Reducing Smartphone Application Delay through Read/Write Isolation

David T. Nguyen[*], Gang Zhou[*], Guoliang Xing[†], Xin Qi[*], Zijiang Hao[*], Ge Peng[*], Qing Yang[*]
[*]College of William and Mary
McGlothlin-Street Hall 126
Williamsburg, VA 23185, USA
{dnguyen, gzhou, xqi, hebo, gpeng, qyang}@cs.wm.edu
[†]Michigan State University
3115 Engineering Building
East Lansing, MI 48824-1226, USA
glxing@cse.msu.edu

ABSTRACT

The smartphone has become an important part of our daily lives. However, the user experience is still far from being optimal. In particular, despite the rapid hardware upgrades, current smartphones often suffer various unpredictable delays during operation, e.g., when launching an app, leading to poor user experience. In this paper, we investigate the behavior of reads and writes in smartphones. We conduct the first large-scale measurement study on the Android I/O delay using the data collected from our Android application running on 2611 devices within nine months. Among other factors, we observe that reads experience up to 626% slowdown when blocked by concurrent writes for certain workloads. Additionally, we show the asymmetry of the slowdown of one I/O type due to another, and elaborate the speedup of concurrent I/Os over serial ones. We use this obtained knowledge to design and implement a system prototype called SmartIO that reduces the application delay by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized concurrency parameters. The system is implemented on the Android platform and evaluated extensively on several groups of popular applications. The results show that our system reduces launch delays by up to 37.8%, and run-time delays by up to 29.6%.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies; C.5.3 [**Computer System Implementation**]: Microcomputers-Portable devices

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Smartphone Application Performance; Flash Disk I/O Optimizations; Application Response Time; Application Launch

1. INTRODUCTION

The number of smartphones used worldwide increases each year. According to International Data Corporation, smartphone vendors shipped a total of 918.6 million smartphones in 2013, up 27.2% from the 722.4 million units shipped in 2012 [15]. With their increasing use, smartphone users tend to demand better performance. Moreover, smartphone users are increasingly using phones for work-related activities such as processing emails, reading documents, etc. A study by Forrester Research [9] found that one quarter of work devices were smartphones and tablets. Therefore, it is crucial to study application performance in smartphones. In particular, reducing the application delay can greatly improve user productivity. In addition, a recent analysis [43] indicates that most user interactions with smartphones are short. Specifically, 80% of the applications are used for less than two minutes. With such brief interactions, applications should be rapid and responsive. However, the same study reports that many apps incur significant delays (up to 10 seconds) during launch and run-time.

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

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

To address the second research question, we design and implement a system prototype called SmartIO on the Android platform. SmartIO measures optimal concurrency parameters for each type of I/O, and issues I/Os with the use of the obtained concurrency parameters. The system reduces the application delay by applying a set of I/O optimizations. Specifically, it assigns higher priority to reads, lower priority to writes, and groups the I/Os based on these priorities. The approach proves to have smaller performance improvement on launch delays of applications currently running in the background (warm launch). This is expected, since once an app is already in memory, its launch is much faster (on average by 65% based on our experiments). Because there is little I/O traffic going to the flash disk during warm launch, SmartIO reduces warm launch delays on average only by 6.8%. Our work focuses on reducing launch delays of applications currently *not* running in the background (cold launch).

Little work in the research community directly relates to ours. Kim et al. [29] present an analysis of storage performance on Android smartphones and external storage devices. Their discovery of a strong correlation between storage and application performance degradation serves as motivation for our work. Yan et al. [43] propose a system predicting application launch using context such as user location and temporal access patterns. Their system reduces perceived delay through application prelaunching. However, the proposed system does not address the issue of slow application launch from the root, but instead lessens its impact.

In summary, the contributions of our paper are as follows:

- First, through a large-scale measurement study based on the data collected from 2611 devices using an app we developed, we find that Android devices spend a significant portion of their CPU active time (up to 58%) waiting for storage I/Os to complete. This negatively affects the smartphone's overall application performance, and results in slow response time. Further investigation reveals that a read experiences up to 626% slowdown when blocked by a concurrent write. Additionally, the results indicate significant asymmetry in the slowdown of one I/O type due to another. While the slowdown ratio of a read is up to 6.15, the slowdown ratio of a write is only up to 1.6. Finally, we study the speedup of concurrent I/Os, and the results suggest that reads benefit more from concurrency.

- Second, we design and implement a system prototype called SmartIO that shortens the application delay by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized concurrency parameters.

- Third, we evaluate our system using 40 popular applications from four groups (games, streaming, miscellaneous, and sensing) and we show that SmartIO reduces launch delays by up to 37.8%, and run-time delays by up to 29.6%. Moreover, SmartIO also reduces power consumption by 6%.

The remainder of this paper is organized as follows. Section 2 presents the related work. In Section 3, we introduce the background of our work. Section 4 provides preliminary measurements and motivation. In Section 5, we present the system architecture of our solution to improve smartphone application performance, and Section 6 elaborates implementation details. Section 7 evaluates our implementation, and Section 8 provides discussion with future work. We conclude our work in Section 9.

2. RELATED WORK

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

Smartphone Storage. Kim et al. [29] present an analysis of storage performance on Android smartphones and external flash storage devices. Their discovery of a strong correlation between storage and application performance degradation serves as motivation for our work. We take one step further and investigate possible reasons of such performance degradation, and propose a system to reduce application response using smart I/O optimizations. Nguyen et al. [33] study the impact of the flash storage on smartphone energy efficiency, while the main focus of our paper is the application performance. Finally, Jeong et al. [28] propose novel journaling methods that, however, are not our focus. We use obtained knowledge from the study of I/O behaviors to design and implement a system that improves the response time by prioritizing reads over writes, and grouping them based on assigned priorities.

Smartphone Application Delay. Yan et al. [43] propose a system that predicts which apps are to be launched using the context such as user location and temporal access patterns. Their system then provides effective application prelaunching that reduces perceived delay. Parate et al. [36] propose another prediction algorithm to reduce the launch delay. Compared to the previous work, their approach does not require prior training or additional sensor context. However, mis-predictions of the proposed approaches will lead to significant memory and energy overhead. We address the problem of slow application launch by analyzing possible reasons of the slowdowns in the granularity of read and write I/Os. With this knowledge, we design a system that improves the response time by prioritizing reads over writes. This has a positive impact on the application performance beyond delay.

Linux I/O Schedulers. The default I/O scheduler since Linux kernel version 2.6 is the Complete Fair Queuing scheduler (CFQ) [17]. This scheduler has also been adopted as the default one in most Android smartphones, including the ones used in our experiments. However, not optimized for smartphone environments, CFQ may cause long application response time that is the main focus of our work. Other available I/O schedulers (Noop and Deadline [2]) are only used for specialized workloads.

Enterprise Solutions. Flash technology has been recognized in enterprise systems. This is mainly due to its technical merits highlighted in [16, 25], including low power consumption, compact size, and fast random access. This motivated researchers to propose I/O schedulers for flash memory based Solid State Drives in computer storage systems [26, 30, 31]. Inspired by these works, we study I/O characteristics of smartphones that have some differences, and require careful design considerations for optimal performance. For instance, while large block sizes dominate in conventional systems, small 4KB I/Os account for up to 65% of smartphone operations [32]. Our proposed solution is simple, and reduces application delays by up to 37.8%, while still being power efficient. Other enterprise solutions focus on fairness policies [37, 40]. SmartIO builds upon the default Linux I/O scheduler, and adds an additional priority level that preserves the original priorities. Further fairness optimization is beyond the scope of this work.

3. BACKGROUND

First, we introduce the background of our work. In particular, the kernel components on the I/O path are discussed, with the emphasis on the block layer and the flash disk that are directly related to our work. We illustrate the main kernel components affected by a block device operation on the I/O path in Figure 1. The figure is adapted from the literature [21].

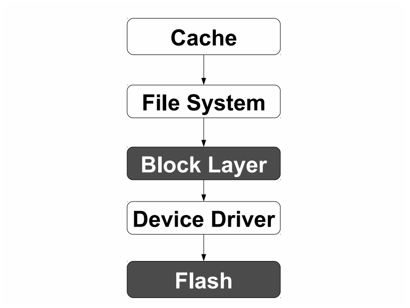


Figure 1: Kernel Components on the I/O Path.

3.1 Block Layer

At the block layer [8], the main work is scheduling I/O requests from above and sending them down to the device driver. The Linux kernels on recent Android smartphones offer 3 scheduling algorithms: Complete Fair Queuing (CFQ), Deadline, and Noop.

CFQ scheduler presented in [17] is the default I/O scheduler in Android smartphones. It attempts to distribute available I/O bandwidth equally among all I/O requests. There are two priority levels: one is the class, and the other is the priority within the class. There are three classes: real-time, best effort, and idle. Real-time class requests have the highest priority, followed by the best effort class whose disk access requests are granted only when there is no real-time request left. The idle class is given a disk access only when the disk is idle. Within the real-time and best effort classes, there are eight additional priorities [0(highest) to 7(lowest)]. Requests are placed into queues where each of the queues gets a time slice allocated. There are 8 queues in the real-time class, 8 queues in the best effort class, and 1 queue in the idle class.

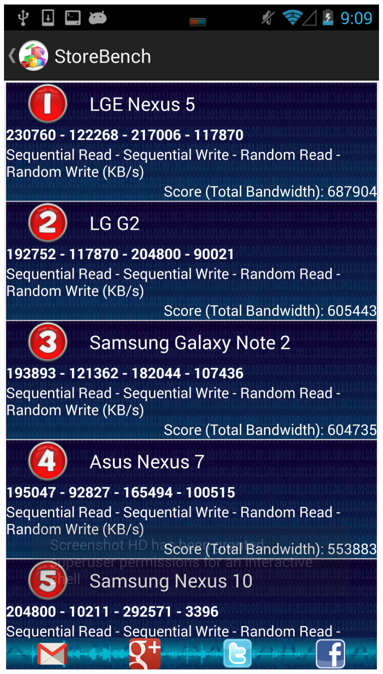
3.2 Flash Disk

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

Flash disk differs significantly from the conventional rotating storage. While rotating disks suffer from the seek time bottleneck, flash disks do not. Although providing superior performance compared to conventional storage, flash does have its own limitations. For instance, the erase-before-write limitation requires erase before overwriting a location. This leads to a substantial read/write speed discrepancy, which, among other issues, is discussed in the following subsections as a motivation for our work.

4. MEASUREMENT STUDY

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


9:09
StoreBench

1 LGE Nexus 5
230760 - 122268 - 217006 - 117870
Sequential Read - Sequential Write - Random Read - Random Write (KB/s)
Score (Total Bandwidth): 687904

2 LG G2
192752 - 117870 - 204800 - 90021
Sequential Read - Sequential Write - Random Read - Random Write (KB/s)
Score (Total Bandwidth): 605443

3 Samsung Galaxy Note 2
193893 - 121362 - 182044 - 107436
Sequential Read - Sequential Write - Random Read - Random Write (KB/s)
Score (Total Bandwidth): 604735

4 Asus Nexus 7
195047 - 92827 - 165494 - 100515
Sequential Read - Sequential Write - Random Read - Random Write (KB/s)
Score (Total Bandwidth): 553883

5 Samsung Nexus 10
204800 - 10211 - 292571 - 3396
Sequential Read - Sequential Write - Random Read -

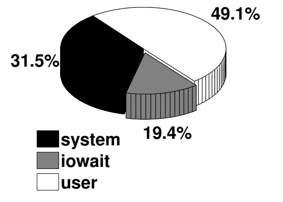
Figure 2: StoreBench Storage Benchmark.

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

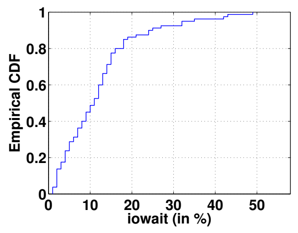
4.1 Measurement Setup

In a small-scale study, a Samsung S5 phone with Android 4.4.2 is utilized. The phone is normally used daily by the first author. During measurements, our Samsung S5 has all radio communication disabled, and the screen is off. Additionally, no app is in the foreground or background, and the cache is cleared before each measurement. To verify small-scale key observations, we design and implement a storage benchmarking tool called StoreBench [12] as an Android app, and make it available for free download on Google Play [11]. StoreBench is utilized to collect data for a large-scale study.

In the large-scale study, through StoreBench we obtain data from 2611 Android devices (complete list at [13]) that installed our benchmark from Google Play (97% of the devices run Android 4.0 or higher) in the period of nine months (November, 2013 - July, 2014). StoreBench tests the I/O performance of the internal flash storage and external SD card. Specifically, the tool measures the I/O bandwidth, response time, and CPU active time spent waiting for disk I/Os to complete (*iowait*). Additionally, it measures the launch and run-time delay of 20 popular apps. With the permission of users, results are submitted to our online database for further analysis and performance ranking. Our app anonymizes all data to maintain users' privacy. Note that we do not collect or derive any data from human subjects. Instead, we only collect technical information of the devices. Therefore, no IRB approval is required in our case. The dataset of the large-scale storage performance study will

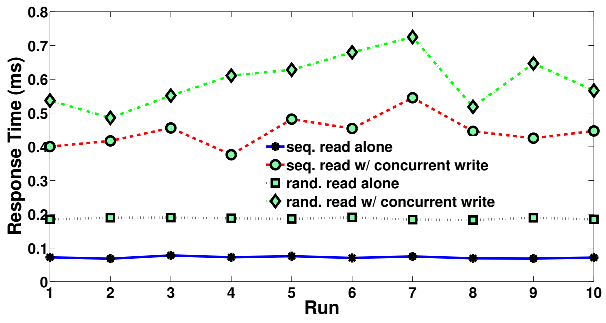


(a) Samsung S5 CPU Breakdown

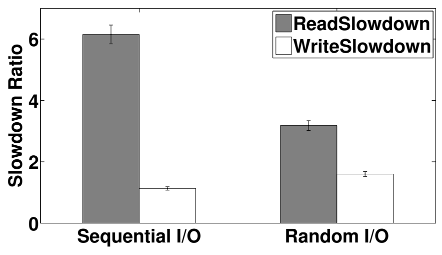


(b) Iowait of 2611 Devices

Figure 3: Iowait Values.



(a) Samsung S5 I/O Slowdown



(b) Samsung S5 Slowdown Asymmetry

Figure 4: I/O Slowdown.

be made available at [12]. StoreBench requires a rooted [10] device with Android 3.0 or higher, and installed BusyBox [1] on the device. The app's screenshot is in Figure 2.

4.2 Storage Contribution

To investigate what portion of the CPU active time is spent in storage, we use the *iostat* [4] shell command to output the I/O statistics of our Samsung S5 phone. The statistics from 30 days of use include detailed numbers of reads/writes of each block device in the flash disk. More importantly, the information includes the breakdown of the CPU active time spent in three domains:

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

The output of *iostat* for each domain is illustrated in Figure 3(a). The results show that a decent portion of time is spent in storage (19.4% of total active time), corresponding to 61.6% of system level time and 39.5% of user time. The output values observed are stable, and the standard deviation is as little as 0.1%. Note that the numbers are from the total use of all apps through the whole time period. Hence, some more I/O intensive apps can spend considerably longer than 19.4% waiting for disk I/Os to complete.

Since the measurements may be different from device to device, we also extract the *iowait* results from our large-scale study obtained through StoreBench to verify the pattern. The *iowait* empirical cumulative distribution function across 2611 Android devices is plotted in Figure 3(b). 40 percent of the devices have *iowait* values between 13% and 58%, which represents a significant portion of CPU active time. The averaged standard deviation is 0.1%. These results are also consistent with those of the Samsung S5.

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

4.3 I/O Slowdown

In the following experiment, the goal is to understand how one I/O type is slowed down due to another, in particular, how reads are slowed down by concurrent writes. For this purpose, we utilize the Linux flexible I/O tester named *fio* [18] to issue read and write I/Os from/to the Samsung S5 phone's internal flash disk. We port *fio* to Android OS, patch the modifications to the original *fio* code, and cross-compile it. We make *fio*'s binary available for interested readers at [3].

First, we want to measure the response times of reads when they are running *alone*. We start by sequentially reading a 128MB file (32768 read I/Os, each I/O size of 4KB), and calculating the average response time of a read I/O as the total response time divided by the number of I/Os. This is repeated for 10 runs. The average response time of a sequential *read when running alone is 0.072ms*, and standard variation is 2.3%. The choice of a 128MB file is to ensure that this workload is large enough to provide statistically significant measurements but at the same time does not overwhelm the phone's storage capacity. We use this size throughout the paper unless otherwise stated. The choice of the 4KB block size in our workloads is due to the fact that the default file system (Ext4) employed in recent Android devices utilizes this block size. Therefore, only 4KB is considered throughout this paper, even though it has been reported that large block sizes can improve performance [20]. Smartphone manufacturers use this small block size, since 4KB I/Os account for up to 65% of smartphone operations [32].

Next, we record the response times of reads in the presence of concurrent writes. We start by sequentially reading a 128MB file and concurrently writing a 256MB file (larger write size to assure there is concurrent write running when we read), and calculate the average response time of a read I/O. This is repeated for 10 runs. The average response time of a sequential *read in the presence of*

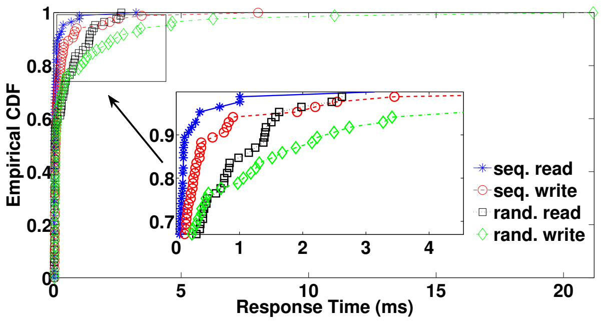


Figure 5: Response Time ECDF of 2611 Devices.

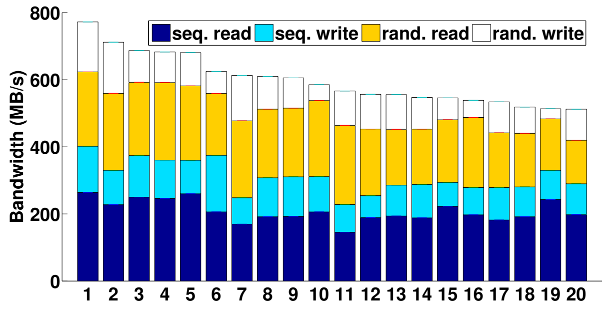


Figure 6: Storage Performance of Top 20 Models. **1:**LG Nexus 5; **2:**OnePlus One (A0001); **3:**Motorola Nexus 6; **4:**Bq Aquaris E10; **5:**Motorola Moto G; **6:**Samsung Galaxy Note 2 (GT-N7100); **7:**Sony Xperia Z Ultra (XL39h); **8:**Samsung Galaxy S3 (GT-I9300); **9:**LG G2 (LG-D800); **10:**Nubia Z7 Max (NX505J); **11:**Sony Xperia Z1 (C6903); **12:**Samsung Galaxy Note 3 (SM-N9002); **13:**Asus Nexus 7; **14:**Sony Xperia Z2 (D6503); **15:**LG L70 (LG-D321); **16:**Lenovo A328; **17:**Hisilicon Hi3798CV100; **18:**LG Optimus F6 (LGMS500); **19:**HTC One M8; **20:**LG G3 (LG-D850).

a concurrent write is 0.445ms, and standard variation is 3.1%. The two concurrent workloads are issued via *fio* as two separate processes. Buffers and caches are bypassed to obtain native properties. The above experiment is repeated for random I/Os. The average response time of a random *read when running alone is 0.187ms*, and standard variation is 3.3%. The average response time of a random *read in the presence of a concurrent write is 0.595ms*, and standard variation is 3.7%. The results of the two experiments are illustrated in Figure 4(a). There are a few observations from the figure. A sequential read experiences on average 515% slowdown (6.15 times slowdown) and up to 626% slowdown when blocked by a concurrent write. Similarly, a random read experiences on average 218% (3.18 times slowdown) and up to 293% slowdown when blocked by a concurrent write. This is important since it can be one of the main sources of slow application launch, when loading data is being blocked by a concurrent write. The root cause of the slowdowns is the flash read/write speed discrepancy (reads take much faster to complete). Additionally, reads become less predictable and the response times vary significantly over runs in the presence of a concurrent write.

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

4.4 Slowdown Asymmetry

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

- *ReadSlowdown = Response time of a read in the presence of a concurrent write / Response time of a read when running alone*
- *WriteSlowdown = Response time of a write in the presence of a concurrent read / Response time of a write when running alone*

The Samsung S5 results for both sequential and random I/Os with standard deviations are displayed in Figure 4(b). For sequential I/Os, while the slowdown ratio of a read is 6.15, the slowdown ratio of a write is only 1.13. For random I/Os, while the slowdown ratio of a read is 3.18, the slowdown ratio of a write is only 1.6.

This large asymmetry in the slowdowns has a following reason. Writes in the flash storage take already significantly longer than reads, hence, there is a smaller impact of the slowdown. While the response time of a sequential write running alone is on average 0.19ms, a sequential read running alone takes only 0.072ms. While the response time of a random write running alone is on average 0.41ms, a random read running alone takes only 0.187ms.

To understand the trend in the large scale, we plot the response time distributions obtained via StoreBench benchmark in Figure 5. In general, writes take longer than reads, and random I/Os take longer than sequential ones. This is consistent with the small-scale study using Samsung S5.

We also add Figure 6 with storage performance ranking obtained from the devices submitted by our users. Specifically, the figure includes the total bandwidth of the top 20 devices in MB/s. If a model has more devices in the ranking, then it is represented by its top device. An interesting observation is that a more recent model does not necessary mean higher ranking. For instance, while Nexus 5 (2013) tops the whole chart, Nexus 6 (2014) only occupies the 3rd place. Nexus 5 manufactured by LG mainly dominates thanks to its strong random write performance.

4.5 Concurrency

The next property that may affect I/O performance (and *iowait* as a result) is concurrency. An obvious approach to speeding up the application response is to issue I/Os concurrently. However, a large number of concurrent I/Os may overwhelm the processing capacity, and thus cause performance degradation. Therefore, it is necessary to find a sweet spot in concurrency to achieve maximal speedup. The last experiment's goal is to study the speedup of the concurrent I/Os over serial ones in the Samsung S5 phone. This is done for reads and writes separately. First, we issue two serial reads, each of size 32MB, and record the total response time. Then we issue two concurrent reads, each of size 32MB, and record the response time (use the larger result of the two reads if they differ). The speedup is calculated as the ratio of the two response times (serial / concurrent). This is repeated with four reads, eight reads, 16 reads,

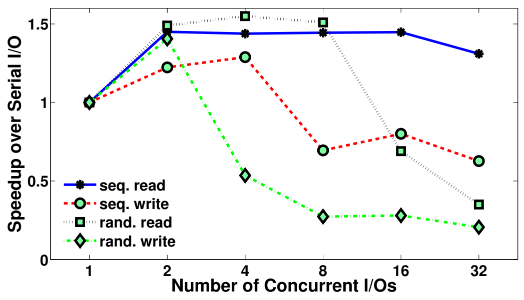


Figure 7: Samsung S5 Speedup over Serial I/O.

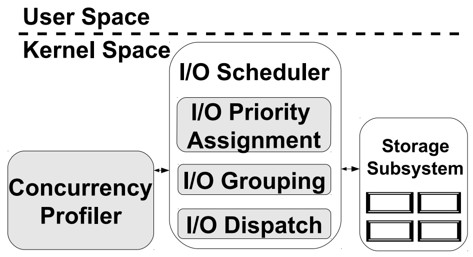


Figure 8: SmartIO.

and 32 reads, respectively. The choice of smaller workloads in this section (32MB) is because we issue up to 32 of such workloads concurrently, and do not want to overwhelm the phone's storage capacity.

To see how writes benefit from concurrency, we repeat the above with writes. First, two serial writes are issued, each of size 32MB, and the total response time is recorded. Then we issue two concurrent writes, each of size 32MB, and record the response time (use the larger result of the two writes if they differ). The speedup is calculated as the ratio of the two response times. This is again repeated with four writes, eight writes, 16 writes, and 32 writes, respectively. The speedup of concurrent I/Os over serial I/Os is illustrated in Figure 7.

We obtain four concurrency parameters from the figure. The number of concurrent sequential reads with maximal speedup (1.45) is 2, and the number of concurrent sequential writes with maximal speedup (1.29) is 4. The number of concurrent random reads with maximal speedup (1.55) is 4, and the number of concurrent random writes with maximal speedup (1.41) is 2. The speedup of reads is higher than the one of writes for both cases, which implies that reads benefit more from concurrency. This is expected. Intuitively, with growing processing time, the wait time also increases. Moreover, if the processing needs exceed the processing capacity, then there is no well-defined average waiting time because the queue can grow without bound. Since writes take longer to process than reads, it is expected that writes would overwhelm the processing capacity sooner, and thus benefit less from increased concurrency. In addition, different devices may benefit differently from concurrency, since they may have different speedup represented by concurrency parameters. Since these concurrency parameters may differ for various devices, a solution with the maximum benefits from concurrency requires a design that is capable of adapting to each phone's concurrency characteristics.

4.6 Summary

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

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

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

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

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

5. SYSTEM ARCHITECTURE

In order to improve the application delay performance in smartphones, we present SmartIO [35, 34], a system that reduces the application response time by prioritizing reads over writes, and grouping them based on assigned priorities. SmartIO issues I/Os with optimized concurrency parameters. The architecture of SmartIO is illustrated in Figure 8. It is fully located in the kernel space, and consists of two main modules: the I/O Scheduler and the Concurrency Profiler. The I/O Scheduler encapsulates 3 submodules: I/O Priority Assignment, I/O Grouping, and I/O Dispatch. We elaborate each module and its functionalities below.

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

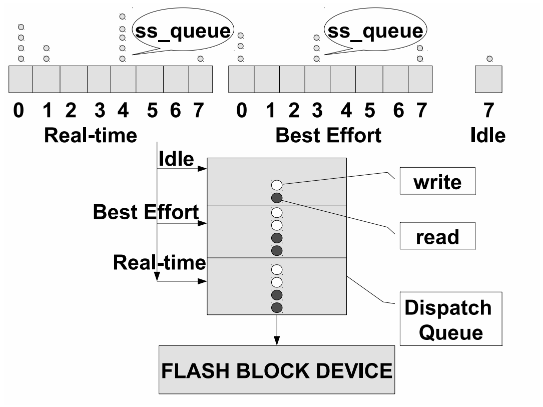


Figure 9: Dispatch Example.

maximal period of time assigned to a process, and is by default 100ms as used in the Linux scheduler time slice concept.

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

This reordering enforced by SmartIO does not affect correctness and semantics of write barriers. It is common knowledge that write barriers [38] are essential for consistency of many file systems. That is, however, maintained at the file system layer, which is above the I/O scheduler. Therefore, requests issued to an I/O scheduler can be reordered without affecting correctness. In fact, reordering is a common practice to minimize the seek costs in mechanical disks.

I/O Dispatch. A sample dispatch is illustrated in Figure 9. In the current CFQ implementation, each block device has 17 queues (ss_queue) of I/O requests (8 Real-time, 8 Best Effort, and 1 Idle). The existing system selects a queue based on the priorities, takes a request in the queue, and inserts it in the dispatch queue. The queue selection process accounts for two priority levels: the class priority (Real-time, Best Effort, Idle), and the priority within the class (0-7).

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

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

6. IMPLEMENTATION

In this section, we elaborate implementation details of the SmartIO system. In particular, we explain the algorithm of the scheduler's dispatch process. Next, we highlight important implementation challenges of the SmartIO system. Specifically, we discuss the I/O testing tool integration in the Concurrency Profiler module. The module utilizes the tool to obtain optimized concurrency parameters that allow SmartIO issue optimal number of I/Os concurrently to block devices.

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

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

Details of the dispatch are explained below. First, the system selects a queue from the 17 priority queues, then chooses a request in the selected queue, and inserts the request into the dispatch queue. If the time slice of the current queue is not expired (default 100ms as in CFQ), and the queue is not empty, the dispatch continues with the current queue. Otherwise, it chooses a different queue based on the priorities. The time slice serves as an ultimate mechanism to avoid starvation. When a queue q is chosen, the algorithm dispatches a request from it. If it may dispatch, it picks a request from the queue in the FIFO fashion, and inserts the request into the dispatch queue. The *dispatch* is elaborated in Algorithm 6.1.

Algorithm 6.1: DISPATCH($queue * q$)

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

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

To find out if we can dispatch from a queue q, *may_dispatch* is envoked. First, it checks whether the queue has more I/Os in flight than allowed. If not, it allows the dispatch. If the queue has already reached the dispatch limit, the system checks how many queues are waiting for dispatch. In case when there is another queue waiting, the dispatch is not allowed. If the queue is the only one, SmartIO sets no limit for it. The number of in-flight I/Os of a queue from the Linux default settings is 8. *may_dispatch* is elaborated in Algorithm 6.2.

Obtaining Concurrency Parameters. As discussed earlier, based on the concurrency parameters, SmartIO issues the appropriate num-

Algorithm 6.2: MAY_DISPATCH($queue * q$)

```
//does this $q$ already have too many I/Os in-flight?
if ($q.dispatched >= max\_dispatch$)
then
    if ($busy\_queues > 1$)
    then
        //we have other queues,don't allow more
        //I/Os from this one
        return ($false$)
    else if ($busy\_queues == 1$)
    then
        //sole queue user, no limit
        $max\_dispatch \leftarrow \infty$
    else
        $max\_dispatch \leftarrow quantum$
        //default init quantum is 8
//if we're below the current max, allow dispatch
return ($q.dispatched < max\_dispatch$)
```

ber of reads (writes) concurrently from the dispatch queue. To achieve this, SmartIO measures the concurrency parameters during installation by invoking the *fio* tool to benchmark the phone. *fio* issues reads and writes, and calculates the speedup of concurrent I/Os over serial ones, as performed in the measurement study. *fio* [18] is a Linux I/O testing tool that directs different types of I/Os to block devices, and returns information on the delay performance. The first step to get *fio* issue a desired workload is to write a job file. The typical contents of the job file is a global section defining shared parameters, and one or more job sections describing the jobs involved. For instance, the following code tests the sequential read and write performance of the /data partition on a phone:

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

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

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

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

To integrate *fio* in SmartIO, we patch the *fio* code with Android compiling adjustment, and cross-compile it to get its binary. We make the binary and job files available at [3]. The binary then is imported into the Concurrency Profiler module, and in run-time transferred to the /data partition directory in the internal flash disk.

7. PERFORMANCE EVALUATION

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

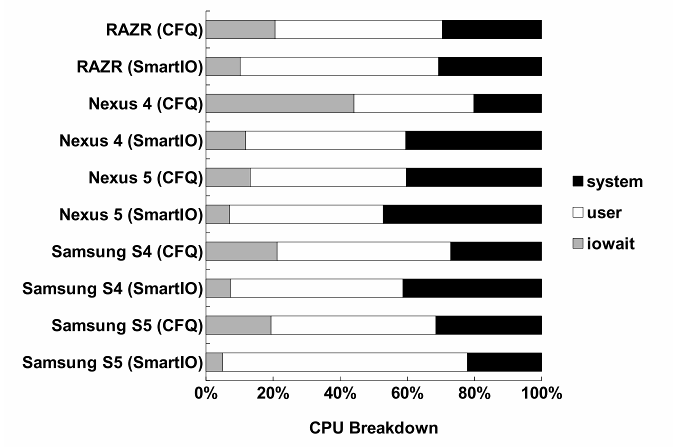


Figure 10: Iowait Before and After.

7.1 Iowait

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

7.2 Benchmark Performance

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

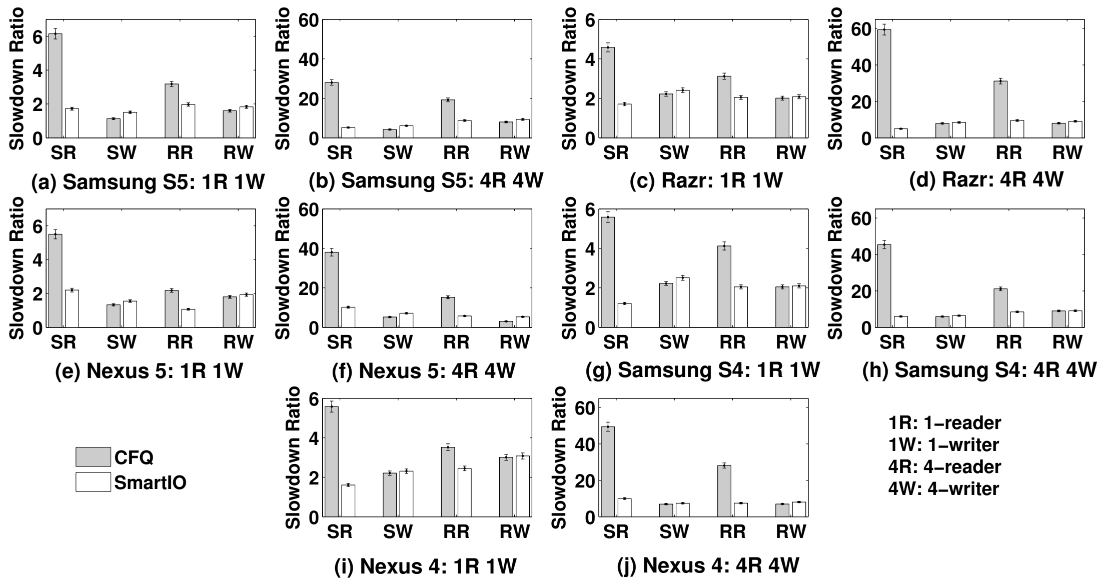


Figure 11: I/O Slowdown. SR=sequential read; SW=sequential write; RR=random read; RW=random write.

process. We consider both sequential and random I/Os. First, the experiment is done on the Samsung S5 phone. The I/Os are issued by the *fio* tool.

Gain vs. Cost. The I/O slowdown of the 1-reader and 1-writer with standard deviations is illustrated in Figure 11(a). For sequential I/Os, the read slowdown improves from 6.15 (CFQ) to 1.72 (SmartIO). Since our system delays writes in favor of reads, it is important to make sure that writes do not suffer a large performance degradation. As observed, this read performance improvement comes with only little cost due to the read/write discrepancy nature of the flash storage (reads take much faster to complete). Specifically, the write slowdown ratio worsens from 1.13 to 1.51. Similar behavior is observed for the random I/Os. While the read slowdown ratio improves significantly from 3.18 to 1.97, the write slowdown worsens slightly from 1.6 to 1.83. However, the random reads achieve smaller performance gain than the sequential ones. This is consistent with the results from the Measurement Study (Section 4), which show the random reads having lower slowdowns in the presence of the concurrent writes, hence, the benefit from the SmartIO read-preference scheduling is smaller.

The I/O slowdown of the 4-reader and 4-writer is illustrated in Figure 11(b). For sequential I/Os, the read slowdown ratio improves dramatically from 28.03 to 5.12. This large performance gain comes from the read-preference of SmartIO, together with the speedup from improved concurrency. The write slowdown ratio worsens from 4.21 to 6.12, which is the cost of SmartIO's lower write's priority. The random read slowdown improves from 19.22 to 8.75, while the write slowdown worsens from 8.01 to 9.32. Again, the random I/Os benefit from SmartIO slightly less than the sequential I/Os, which agrees with the theory.

Adaptation to Different Phones. As for validation, we also deploy our solution on other phones. First, we look at the Motorola Razr smartphone with the Android OS 4.0 (ICS), kernel 3.0, Ext4 file system, and duo-core. The Razr's default I/O scheduler is also CFQ, and its four concurrency parameters with maximal speedup found by SmartIO are: 2 concurrent seq. reads, 2 concurrent seq. writes (different from Samsung S5), 2 concurrent random reads (different from Samsung S5), and 2 concurrent random writes. The I/O slowdown of the 1-reader and 1-writer is illustrated in Figure 11(c). The I/O slowdown of the 4-reader and 4-writer is illustrated in Figure 11(d). Both figures are plotted with standard deviations. The 1-reader and 1-writer shows a similar behavior as on the Samsung S5 phone. The 4-reader and 4-writer indicates even larger read performance improvement compared to the Samsung S5 phone. The sequential read slowdown ratio improves from 59.4 to 4.98, while its write slowdown only worsens from 7.92 to 8.41. The random I/Os also show great improvement, the read slowdown improves from 31.12 to 9.5, while the write worsens from 8.01 to 9.08. This large performance boost is due to higher gains from concurrency, and demonstrates that SmartIO with its concurrency parameters measurement can adapt to different flash characteristics. The Samsung S5's smaller performance gain is due to the fact that the phone is more recent, and its four cores already offer great baseline performance. While the Razr's duo-core architecture shows even larger read performance improvement due to the lower baseline performance of the smaller number of cores. For comparison, we also display further results on the rest of the devices: Nexus 5 in Figure 11(e)(f), Samsung S4 in Figure 11(g)(h), and Nexus 4 in Figure 11(i)(j). They all demonstrate significant reductions in the read slowdown, while the write slowdown only worsens little. From these three devices, Samsung S4 has the largest read slowdown reduction (7.6 times in (h)), while Nexus 5 has the largest write slowdown increment (1.77 times in (f)). In summary, the above benchmarking experiments show different performance gains for a diverse set of devices. This is reasonable, since each device is equipped with different hardware components, and hence different results are expected. However, the experiments also confirm that SmartIO is able to adapt to different phones.

7.3 Scheduler Comparison

This section aims to compare SmartIO with other existing I/O schedulers: Complete Fair Queuing (CFQ), Deadline, and Noop. These are the only three schedulers available on recent Android de-

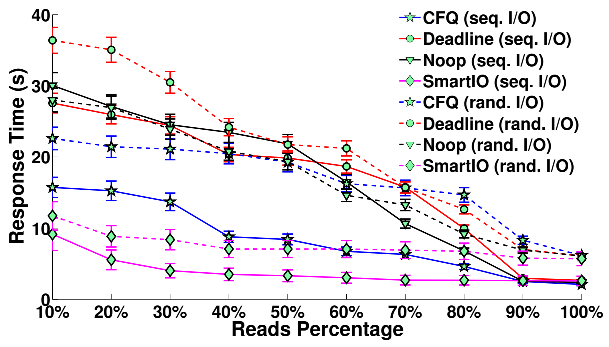


Figure 12: Scheduler Comparison. Solid lines are sequential I/Os; dashed lines are random I/Os.

vices. CFQ attempts to distribute available I/O bandwidth equally among all I/O requests. The requests are placed into per-process queues where each of the queues gets a time slice allocated. Further details on CFQ are explained earlier in the Background section. Deadline algorithm attempts to guarantee a start time for a process. The queues are sorted by expiration time of processes. Noop inserts incoming I/Os into a FIFO fashion queue and implements request merging.

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

Sequential I/O. For each scheduler we issue a 128MB mixed workload with 10% of sequential reads (90% of sequential writes), and record the response time. Next, we issue a 128MB mixed workload with 20% of reads (80% of writes), and record the response time. We continue issuing a workload with 30% reads, 40% reads, etc. Until the workload with 100% reads. The block size is set to 4KB, the queue depth to 128, and the cache is cleared after each measurement.

The resulting response times are plotted in Figure 12 *(solid lines)*. In general, for all four schedulers, with the increased percentage of reads, the response time decreases. For instance, with a workload consisting 10% reads, the response time for SmartIO is 9 seconds, CFQ 16 seconds, Deadline 28 seconds, and Noop 30 seconds. With 50% of reads, the response time is faster, SmartIO needs 3 seconds, CFQ 8 seconds, Deadline 20 seconds, and Noop 22 seconds. This is consistent with our measurement study, since reads are faster to complete, and less writes also means smaller I/O slowdown. For most workloads, SmartIO provides the fastest response time, while the current I/O scheduler in Samsung S5 (CFQ) is second best. Deadline and Noop perform poorly, and one beats another depending on the workload. Consequently, by changing the scheduler from the default CFQ to the proposed SmartIO, we achieve on average 42% faster response times (max of 64%).

Random I/O. The above experiment is reiterated for random I/Os. The resulting response times are plotted in Figure 12 *(dashed lines)*. Again, it is safe to say that with the increased percentage of reads, the response time decreases for all schedulers. This is consistent with our experimental study, since reads are faster to complete, and less writes also means smaller I/O slowdown. For all random I/O workloads, SmartIO has fastest response times. As a result, by changing the scheduler from the default CFQ to the proposed SmartIO, we may achieve on average 49% faster response times (max of 66%). Compared to sequential I/Os, random I/Os take longer to complete. This is also consistent with our findings in the measurement study, which identifies that random activities generally take longer to complete.

7.4 Application Performance

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

Launch Delay. The Android Monkey tool [6] is utilized to trigger the launch process of each app. The application *launch delay* starts when the launch process is triggered, and ends when the process completes. The launch delay includes three components. We use the *time* command [14] to output the three time components: the time taken by the app in the user mode (*user*), the time taken by the app in the kernel mode (*system*), and the time the app spends waiting for the disk and network I/Os to complete (*totalIO*). The storage I/O delay is obtained by dividing the total number of I/Os completed *(kBread + kBwrtn)* over the total rate of I/Os completed *(kBreadRate + kBwrtnRate)* in a flash block device. The network I/O delay is then calculated as the total I/O delay *(totalIO)* subtracted by the storage I/O delay *(storageIOdelay)*.

Formally,

$$storageIOdelay = \frac{kBread + kBwrtn}{kBreadRate + kBwrtnRate}, \quad (1)$$

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

$$networkIOdelay = totalIO - storageIOdelay, \quad (2)$$

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

The *cold launch delay* is a launch delay required to launch an application not currently running in the background. Such application also has its cache cleared before each measurement. The cold launch delay of the 40 apps with and without SmartIO is illustrated in Figure 13(a). The figure includes 10 games (1-5, 21-25), 10 streaming apps (6-10, 26-30), 10 miscellaneous apps (11-15, 31-35), and 10 sensing apps (16-20, 36-40). Applications running with

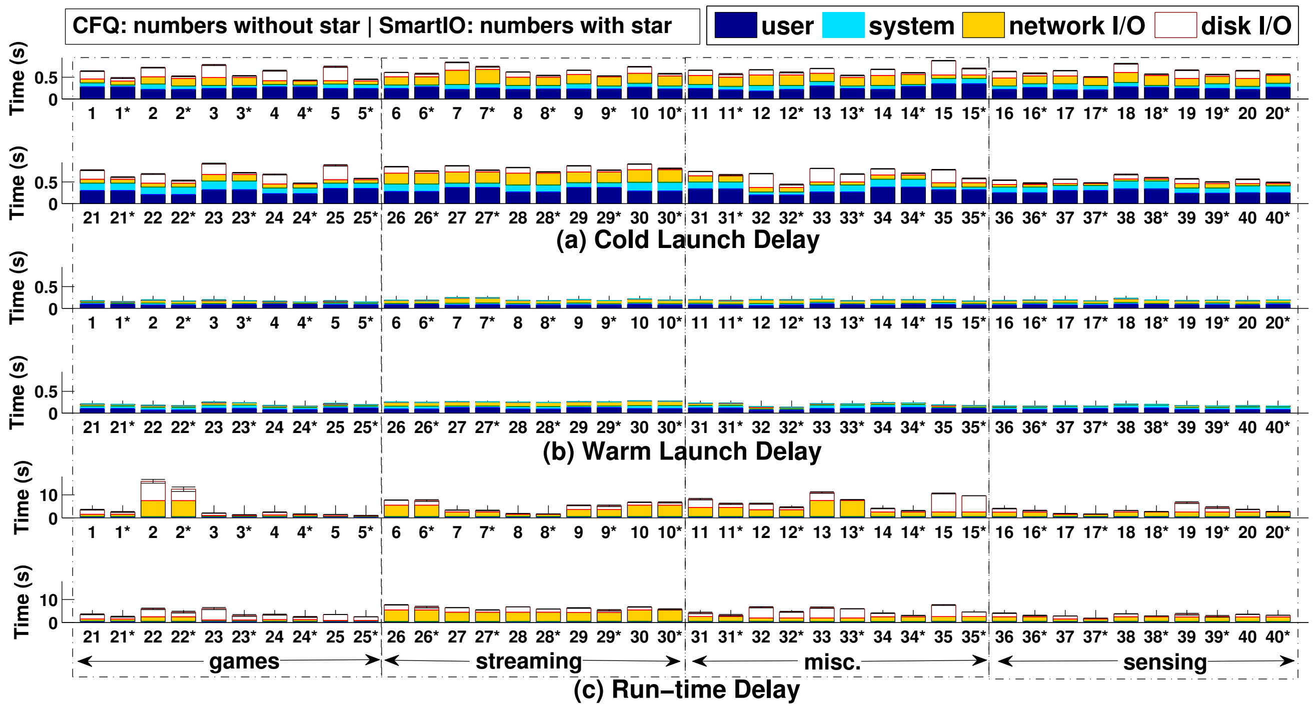


Figure 13: Launch and Run-time Delay. **1:**Angry Birds; **2:**GTA; **3:**Need for Speed; **4:**Temple Run; **5:**The Simpsons; **6:**CNN; **7:**Nightly News; **8:**ABC News; **9:**YouTube; **10:**Pandora; **11:**Facebook; **12:**Twitter; **13:**Gmail; **14:**Google Maps; **15:**ZArchiver; **16:**Accelerometer M.; **17:**Gyroscope Log; **18:**Proximity Sensor; **19:**Compass; **20:**Barometer; **21:**2048 Puzzle; **22:**Pet Rescue Saga; **23:**Pou; **24:**Solitaire; **25:**Words; **26:**CT 24; **27:**Live Extra; **28:**VEVO; **29:**VOYO.cz; **30:**WATCH ABC; **31:**Instagram; **32:**File Commander; **33:**RAR for Android; **34:**Dropbox; **35:**File Manager; **36:**Physics Toolbox; **37:**Sensor Kinetics; **38:**Android Sensor Box; **39:**Sensor Music Player; **40:**Sensor Mouse.

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

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

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

The run-time delay of the 40 apps with and without SmartIO is illustrated in Figure 13(c). The figure is plotted with standard deviations. The reduction in run-time delay with SmartIO ranges from 2% (Pandora) to 29.6% (Angry Birds) as compared to run-time delay without SmartIO. The run-time delay with SmartIO enabled for all the 40 apps is on average 16.9% smaller than with SmartIO disabled. Clearly, the run-time delays do not benefit from using SmartIO as much as the application launch. This is reasonable, since the application launch is more I/O intensive than the application run-time. For the 40 apps, the average number of I/Os during launch is 2 times higher than during run-time. While the run-time delay of the games with SmartIO is on average 23% smaller, the streaming apps have on average only 4% smaller run-time delay. This is

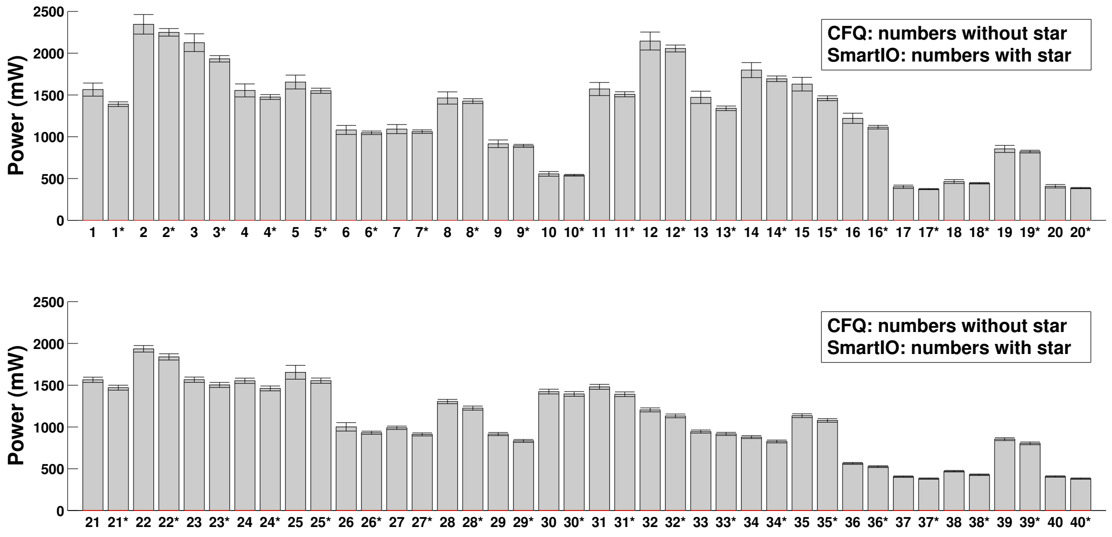


Figure 14: Power Consumption. **1:**Angry Birds; **2:**GTA; **3:**Need for Speed; **4:**Temple Run; **5:**The Simpsons; **6:**CNN; **7:**Nightly News; **8:**ABC News; **9:**YouTube; **10:**Pandora; **11:**Facebook; **12:**Twitter; **13:**Gmail; **14:**Google Maps; **15:**ZArchiver; **16:**Accelerometer M.; **17:**Gyroscope Log; **18:**Proximity Sensor; **19:**Compass; **20:**Barometer; **21:**2048 Puzzle; **22:**Pet Rescue Saga; **23:**Pou; **24:**Solitaire; **25:**Words; **26:**CT 24; **27:**Live Extra; **28:**VEVO; **29:**VOYO.cz; **30:**WATCH ABC; **31:**Instagram; **32:**File Commander; **33:**RAR for Android; **34:**Dropbox; **35:**File Manager; **36:**Physics Toolbox; **37:**Sensor Kinetics; **38:**Android Sensor Box; **39:**Sensor Music Player; **40:**Sensor Mouse.

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

Power Consumption. While improving the application performance is important, having solid power efficiency is equally important. To measure power consumption, the Monsoon Power Monitor [7] is utilized. Each of the 40 apps is run with SmartIO disabled, and then enabled. The Android Monkey tool triggers the launch process of each app, and then generates the same stream of 500 user events as previously. The results with standard deviations are presented in Figure 14. The average power consumption with SmartIO enabled is lower than the consumption with SmartIO disabled by 6%. Hence, our solution does not have energy overhead, and even contributes to lower power levels. We attribute this to the read-preference approach of the system that essentially allows shorter jobs to be completed first, which contributes to smaller application delay and consequently also lower power consumption.

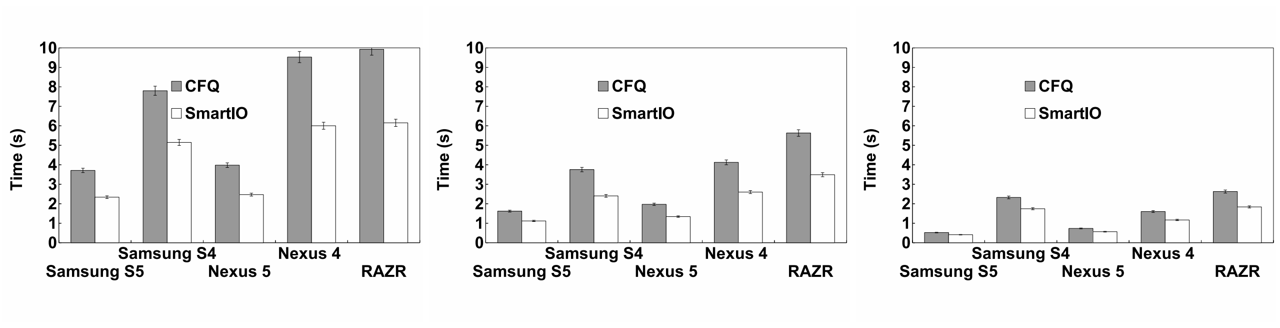
7.5 User-Perceived Performance: Facebook

In this subsection we conduct an experiment on the Facebook application to determine the user-perceived performance improvement of our solution. Since the delays in Figure 13 are obtained in the OS layer, the values are precise but significantly smaller than if obtained in the application layer. In order to acquire measurements in the application layer, we may use a stop watch, which is however inaccurate. Instead, we choose to slightly modify the Facebook source code[1] to record timestamps of several performance parameters. Specifically, we focus on three metrics that are critical to Facebook users: cold launch, warm launch, and timeline loading. A short demo of a modified Facebook version is available at [5]. The app uses test accounts and automates 150 measurements per metric without necessity of any user interaction. The experiment is conducted on the five phones listed above.

Cold Launch. Cold launch in Facebook is defined as the time required to complete loading all components of the start activity and rendering of the News Feed. All cache data is cleared except the login information. The ultimate goal of Facebook Inc. for the following years is to have cold launch of less than 5 seconds on devices released in 2012 or newer, and less than 10 seconds on older devices. The results in Figure 15(a) show that cold launch on our oldest device RAZR (2012) takes 9.9 seconds with CFQ and 6.2 seconds with SmartIO. The newest phone Samsung S5 (2014) spends 3.7 seconds on cold launch with CFQ, and 2.3 seconds with SmartIO. Finally, cold launch with CFQ on Nexus 5 (2013), Nexus 4 (2012), and Samsung S4 (2013) requires 4 seconds, 9.5 seconds, and 7.8 seconds, respectively. While with SmartIO, the three devices need 2.5 seconds, 6 seconds, and 5.1 seconds, respectively. Since the shortest human perceivable delay is 100ms [22], we can conclude that SmartIO can contribute significantly to reducing the user-perceivable cold launch delay.

Warm Launch. Warm launch is defined similarly as cold launch, except the cache is not cleared before each measurement. Figure 15(b) indicates that RAZR has the most noticeable reduction in the delay. Specifically, warm launch with CFQ takes 5.6 seconds, while with SmartIO it takes 3.5 seconds. Nexus 4's warm launch delay is reduced from 4.1 seconds to 2.6 seconds. Samsung S4 shows a reduction from 3.8 seconds to 2.4 seconds. Finally, the

[1]The first author interned with Facebook Inc.



(a) Cold Launch (b) Warm Launch (c) Timeline Loading

Figure 15: User-Perceived Performance of Facebook

newest devices Samsung S5 and Nexus 5 get their delays reduced from 1.6 second to 1.1 second, and from 2 seconds to 1.3 second, respectively.

Timeline Loading. Timeline is a user profile page. Its loading is defined as the time required to complete loading and rendering of all components in the profile activity, where the origin activity is the News Feed. This can be seen as switching from the News Feed to the Timeline page. The results in Figure 15(c) show less noticeable reductions in the delay. This is reasonable, since this timeline loading corresponds to run-time delays in Figure 13, where the I/O traffic is usually less intensive. RAZR and Samsung S4 show most significant delay reductions: from 2.6 seconds to 1.8 second, and from 2.3 seconds to 1.7 second, respectively.

8. DISCUSSION AND FUTURE WORK

Launch and run-time delays are critical to user experience, since one launches and runs apps repeatedly throughout the day. Therefore, we focus on launch and run-time delays. However, in future work we plan to evaluate the impact of other stages of the life cycle on application performance such as install, update, switch, and uninstall, and quantify their effects on everyday phone usage. We intend to extend this study by researching how other common usage patterns are impacted. For instance, taking photos, recording movies, messaging, calling, email sync (recently studied in [41]), etc.

As discussed earlier, one of the main reasons causing longer launch delay is the disk I/O performance, specifically read I/O performance. This is due to the read-intensive nature of application launch. The average number of reads observed during launch on the 40 popular apps in our experiment is five times higher than writes. Other factors may also play a role in the high variation of launch delays. In particular, the launch delay also depends on the app's physical location, i.e., whether on the internal flash or external SD card. According to our analysis, the application size is not a big contributor to the launch delay. While the three largest apps Angry Birds (42.4MB), The Simpsons (41.7MB), and Temple Run 2 (36.7MB) have the launch delay around 0.65s, the smallest app Proximity Sensor (0.02MB) has the fifth largest launch delay (0.8s). Finally, we plan to analyze the impact of network I/O based on existing results [42, 19, 23, 24, 27, 39, 44].

Our work only focuses on reducing the application delay with respect to the internal flash storage. It may be also interesting to study how different applications use SD cards. Kim et al. [29] already performed a series of benchmarking experiments on SD cards from multiple speed classes. However, it will be useful to go beyond benchmarking and investigate I/O access patterns on these devices. This especially can benefit multimedia applications that store data on the external storage.

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

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

Finally, the observations made in our measurement study are based on data obtained in the Samsung S5 phone and 2611 Android devices through StoreBench. I/O slowdown and concurrency measurements were excluded from StoreBench, since these tests take too long (around 1 hour) to complete, and would discourage users from using this benchmark tool.

9. CONCLUSION

This paper presents a measurement study on the behavior of reads and writes in smartphones. Among others, we observe that reads experience up to a 626% slowdown in the presence of concurrent writes. The obtained insights are used to design and implement a system that reduces the application delay by prioritizing reads over writes, and grouping them based on assigned priorities. The evaluation on 40 apps demonstrates that SmartIO reduces launch delays by up to 37.8%, and run-time delays by up to 29.6%.

10. ACKNOWLEDGMENTS

We extend our thanks to Prof. Mahadev Satyanarayanan (CMU) for shepherding this work. We would also like to thank Aaron Carroll (NICTA), Dr. Duy Le (EMC), and Tommy Nguyen (RPI) for helpful discussions. Many thanks also go to Mai Anh Do (CNU) for helping with measurements, and Daniel Graham for recruiting StoreBench users. Finally, we thank anonymous reviewers for their comments. This work was supported in part by U.S. National Science Foundation under grants CNS-1250180 and CNS-1253506 (CAREER).

11. REFERENCES

[1] Busybox. `http://goo.gl/CF6vJ`, 2014.
[2] Deadline io scheduler tunables. `http://goo.gl/mB9alK`, 2014.
[3] fio: Flexible io tester ported for android. `http://storebench.com/fio.html`, 2014.
[4] Iostat. `http://goo.gl/OtZ33`, 2014.
[5] Modified facebook application demo. `http://goo.gl/b1AxQ2`, 2014.
[6] Monkey. `http://goo.gl/F14hW`, 2014.
[7] Monsoon monitor. `http://www.msoon.com`, 2014.
[8] Notes on the generic block layer rewrite in linux 2.5. `http://goo.gl/SwdLZ5`, 2014.
[9] One quarter of work devices are smartphones and tablets, forrester finds. `http://goo.gl/K23yGu`, 2014.
[10] Rooting your android. `http://www.androidcentral.com/root`, 2014.
[11] Storebench download. `http://goo.gl/ava9eV`, 2014.
[12] Storebench web. `http://StoreBench.com`, 2014.
[13] Storebench's list of devices. `http://StoreBench.com/list.html`, 2014.
[14] Time man page. `http://goo.gl/dEKuxs`, 2014.
[15] Worldwide smartphone 2013-2017 forecast and analysis. `http://goo.gl/v5vg2b`, 2014.
[16] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX ATC 2008*.
[17] J. Axboe. Linux block io-present and future. In *Ottawa Linux Symp 2004*.
[18] J. Axboe. fio: Flexible io tester. `http://linux.die.net/man/1/fio`, 2014.
[19] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *ACM MobiSys 2010*.
[20] S. Boboila and P. Desnoyers. Performance models of flash-based solid-state drives for real workloads. In *IEEE MSST 2011*.
[21] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 2005.
[22] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *ACM SIGCHI 1991*.
[23] R. Chakravorty, S. Banerjee, P. Rodriguez, J. Chesterfield, and I. Pratt. Performance optimizations for wireless wide-area networks: Comparative study and experimental evaluation. In *ACM MobiCom 2004*.
[24] M. C. Chan and R. Ramjee. Tcp/ip performance over 3g wireless links with rate and delay variation. In *ACM MobiCom 2002*.
[25] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS 2009*.
[26] M. P. Dunn. *A new I/O scheduler for solid state devices*. PhD thesis, Texas A&M University, 2009.
[27] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *ACM MobiSys 2010*.
[28] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/o stack optimization for smartphones. In *USENIX ATC 2013*.
[29] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *USENIX FAST 2012*.
[30] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *ACM EMSOFT 2009*.
[31] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh. Parameter-aware i/o management for solid state disks (ssds). In *IEEE Transactions on Computing 2012*.
[32] K. Lee and Y. Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *ACM EMSOFT 2012*.
[33] D. T. Nguyen, G. Zhou, X. Qi, G. Peng, J. Zhao, T. Nguyen, and D. Le. Storage-aware smartphone energy savings. In *ACM UbiComp 2013*.
[34] D. T. Nguyen, G. Zhou, and G. Xing. Poster: Towards reducing smartphone application delay through read/write isolation. In *Proc. of ACM MobiSys*, 2014.
[35] D. T. Nguyen, G. Zhou, and G. Xing. Video: Study of storage impact on smartphone application delay. In *Proc. of ACM MobiSys*, 2014.
[36] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *ACM UbiComp 2013*.
[37] S. Park and K. Shen. Fios: A fair, efficient flash i/o scheduler. In *USENIX FAST 2012*.
[38] P. Reisner and L. Ellenberg. Replicated storage with shared disk semantics. In *Linux System Technology 2005*.
[39] S. Sen, N. K. Madabhushi, and S. Banerjee. Scalable wifi media delivery through adaptive broadcasts. In *USENIX NSDI 2010*.
[40] K. Shen and S. Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *USENIX ATC 2013*.
[41] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing background email sync on smartphones. In *ACM MobiSys 2013*.
[42] Q. Xu, S. Mehrotra, Z. Mao, and J. Li. Proteus: Network performance forecast for real-time, interactive mobile applications. In *ACM MobiSys 2013*.
[43] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *ACM MobiSys 2012*.
[44] Z. Zhuang, T.-Y. Chang, R. Sivakumar, and A. Velayutham. A 3: Application-aware acceleration for wireless data networks. In *ACM MobiCom 2006*.