have potentials also for regular desktop and server systems. Thus, such an investigation offers new opportunities for future research. We may apply Fjord mechanisms to enterprise DBMS system to improve performance, or information based control idea to the regular desktop OS to enhance user experiences. If file systems or storage devices knowledge from the upper layers of the software stack, they will be able to optimize their functionality in a much better way. Investigating an information sharing abstraction that allows such cross-layer optimizations without affecting the modularity of the software stack is part of the proposed future work.

CHAPTER VIII

CONCLUSION

Storage devices are rapidly changing, and we need to adapt the OS storage software stack to keep up with the changes. Such a re-evaluation of the storage software stack is especially required for mobile platforms because they are relying on inexpensive flash storage devices having very different performance characteristics from the familiar hard disk.

In this thesis work, we first show the importance of storage in mobile platforms; contrary to conventional wisdom, we find evidence that storage is a significant contributor to application performance on mobile devices. Then, we explore the solution space for flash storage; user-level library for selective logging, host-side write buffering layer, and OS buffer replacement scheme for flash storage have been studied. Finally, we build an integrated solution for smartphone storage, named Fjord. In the Fjord study, we re-design logging and RAM buffering solutions for smartphones, and also propose fine-grained reliability control mechanisms. We prove that non-volatile logging can improve storage performance remarkably. Understanding the characteristics of cloud-backed applications and controlling the reliability constraint for chosen cloud-backed applications can achieve additional significant performance gain.

We have presented several insights learned from our experience in Chapter VII. The thesis statement and the dissertation work were motivated by mobile platforms, but some research outputs from this dissertation have implications beyond mobile platforms; Fjord's fine-grained reliability control and information based dynamic solution control can be useful beyond smartphones. Another very interesting question is who does what between the operating system software and flash storage devices.

Findings from this dissertation work call to question the wisdom of increasing the hardware complexity of the flash storage when it is possible to achieve the same performance with the right OS support on the host side.

We have also presented potential future works. Applying the idea of Fjord to the other layer of the storage software stack will be useful to find the most desirable approach. Power consumption related evaluation would also be very interesting considering the importance of battery power in mobile platforms. Storage has always been expected to be reliable. even though it is not always critically required for some applications (e.g., cloud-backed applications), and for some storage devices, it is not practically possible (e.g., Triple-Level Cell (TLC) NAND flash memory). Fjord is one approach on relaxing the reliability constraint of the storage; an interesting follow on work is investigation of application behavior under such relaxed reliability constraint. In addition, it will be very interesting to extend this study to include emerging storage devices like PCM. Finally, it will be possible to apply the ISM ideas to desktop and server systems as well.

There are always design tradeoffs involved in building software systems. Especially when it comes to system software, these tradeoffs have to be evaluated very carefully since the design choices at this level affect application performance and user experience. Design choices are always fraught with such tradeoffs. Usually, system design chooses a "sweet spot" that optimizes the solution for meeting certain requirements and/or assumptions about the environment (application behavior, device characteristics, etc.). Unfortunately, since the real world use cases are very dynamic and the technology landscape is continually evolving, often such assumptions may turn out to be incorrect; further, statically fixing the design based on certain requirements may force the design to be conservative. This argues for the need to adjust the solution dynamically based on the use case, the technological evolution, and the workload. We name the idea as *Informed Storage Management (ISM)*, and we aim at providing such a dynamic decision-making framework for system design, specifically targeted to storage systems.

We would like to conclude this dissertation with two salient observations. The first concerns developing a deep understanding of the components of the target system such as storage devices, storage interface protocols, OS storage software layers, and all the way up to the characteristics of the applications. Such an understanding is crucial to avoid the pitfalls of making superficial design decisions. The second observation concerns the importance of information. Due to modularity considerations in building complex software systems, it is natural to use levels of abstraction leading to barriers between these levels. Sharing information across such the barriers holds significant promise for the future evolution of computer systems.

REFERENCES

[1] "Android Debug Bridge (ADB)." `http://developer.android.com/guide/developing/tools/adb.html`.

[2] "Android Open Kang Project." `http://aokp.co/`.

[3] "Android Open Source Project." `http://source.android.com/index.html`.

[4] "Block I/O Layer Tracing: blktrace." `http://linux.die.net/man/8/blktrace`.

[5] "Busybox unix utilities." `http://www.busybox.net/about.html`.

[6] "Clockworkmod rom manager and recovery image." `http://www.koushikdutta.com/2010/02/clockwork-recovery-image.html`.

[7] "Compete ranking of top 50 web sites for february 2011 reveals familiar dip." `http://tinyurl.com/3ubxzbl`.

[8] "CrystalDiskMark Benchmark V3.0.1." `http://crystalmark.info/software/CrystalDiskMark/index-e.html`.

[9] "Cyanogenmod." `http://wiki.cyanogenmod.com/index.php?title=What_is_CyanogenMod`.

[10] "Documentation/filesystems/ext4.txt." `http://lwn.net/Articles/203915/`.

[11] "Google nexus one." `http://en.wikipedia.org/wiki/Nexus_One`.

[12] "Htc desire." `http://www.htc.com/www/product/desire/specification.html`.

[13] "HTC EVO Phone ." `http://www.htc.com/us/products/evo-sprint#tech-specs`. Retrieved in Sep 2011.

[14] "LG G2X P999 Phone." `http://www.lg.com/us/products/documents/LG-G2x-Datasheet.pdf`. Retrieved in Sep 2011.

[15] "MonkeyRunner for Android Developers." `http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html`.

[16] "SD Speed Class/UHS Speed Class." `https://www.sdcard.org/consumers/speed_class/`.

[17] "Starburst data2sd." `http://starburst.droidzone.in/`.

[18] "Unrevoked 3: Set your phone free." `http://unrevoked.com/recovery/`.

[19] "Usb reverse tethering setup for android 2.2." `http://blog.mycila.com/2010/06/reverse-usb-tethering-with-android-22.html`.

[20] "Motorola Webtop: Release Your Smartphone's True Potential." `http://www.motorola.com/Consumers/US-EN/Consumer-Product-and-Services/WEBTOP/Meet-WEBTOP`, 2011.

[21] Adelson-Velskii, G. and Landis, E. M., "An algorithm for the organization of information," in *Proc. of the USSR Academy of Sciences*, 1962.

[22] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R., "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical conf.*, 2008.

[23] Alphonso Labs, "Pulse News Reader." `https://market.android.com/details?id=com.alphonso.pulse&hl=en`.

[24] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V., "Fawn: a fast array of wimpy nodes," in *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 1–14, ACM, 2009.

[25] Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Bairavasundaram, L. N., Denehy, T. E., Popovici, F. I., Prabhakaran, V., and Sivathanu, M., "Semantically-smart disk systems: past, present, and future," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, pp. 29–35, Mar. 2006.

[26] Bensoussan, A., Clingen, C., and Daley, R. C., "The multics virtual memory: Concepts and design," *Communications of the ACM*, vol. 15, pp. 308–318, 1972.

[27] Birrell, A., Isard, M., Thacker, C., and Wobber, T., "A design for high-performance flash disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, 2007.

[28] Bovet, D. and Cesati, M., *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[29] "Cellphone penetration worldwide." URL: http://tecverse.com/design/cellphone-penetration-worldwide.html, 2010.

[30] Chang, L.-P., "On efficient wear leveling for large-scale flash-memory storage systems," in *SAC '07: Proc. of the 2007 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1126–1130, ACM, 2007.

[31] Chen, F., Koufaty, D. A., and Zhang, X., "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. of the 11th international joint conf. on Measurement and modeling of computer systems*, 2009.

[32] Choi, J., Noh, S. H., Min, S. L., and Cho, Y., "Towards application/file-level characterization of block references: a case for fine-grained buffer management," in *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '00, (New York, NY, USA), pp. 286–295, ACM, 2000.

[33] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A., "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 301–314, ACM, 2011.

[34] Cohen, B., "Incentives build robustness in bittorrent," in *P2PECON: 1st Workshop of Peer-to-Peer Systems*, 2003.

[35] Compellent, "Data progression: Automated Tiered Storage." `http://www.compellent.com/Products/Software/Automated-Tiered-Storage.aspx`.

[36] Debnath, B., Sengupta, S., and Li, J., "Flashstore: high throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, pp. 1414–1425, September 2010.

[37] Dirik, C. and Jacob, B., "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proc. of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 279–289, ACM, 2009.

[38] Dumitru, D., "Understanding Flash SSD Performance." Draft, `http://www.storagesearch.com/easyco-flashperformance-art.pdf`, 2007.

[39] Dunn, M. and Reddy, A. N., "A new i/o scheduler for solid state devices." `http://dropzone.tamu.edu/techpubs/2009/TAMU-ECE-2009-02.pdf`, 2009.

[40] EMC, "FAST: Fully Automated Storage Tiering." `http://www.emc.com/about/glossary/fast.htm`.

[41] EMC, "VFCache: A server Flash caching solution." `http://www.emc.com/storage/vfcache/vfcache.htm`.

[42] Ganger, G. R., McKusick, M. K., Soules, C. A. N., and Patt, Y. N., "Soft updates: a solution to the metadata update problem in file systems," *ACM Trans. Comput. Syst.*, vol. 18, pp. 127–153, May 2000.

[43] Ganger, G. R. and Patt, Y. N., "Metadata update performance in file systems," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, (Berkeley, CA, USA), USENIX Association, 1994.

[44] Gartner, "Gartner highlights key predictions for it organizations and users in 2010 and beyond." `http://www.gartner.com/it/page.jsp?id=1278413`.

[45] Glass, G. and Cao, P., "Adaptive page replacement based on memory reference behavior," in *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '97, (New York, NY, USA), pp. 115–126, ACM, 1997.

[46] Gundotra, V. and Barra, H., "Android: Momentum, Mobile and More at Google I/O." Keynote at Google I/O, May 2011.

[47] Halperin, D., Kandula, S., Padhye, J., Bahl, P., and Wetherall, D., "Augmenting data center networks with multi-gigabit wireless links," in *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, (New York, NY, USA), pp. 38–49, ACM, 2011.

[48] IBM, "IBM System Storage DS8000 Easy Tier." `http://www.redbooks.ibm.com/abstracts/redp4667.html`.

[49] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification." White Paper, `http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf`, 1998.

[50] International Data Corporation, "Worldwide Quarterly Mobile Phone Tracker." `http://www.idc.com/research/viewfactsheet.jsp?containerId=IDC_P8397`.

[51] "iperf network performance tool." http://sourceforge.net/projects/iperf.

[52] Jiang, S., Chen, F., and Zhang, X., "Clock-pro: an effective improvement of the clock replacement," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 35–35, USENIX Association, 2005.

[53] Jiang, S., Ding, X., Chen, F., Tan, E., and Zhang, X., "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proc. of the 4th USENIX conf. on File and Storage Technologies*, 2005.

[54] Jiang, S. and Zhang, X., "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. of the ACM SIGMETRICS International conf. on Measurement and Modeling of Computer Systems*, 2002.

[55] Jo, H., Kang, J.-U., Park, S.-Y., Kim, J.-S., and Lee, J., "FAB: flash-aware buffer management policy for portable media players," *Consumer Electronics, IEEE Transactions on*, vol. 52, no. 2, pp. 485–493, 2006.

[56] John, Breitkreuz, H., Monk, and Bjoern, "eMule." `http://sourceforge.net/projects/emule`.

[57] Johnson, T. and Shasha, D., "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), pp. 439–450, Morgan Kaufmann Publishers Inc., 1994.

[58] Jung, H., Sim, H., Sungmin, P., Kang, S., and Cha, J., "LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, pp. 1215–1223, 2008.

[59] Karagiannis, T., Broido, A., Brownlee, N., and Claffy, k., "Is p2p dying or just hiding?," in *In Proceeding of IEEE Globecom 2004*, (Dallas,), 2004.

[60] Kawaguchi, A., Nishioka, S., and Motoda, H., "A flash-memory based file system," in *USENIX Winter*, pp. 155–164, 1995.

[61] Kim, H., Agrawal, N., and Ungureanu, C., "Examining storage performance on mobile devices," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, MobiHeld '11, (New York, NY, USA), pp. 6:1–6:6, ACM, 2011.

[62] Kim, H., Agrawal, N., and Ungureanu, C., "Revisiting storage for smartphones," in *Proceedings of the 10th USENIX conference on File and stroage technologies*, FAST'12, (Berkeley, CA, USA), USENIX Association, 2012.

[63] Kim, H. and Ahn, S., "BPLRU: a buffer management scheme for improving random writes in flash storage," in *Proc. of the 6th USENIX conf. on File and Storage Technologies*, 2008.

[64] Kim, H. and Ramachandran, K., "FlashLite: A user-level library to enhance durability of ssd for p2p file sharing," in *Proc. of The 29th International conf. on Distributed Computing Systems*, 2009.

[65] Kim, H. and Ramachandran, U., "Flashfire: Overcoming the performance bottleneck of flash storage technology," Tech. Rep. GT-CS-10-20, `http://smartech.gatech.edu/handle/1853/36335`, School of Computer Science, College of Computing, Georgia Institute of Technology, 2010.

[66] Kim, H., Ryu, M., and Ramachandran, U., "What is a good buffer cache replacement scheme for mobile flash storage?," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, (New York, NY, USA), pp. 235–246, ACM, 2012.

[67] Kim, H., Seshadri, S., Xue, Y., Chiu, L., and Ramachandran, U., "Beacon: Guiding Data Placement with Application Knowledge in Multi-Tiered Enterprise Storage System." `http://static.usenix.org/publications/login/2011-02/openpdfs/OSDI10reports.pdf`.

[68] Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D., and Noh, S. H., "Disk schedulers for solid state drives," in *Proc. of the 7th ACM international conf. on Embedded software*, 2009.

[69] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S., "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2000.

[70] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., and Moriai, S., "The linux implementation of a log-structured file system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 102–107, 2006.

[71] Koukoumidis, E., Lymberopoulos, D., Strauss, K., Liu, J., and Burger, D., "Pocket cloudlets," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, (New York, NY, USA), pp. 171–184, ACM, 2011.

[72] Lee, S.-W., Moon, B., Park, C., Kim, J.-M., and Kim, S.-W., "A case for flash memory ssd in enterprise database applications," in *SIGMOD '08: Proc. of the 2008 ACM SIGMOD international conf. on Management of data*, (New York, NY, USA), pp. 1075–1086, ACM, 2008.

[73] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M., "Silt: a memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 1–13, ACM, 2011.

[74] Ltd, A. O., "Yaffs: A NAND-Flash Filesystem." `http://www.yaffs.net`.

[75] Lv, Y., Cui, B., He, B., and Chen, X., "Operation-aware buffer management in flash-based systems," in *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 13–24, ACM, 2011.

[76] M-Systems, "Two Technologies Compared: NOR vs. NAND." White Paper, 2003.

[77] Megiddo, N. and Modha, D. S., "ARC: a self-tuning, low overhead replacement cache," in *FAST '03: Proc. of the 2nd USENIX conf. on File and Storage Technologies*, (Berkeley, CA, USA), pp. 115–130, USENIX Association, 2003.

[78] Micron, "e-mmc." `http://www.micron.com/products/mcp/managed_nand.html`.

[79] Microsystems, S., "ZFS: the last word in file systems." `http://web.archive.org/web/20060428092023/http://www.sun.com/2004-0914/feature/`.

[80] "'Mobile is my soul': more about cell phones in the south of africa." `http://mfeldstein.com/mobile-is-my-soul-cell-phones-in-south-afric/`, 2010.

[81] Moshayedi, M. and Wilkison, P., "Enterprise ssds," *Queue*, vol. 6, pp. 32–39, July 2008.

[82] Park, C., Cheon, W., Lee, Y., Jung, M.-S., Cho, W., and Yoon, H., "A re-conf.gurable ftl (flash translation layer) architecture for nand flash based applications," in *RSP '07: Proc. of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, (Washington, DC, USA), pp. 202–208, IEEE Computer Society, 2007.

[83] Park, S.-Y., Jung, D., Kang, J.-U., Kim, J.-S., and Lee, J., "CFLRU: a replacement algorithm for flash memory," in *CASES '06: Proc. of the 2006 International conf. on Compilers, Architecture and Synthesis for Embedded Systems*, (New York, NY, USA), pp. 234–241, ACM, 2006.

[84] Park, S. and Shen, K., "Fios: A fair, efficient flash i/o scheduler," in *Proceedings of the 10th USENIX conference on File and stroage technologies*, FAST'12, (Berkeley, CA, USA), USENIX Association, 2012.

[85] Prabhakaran, V., Rodeheffer, T. L., and Zhou, L., "Transactional flash," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, (Berkeley, CA, USA), pp. 147–160, USENIX Association, 2008.

[86] Redhat, "Device-mapper resource page." `http://sources.redhat.com/dm`.

[87] Redhat, "JFFS2: The Journalling Flash File System, version 2." `http://sources.redhat.com/jffs2`.

[88] RedLicense Labs, "RL Benchmark: SQLite." `https://market.android.com/details?id=com.redlicense.benchmark.sqlite`.

[89] Richard Pentin (Summary), "Gartner's mobile predictions." `http://ifonlyblog.wordpress.com/2010/01/14/gartners-mobile-predictions/`.

[90] Rosenblum, M. and Ousterhout, J. K., "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[91] Russinovich, M., "DiskMon for Windows v2.01." `http://www.microsoft.com/technet/sysinternals/utilities/diskmon.mspx`, 2006.

[92] Ryu, M., Kim, H., and Ramachandran, U., "Impact of flash memory on video-on-demand storage: analysis of tradeoffs," in *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, (New York, NY, USA), pp. 175–186, ACM, 2011.

[93] Samsung, "Galaxy Note." `http://en.wikipedia.org/wiki/Samsung_Galaxy_Note`.

[94] Samsung, "GalaxyS II." `http://www.samsung.com/global/microsite/galaxys2`.

[95] Samsung, "GalaxyS III." `http://www.samsung.com/global/galaxys3/`.

[96] Samsung Corp., "Samsung ships industry's first multi-chip package with a pram chip for handsets." `http://tinyurl.com/4y9bsds`.

[97] SanDisk, "inand." `http://www.sandisk.com/business-solutions/embedded-products/inand`.

[98] Satyanarayanan, M., "Mobile computing: the next decade," in *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, MCS '10, (New York, NY, USA), pp. 5:1–5:6, ACM, 2010.

[99] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N., "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, pp. 14–23, October 2009.

[100] Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., and Stein, C. A., "Journaling versus soft updates: asynchronous meta-data protection in file systems," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 2000.

[101] Sharma, C., "Global Mobile Market Update." `http://www.chetansharma.com/GlobalMobileMarketUpdate2012.htm`.

[102] Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Semantically-smart disk systems," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, (Berkeley, CA, USA), pp. 73–88, USENIX Association, 2003.

[103] Smaragdakis, Y., Kaplan, S., and Wilson, P., "Eelru: Simple and effective adaptive page replacement," tech. rep., Austin, TX, USA, 1998.

[104] Soundararajan, G., Prabhakaran, V., Balakrishnan, M., and Wobber, T., "Extending ssd lifetimes with disk-based write caches," in *Proc. of the 8th USENIX conf. on File and storage technologies*, FAST'10, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2010.

[105] Srinivasan, M., "Flashcache : A Write Back Block Cache for Linux." `https://github.com/facebook/flashcache`.

[106] Ted Tso, "Android will be using ext4 starting with Gingerbread." `http://www.linuxfoundation.org/news-media/blogs/browse/2010/12/android-will-be-using-ext4-starting-gingerbread`, Dec. 2010.

[107] "Telecom subscription data as on 30th september 2010." `http://www.trai.gov.in/WriteReadData/trai/upload/PressReleases/777/PrfinalSeptember2dec10erdiv.pdf/`, 2010.

[108] tom's hardware, "Performance Charts Hard Drives and SSDs." `http://www.tomshardware.com/charts/hard-drives-and-ssds,3.html`.

[109] WG802.11 - Wireless LAN Working Group, "IEEE STANDARD 802.11n-2009." `http://standards.ieee.org/findstds/standard/802.11n-2009.html`.

[110] Zhang, Y., Arulraj, L. P., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "De-indirection for flash-based ssds with nameless writes," in *Proceedings of the 10th USENIX conference on File and stroage technologies*, FAST'12, (Berkeley, CA, USA), USENIX Association, 2012.