Understanding Storage I/O Behaviors of Mobile Applications

Jace Courville
Louisiana State University
jcourv@csc.lsu.edu

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

***Abstract*—In the past few years, mobile devices quickly gained high popularity in our daily life. Designed for ultra-mobility, these small yet powerful devices are fundamentally distinct from traditional computer systems (e.g., PCs and servers) – from the internal hardware architecture and software stack, to application behaviors. Storage, the slowest component in the I/O stack, plays an important role in mobile systems and can greatly affect user experience. In this paper, we present a set of comprehensive experimental studies on mobile storage and attempt to gain insight on the unique behaviors of mobile applications and characterize the performance properties of underlying mobile storage. In our experiments, we carefully selected 13 representative mobile workloads from 5 different categories. Our studies reveal several unexpected observations on mobile storage. Based on these findings, we further discuss the associated implications to mobile systems and application designers. We hope this work can inspire system architects, application designers, and practitioners to pay specific attention to the high-latency I/O operations, rather than completely relying on the default APIs. We also suggest a further look to new opportunities, such as adopting a faster medium in the mobile system architecture, for future research.**

***Index Terms*—Mobile systems; Storage performance; Flash memory; Measurement.**

I. Introduction

Within the last decade, we have experienced the rise of modern mobile devices. Apple recently sold its 500 millionth iPhone [2], and sales of Android devices exceeded one billion units in 2014 [3]. While mobile devices provide high levels of convenience and enable ubiquitous computing to typical users, these small devices, compared to their traditional computer system counterparts (e.g., PCs and servers), carry a set of fundamentally distinct characteristics, from the hardware architecture and software system stack, to application behaviors. These distinctions demand a careful reconsideration of optimizations for system and application design in various aspects.

Storage, the slowest component in the I/O stack, plays a critical role in overall system performance. Interestingly, storage in mobile devices differentiates itself from conventional platforms in several unique properties. (1) *Mobile devices use a flash-based storage medium*. Unlike PC or server systems, which often adopt large-capacity magnetic disks as storage, mobile devices are almost all reliant on NAND flash memory. As a type of semiconductor device, NAND flash memory delivers high-speed read accesses but is highly sensitive to random writes and may suffer from low performance when such writes are encountered [5]. (2) *Mobile devices require latency-oriented optimization*. For mobile devices, it is of high priority to ensure an optimal user experience. This user experience may be severely impacted by storage performance, or more precisely: *latencies*. A high I/O latency may render the device unresponsive. Even worse, such slight slowness may be easily noticed by users and negatively affect user experience. In contrast, *throughput*, another performance metric widely used in traditional storage benchmarking, is not as important in mobile systems. (3) *Mobile devices have both a distinct software stack and distinct application behaviors*. Mobile device applications (i.e., mobile apps) typically run in a protected environment, or sandbox. For example, Android apps normally run in a Java virtual machine. Privileged operations are encapsulated in a small set of strictly defined API interfaces. As a result, popular APIs such as the SQLite library are heavily used in nearly all mobile apps. Such a development practice results in certain patterns which can be commonly found across various mobile apps. On conventional PC and server systems, this is unlikely to happen. In short, because of its unique physical nature, optimization goal, and software stack, mobile storage, compared to traditional computer systems, inevitably exhibits radically different properties. A more important implication to us is that our prior wisdom about storage and the understanding about its influence to system and application performance may not continue to be applicable to mobile devices. Therefore, a demand of a detailed and thorough study to properly understand the critical issues of mobile storage affecting user experience is necessary. In particular, we desire to answer the following important questions:

- *Do there exist any consistent trends in application performance and behaviors over several different categories of applications?* The presence of those trends may suggest that such behavior is not application specific and may exist across an even more broad spectrum of applications.
- *How much of an impact, if any, do storage I/Os contribute to application performance?* Given the diversity of mobile apps, not all applications may be affected by storage I/Os in the same way. It is necessary to understand how much latency applications experience as a result of these I/Os to ensure that each application can perform as efficiently as possible.
- *Which type of storage I/Os contribute most to latency, and what is the root cause behind such impact?* Only by identifying the critical storage I/Os which affect the performance the most, can we effectively identify the most appropriate solutions to address these issues.
- *Does there exist any room for a system level solution to resolve storage I/O latency?* By answering these fundamental questions, we can identify a potential room for optimization of overall mobile app performance through

minimizing the total amount of latency contributed by storage I/Os.

In this paper, we present a comprehensive experimental study to explore several important aspects of mobile storage. We carefully select a set of 13 representative mobile workloads from 5 different categories: ranging from games, multimedia, productivity, network, and device utilities. We run these workloads on a Google Nexus 5 mobile phone with a recompiled Linux kernel. By using `blktrace` and `blkparse` tools, we trace the storage I/O activities for each application and perform an offline analysis on the collected experimental data. It is worth noting that our main purpose is not to benchmark these mobile apps or the device itself. Instead, we attempt to observe the I/O activities from the perspective of the lowest storage layer, characterize and understand the storage I/O behaviors of typical mobile apps, and identify the critical issues of mobile storage with a goal of finding the key aspects for potential optimizations in the future. Based on these observations and analysis, we further discuss important implications to mobile system and application designs. We hope this work can inspire the research community, especially mobile OS architects and mobile app designers, to carefully consider the use of many storage I/O related operations and enhance user experience successfully.

This paper is organized as follows. Section II introduces the background about mobile systems and the storage I/O stack. Section III gives the experimental methodology. Section IV and V discuss the experimental results and their system implications. Related work is presented in Section VI, and the last section concludes this paper.

II. Background

Android is a mobile OS developed by Google for mobile devices. Initially released in 2008, Android currently powers a majority of the mobile devices on the market. Our experiments were performed with the stable Android version 5, "Lollipop". In this section, we give a brief overview of the Android architecture, especially the I/O stack. Figure 1 illustrates the basic architecture of Android OS.

At the top level of the Android architecture is the *application layer*. Unlike traditional desktop systems, an application in the Android OS can be considered a different "user". Each mobile app is assigned a user ID and has respective permissions unique to this ID. These applications are written in Java and run in their own virtual machine, meaning that each application runs independently of another – a quality that is not seen on traditional systems [4]. The *framework layer* consists of the various managers which these applications interact with. For example, an application which uses location based services (e.g., Google Maps) interacts with the location manager to get the geographic location of the device.

The *library/runtime layer* is responsible for interacting with the OS kernel. These libraries allow application developers to quickly access core system services in a protected manner. For example, SQLite (a journaling based light-weight database) enables application developers to keep data (e.g., user settings)

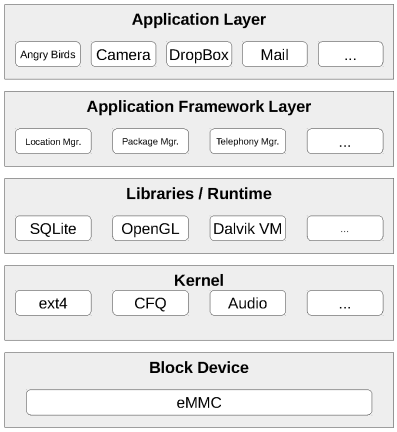


Fig. 1. The Android Architecture

persistent in the form of key/value pairs. Another key component is the Android runtime. Since Android apps are written in Java, the virtual machine runtime, Dalvik (OS versions 4.4 and earlier) or ART, is responsible for application isolation and memory management. The bottom layer is the Linux-based *OS kernel*. The Android OS kernel is a variation of the open-source Linux kernel and contains a set of low-level drivers to control hardware devices, such as the eMMC device and display. The primary file system in Android is the Ext4 file system, which replaced the older YAFFS2 [1]. The CFQ I/O scheduler is responsible for dispatching the I/O requests to the actual eMMC flash block device [4], which completes the whole I/O path.

III. Experimental Methodology

Our experiments were conducted on a Google Nexus 5 device – an Android-powered smartphone. This device is equipped with 32GB of internal eMMC NAND flash based storage and runs an Android Open Source Project (AOSP) version of Android 5. We recompiled the Linux kernel 3.4.0 and ported it to the device to support the capability of block level I/O tracing. We use `blktrace` in Linux to collect the I/O traces of various workloads. The blktrace tool monitors the time stamped events in the I/O path, such as a dispatch of an I/O request and a completion of an I/O request. The traces are first reserved in ramfs and later dumped to persistent storage. We use `blkparse` and our post-processing scripts to process the I/O traces and analyze the traces offline.

For our experiments, we carefully selected 10 mobile apps from both first party and third party sources in order to obtain a true representation of an environment that a typical user may have. When selecting these applications, we chose to prioritize a more real-world set of workloads over choosing workloads that would generate an unrealistically high volume of I/Os, at the trade off of code availability in several of the closed-source apps. As one of our goals is to determine the true impact of storage I/Os on users, we felt that data from these apps would better indicate this impact. Using these mobile apps, 13 different use cases from 5 categories (games,

Workload	Application Type	Read/Write Ratio	Description
Angry Birds	Game	2.03/1	Loading the Angry Birds application
App Removal	Device Utilities	1.35/1	Uninstalling an application from the device
Batch Uninstall	Device Utilities	1/2.79	Using ADB to uninstall several applications at once
Burst Mode Camera	Multimedia	1/204.1	Uses Burst Mode Camera to take a sequence of 100 pictures as a burst
Camera	Multimedia	1/9.12	Uses default camera to take three pictures in quick sequence
Contacts	Productivity	1/2.07	Adding a new contact to the device
Dropbox Sync	Network	1/5.63	Linking an existing Dropbox account to the device and performing an initial sync
E-mail Sync	Network	1/4.25	Linking an existing e-mail account to the device and performing an initial sync
Web Request	Network	1/1.47	Loading the Facebook web site
Route Plotting	Network	1/2.54	Plotting a GPS route using the Google Maps application
MP3 Streaming	Network	1/41.8	Streaming 15 seconds of audio using the Spotify application
Video Playback	Multimedia	1.81/1	Playing back a 5 second recorded video
Video Recording	Multimedia	1/4.25	Recording a 5 second video using the default camera application

TABLE I
WORKLOAD DESCRIPTIONS

multimedia, productivity, network, and device functions) were then designed to create workloads that would best represent a situation that may generate various kinds of I/Os. In order to remove unexpected variance, each test case was completed by first restarting the device. Once booted, we start blktrace and perform the test. Upon completion, we stop blktrace and dump the trace into persistent storage for offline analysis. As overhead resulting from the blktrace operation was of concern, we purposefully stored the output of blktrace within a small amount of DRAM memory to ensure that I/Os directly related to running blktrace would not pollute the collected trace. Each test was run 5 times to ensure that the data was consistent.

All workloads were carefully selected to capture the anticipated largest number of I/Os in critical parts of run time. It is worth noting that our main purpose is to study the impact of storage to user-perceivable performance in practice. As so, we avoid using artificial benchmarks to generate extremely high I/O traffic, which exercises the storage but does not reflect the real-world usage patterns. Also, in our experiments, all workloads were designed to show cases which both involve storage I/Os and practically affect user experience in a typical real-life environment. For example, we were more interested in the process of loading Angry Birds, as the user will be idly waiting for their game to start, over a workload including the user playing Angry Birds, as they will no longer be idling due to storage I/Os. In order to minimize the possibility of latency caused by human interaction, all workloads start at the moment human interaction ends. In all workloads, default configurations were used to get representative results. Table 1 details the type and description of each selected workload.

IV. EXPERIMENTAL RESULTS

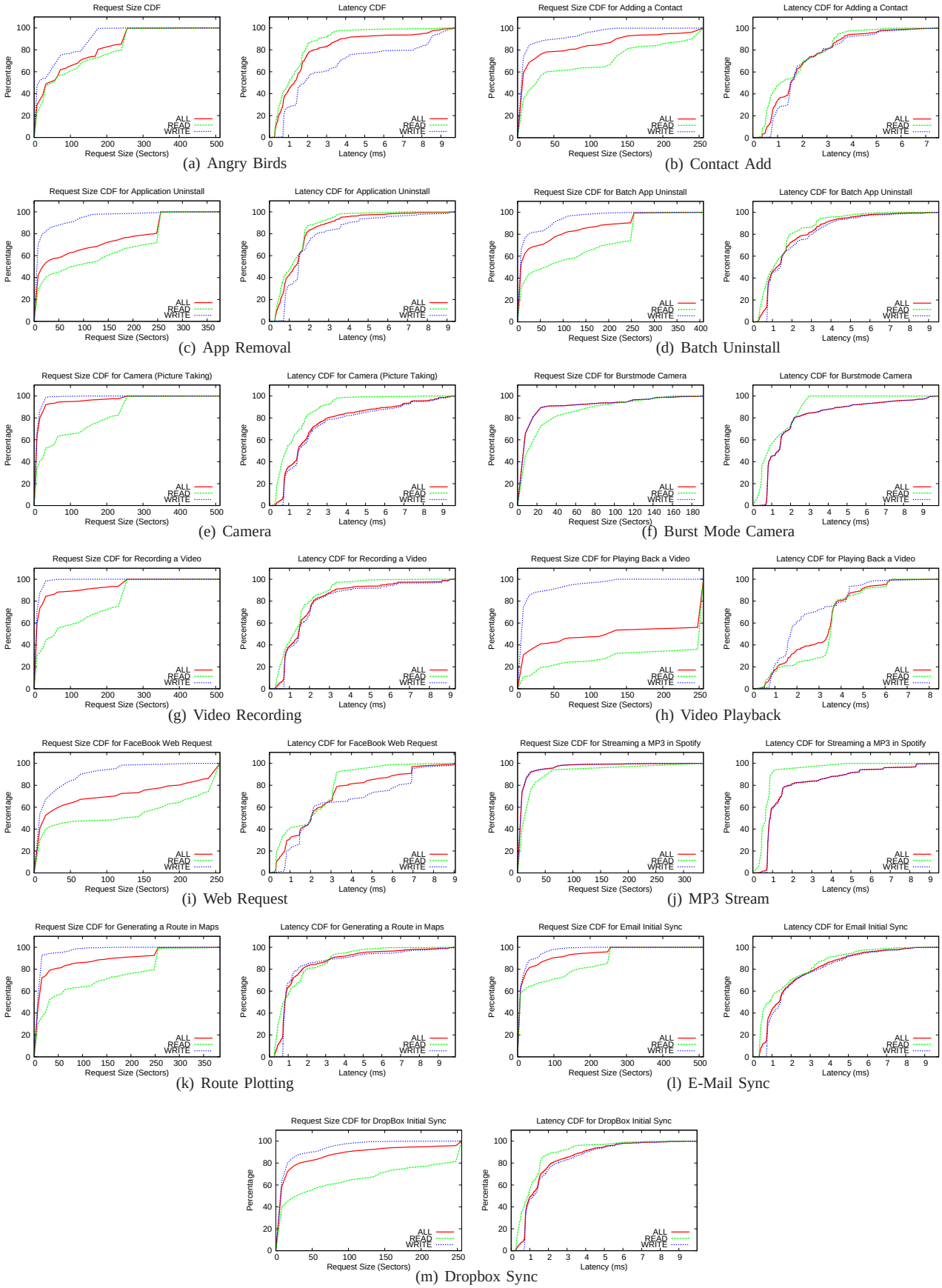
This section presents our experimental results. We first study the two key factors that describe the basic I/O patterns of a workload, namely *request sizes* and *latencies*. Then, we focus on the *flush* operations, which directly impact the I/O speed on an NAND flash based storage. Next, we consider the data access *locality*, which has a strong implication to cache efficiencies. Finally, we discuss the relative influence of I/O operations to the end-to-end application performance.

A. Request Size and Latency Distribution

Request size and latency are two key factors describing the I/O patterns of a workload. The former determines how large each I/O request is, while the latter measures how long each I/O request takes to the point of completion. These two metrics have a direct but non-linear relationship – a small request is not necessarily equal to a smaller latency, and vice versa. Figure 2 shows the distributions of request sizes and latencies of the 13 workloads. In the following, we examine the workloads based on their categories.

Angry Birds: As a typical mobile game, the Angry Birds workload sees mostly smaller request sizes - 67.8% of all requests are less than 64 KB. Comparatively, however, these write sizes are more variable than the other tests. For instance, in Figure 3(a), we can see two vertical bands of writes at the 36 KB and 88 KB ranges. Write latency for Angry Birds is noticeably longer than reads, as 80% of reads are completed in less than 1.87 ms, while it takes up to 7.50 ms for 80% of the total number of writes to be completed. Of these writes, synchronous writes contribute most to the latency incurred from write I/Os. We also find that reads are more predictable than writes. In Figure 3, we see a nearly linear pattern of reads between latency and request size – smaller reads generally take shorter times to complete while larger reads take longer. In contrast, writes show a much larger variance. There exist distinct patterns of a wide range of latency for a similar request size. For example, latency for a request size of 88 KB ranges from 7-10 ms. Such write latency is surprisingly high, especially considering the eMMC flash device has no mechanical components. This is mostly because writes in flash memory may trigger some high-overhead internal operations, such as block cleaning, which make the I/O latencies more variant [5]. Also, large reads tend to have a relatively higher variance in latencies than small ones, as a large read would take longer to complete and is more likely to affected.

Uninstall Apps: Removing both a single application and several applications in a batch job often causes noticeable delay. In Figure 2(c) and 2(d), we see similar patterns in both request sizes and latencies. For request size, we find that 82.4% of writes in a single application uninstall and 80.1% of writes



(a) Angry Birds (b) Contact Add

(c) App Removal (d) Batch Uninstall

(e) Camera (f) Burst Mode Camera

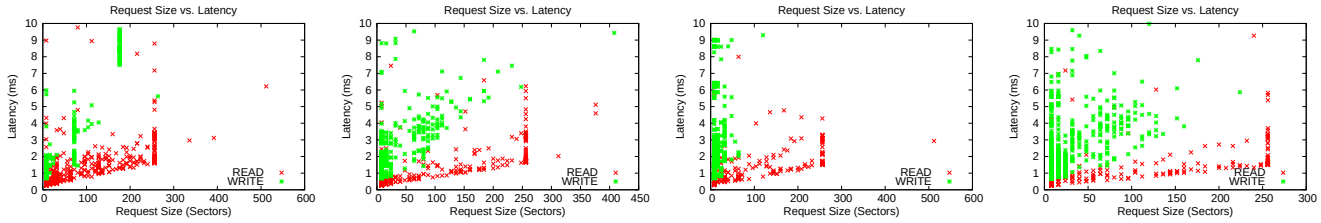
(g) Video Recording (h) Video Playback

(i) Web Request (j) MP3 Stream

(k) Route Plotting (l) E-Mail Sync

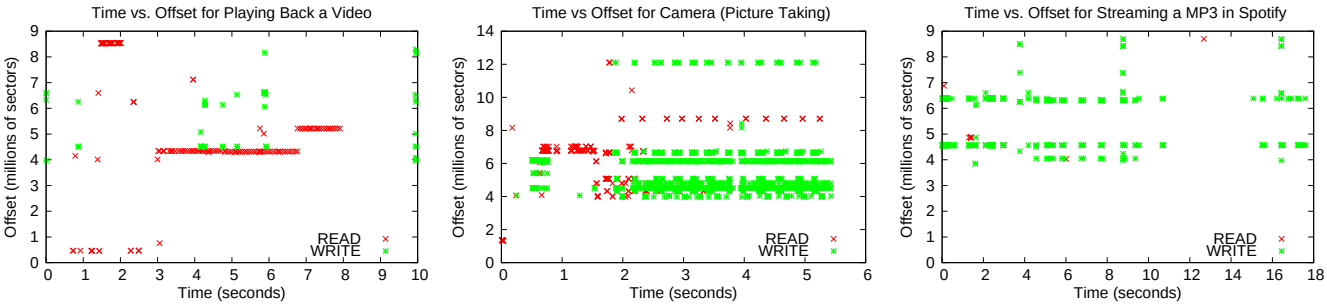
(m) Dropbox Sync

Fig. 2. Request Size and Latency Data by Type for all Workloads (All, Read, Write)



(a) Angry Birds (b) Batch Uninstall (c) Video Recording (d) DropBox Sync

Fig. 3. Selected Request Size vs. Latency Data



(a) Video Playback (b) Camera (c) MP3 Streaming

Fig. 4. Spatial Access Patterns vs. Time. Offset is in units of Sectors.

in a batch application uninstall are less than 16 KB in size. This is because the two workloads involve intensive file system metadata operations, most of which are rather small (e.g., updating inodes). We also find that writes are slightly slower than reads with 80% of I/Os taking under 2.39 ms for writes and 1.77 ms for reads in a single uninstall and under 3.04 ms for writes and 1.89 ms for reads in a batch uninstall. When comparing latency and request size, we find a similar trend of slow writes of very small size and reads in a linear pattern, as seen in Figure 3(b). The only apparent difference between the single- and batch- uninstall workloads is the quantity of reads and writes being appropriately larger in scale. This indicates that, for such a batch of metadata-intensive workloads, storage I/Os happen mostly in a sequence, and no buffering effect has been observed.

Contact: The Contact Addition experienced several very small writes - about 73.7% of all writes were only 4 KB. Reads, however, were larger than writes - only 64.6% of reads were less than 64 KB. Other data continued to follow typical trends – writes were slightly slower than reads. Compared to other workloads, this workload does not have to load or store data of any significant size, and subsequently it does not present any surprising information.

Multimedia: As shown in Figure 2(e-h), we find several key themes in the four multimedia workloads, Camera, Burst Mode Camera, Video Recording, and Video Playback. First, writes tend to be small. Write sizes of less than 16 KB make up 86.9% of the Camera workload, 81.2% of the Burst Mode Camera workload, and 88.0% of the Video Recording workload. This is because the three workloads involve intensive writes, and frequent flushes create a sequence of small writes. Video Playback is unique. It sees a wider spectrum of write request sizes; however, we still find that most of these writes are small. Second, latency distributions are also similar between the write-intensive multimedia applications. In the Burst Mode Camera, Camera, and Video Recording workloads, 80% of the I/Os are completed in less than 2.20 ms, 3.02 ms, and 2.21 ms respectively. Video Playback shows different patterns - I/Os tended to experience higher latency, with 58.2% of I/Os taking over 3 ms to complete. Third, the latency-vs-request-size trends in multimedia workloads are similar to other workloads – a heavily variable concentration of small writes with a relatively more linear pattern of reads, shown in Figure 3(c). We also find that Burst Mode Camera, Camera, and Video Recording are all quite variable as there are a significant number of writes between 1 and 10 ms of latency. Finally, Video Playback shows a unique pattern. This workload has only 75 total writes, and of these writes, only 5 were larger than 4.5 ms. Unlike other multimedia workloads, Video Playback is read intensive. A large portion of this workload involves retrieving the video from storage to play it back. We see a larger number of slower reads than in other workloads. Also, this workload has the second fewest number of I/Os of any other workload, as it retrieves the video from storage in large (128 KB) chunks. Figure 4(a) illustrates this behavior. We can see a distinct band of I/O reads at the same offset for a large duration of the process of playing back the video. Comparatively, we see the other multimedia applications which save data to storage (e.g., Camera) writing data in small chunks, as shown in Figure 4(b). This process also proves to be extremely costly. Our Camera workload had the largest percentage of I/O latency than any other workload at nearly 70% of the run time.

Network Apps: The storage I/O behaviors of network-intensive apps follow patterns unique to this category. Of the 5 network applications, shown in Figure 2(i-m), each workload

had a majority of small writes. For each network workload, most writes were smaller than 16 KB. For example, the MP3 streaming workload had 92.6% writes smaller than 16 KB, and the Web Request workload had 76.5% writes being less than 16 KB. Reads had some slight variation between each workload and were larger than writes. The MP3 Streaming workload is unique. It has significantly smaller I/O reads, with 76.5% of reads being less than 16 KB, compared to the other Network workloads which, as in the Route Plotting workload which had only 41.0% reads less than 16 KB. This is likely due to the streaming effect, where most reads can be directly satisfied in memory. In this category, we also found that unlike other workloads, I/O request latencies had slower asynchronous writes than synchronous writes. This difference can be most drastically noted in MP3 Streaming, where asynchronous writes are greater than 4 ms in over 74.1% of requests. We continue to see patterns of small slow writes and linear reads when comparing latency and request size. In summary, network workloads are unique in that they experience I/O behavior somewhat similar to other workloads, which is an unexpected finding. An example of this may be seen in the spatial write pattern of MP3 Streaming shown in Figure 3(d) – there are constant I/O writes with very few reads. The reasoning for this may be due to the device having to download and store the data from the network.

B. Flushes

In Android systems, the SQLite library provides a light-weight database for mobile apps to store small pieces of data persistently (e.g., user settings). In order to ensure data consistency, the dirty data in the OS page cache needs to be synchronized to the persistent storage. As a result, the Android operating system frequently uses a flush operation (e.g. FUA and FLUSH) to send buffered data to storage to ensure the persistence. While this is a necessary function to preserve data, too much flushing can result in increased latency, thus degrading application and device performance. We find such a trend of excessive flushing in our analysis of our workloads, characterized by flushes at short intervals, with a only small number of small sized I/O writes between each flush operation. We will examine specific workload categories in greater depth. Figure 5 depicts an average case of flushing in three metrics: the number of I/O requests which occur between flushes, the size of I/O requests between each flush, and the time between successive calls to flush. Table II lists the number of I/O requests, the total I/O request size, and the time interval between two consecutive flushes.

We find in Figure 5 that there are very few I/O requests that take place between successive flushing operations. In 90% of cases, the largest number of I/O requests between two flushes is 74 in the Web Request workload, with all other workloads having less than 49 I/O requests. Workloads such as Angry Birds and Application Remove saw less than 26 requests and 29 requests respectively, and Dropbox Sync had fewer than 14 requests in 90% of cases. We see a varying number of I/O requests between workload categories as well. Network workloads saw as few as less than 10 requests at the 90th percentile (MP3 Streaming, Route Plotting) and as many as less than 74 requests at the 90th percentile (Web Request).

Workload	Requests	Data Size	Time
Angry Birds	26	2028 KB	0.398 sec
App Removal	29	808 KB	0.289 sec
Batch Uninstall	16	332 KB	0.252 sec
Contacts	40	240 KB	1.41 sec
Burst Mode Camera	16	80 KB	0.116 sec
Camera	22	124 KB	0.060 sec
Video Recording	23	204 KB	0.099 sec
Video Playback	49	4196 KB	3.30 sec
Dropbox Sync	14	116 KB	0.216 sec
E-mail Sync	18	180 KB	1.10 sec
Web Request	74	4412 KB	3.13 sec
MP3 Streaming	10	60 KB	0.512 sec
Route Plotting	10	64 KB	0.147 sec

TABLE II
I/OS BETWEEN FLUSHES (90TH PERCENTILE OF CDF)

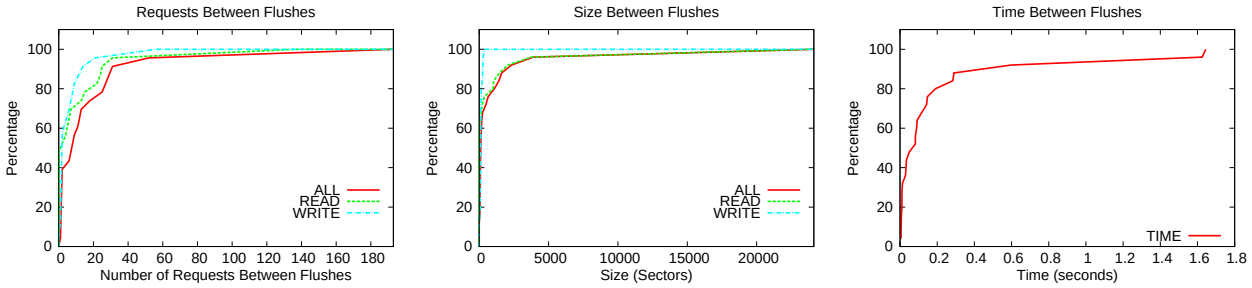
The amount of data of I/O requests between successive flushes is also small but varies between the workloads. Several workloads had total data sizes of less than 128 KB between flushes in 90% of cases, while three outliers are the Angry Birds (2028 KB), Video Playback (4196 KB), and Web Request (4412 KB) workloads.

The time interval between successive flushes is also small. We found a range of typically short intervals. In 8 of the 13 workloads, these intervals were between 0.1 and 0.4 seconds, which means 2-10 flushes happen every second. Workload categories seem to have some influence: three of the five longest intervals occur in the network category, while the three shortest intervals occur in the multimedia category. Since the multimedia apps, such as Camera, involve heavy writes, this suggests that the number of storage I/O writes affects flushing behavior; as more data is written, flushing becomes more frequent.

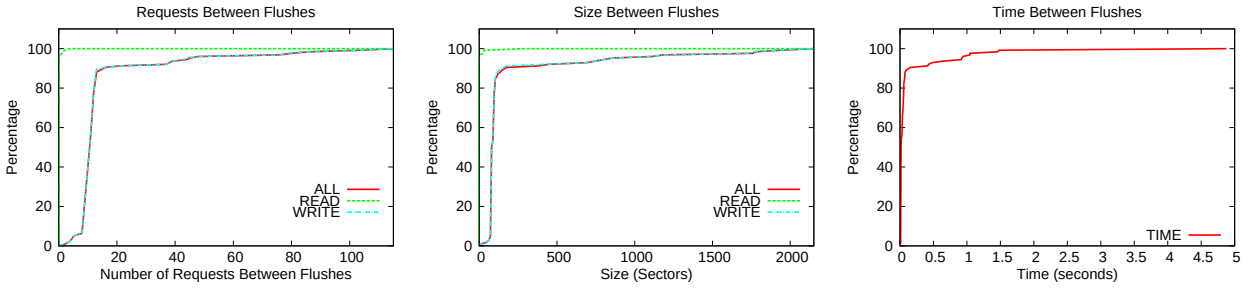
In all, we see very aggressive flushing for the mobile applications. Most have very few requests of small sizes at short intervals between flushes. Application category has influence on the flushing behavior of workloads. Write I/O intensive categories have high frequency, low request numbers, and small request sizes between each flush operation. Less I/O write intensive workloads show less frequent flushes with more requests and larger request sizes between flushes. The biggest reason for variation occurring in flushing between different workloads is the importance of ensuring that the data generated by the respective workload is written to storage. This flushing scheme, however, is problematic because it contributes heavily to the overall latency of storage I/Os.

C. Locality

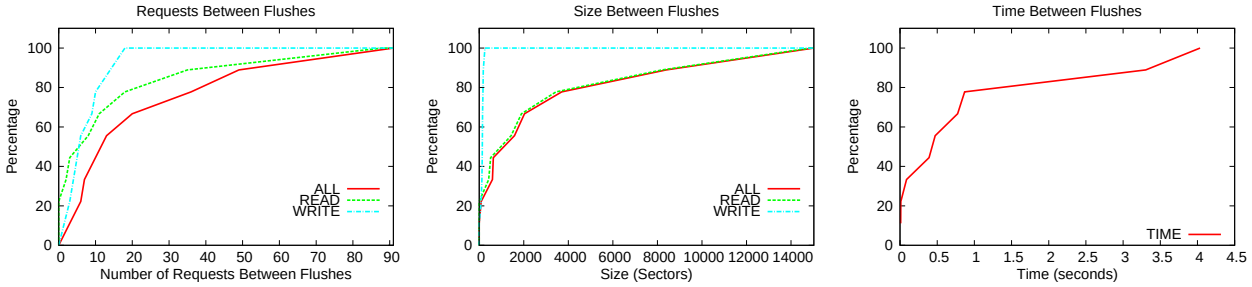
In this section, we briefly analyze spatial locality trends among the workloads in this study. This analysis will refer exclusively to the spatial locality of storage writes, as in each workload, blocks being accessed from read operations were accessed only one time. This is due to the memory being able



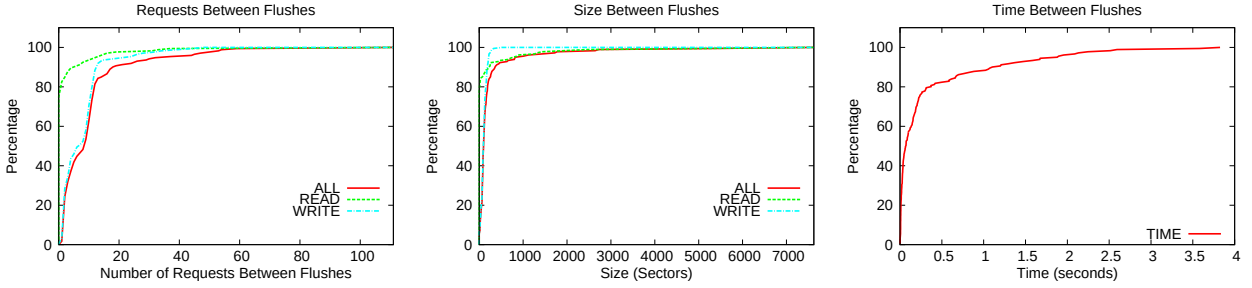
(a) Application Removal



(b) Burst Mode Camera

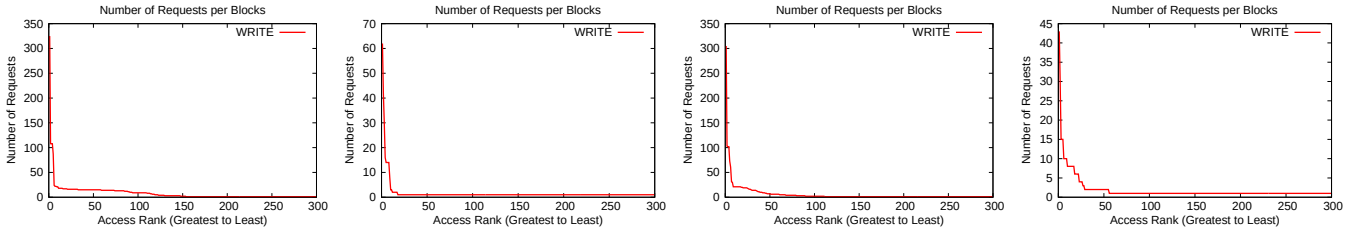


(c) Video Playback



(d) E-mail Sync

Fig. 5. Data Gathered Between Two Flushes



(a) Burst Mode Camera (b) MP3 Streaming (c) Camera (d) Batch Uninstall

Fig. 6. Selected Locality Behavior. All access ranks after 300 have been ignored.

to fully contain the working set. To confirm this, we repeated the workload under the conditions of reduced available RAM (1 GB) and found that blocks were being accessed twice due to page cache replacement. In general, we have found some localities to be good, while others are heavily skewed.

Workloads with good localities include Burst Mode Camera, Figure 6(a), with 153 blocks covering nearly all of the I/O write requests to 880 total unique blocks, with the 10 most accessed blocks accounting for 25.6% of all accesses. Dropbox Sync also had 187 blocks being re-written. Other workloads with multiple blocks being re-written include Contact Add, MP3 Streaming, and Web Request. Most workloads, however, had heavily skewed localities, where one or very few blocks were re-written in some cases hundreds of times. In an extreme case, Camera saw one block out of 3,293 unique blocks being re-accessed 305 times, as shown in Figure 6(c). Other workloads also saw very few blocks being accessed in a range between about 10 to 150 accesses.

D. End-to-end Impact of Storage I/Os

Although storage I/Os are generally slow, the end-to-end impact of storage I/Os to mobile application performance is complex and depends on various factors, such as relative computing and network speed. In this section, we show the aggregate storage I/O time in the overall workload completion time. Figure 7 and Table III provide the details.

Based on the percentage of I/O latency throughout the duration of the workload, we can distinctly identify 3 major groupings of applications. *(1) Light-I/O Applications:* Four of the five network workloads were found to be lightly affected by I/O latency, with the exception of Dropbox Sync, which stores a lot of data to the device in its initial sync. All other network workloads were found to have less than 9% of I/O latency. This suggests that the network may have a larger contribution in the run time of certain applications. Video Playback was also only lightly affected by storage I/Os, as a large number of sequential reads can be quickly loaded to memory by the OS prefetching. *(2) Moderate-I/O applications:* The two uninstall workloads, Burst Mode Camera, and Dropbox Sync are moderately affected by storage I/Os. This implies that in some cases, even a network based application may be affected in part by storage I/O latency if it is storing a large amount of data. *(3) Heavy-I/O applications:* Three applications are heavily affected by storage I/O latency. The Video Recording application is understandably affected, as it stores a high quantity of data to the storage when the video is completed and saved. Angry Birds was the second worst affected workload. This may be in part due to the complexity of the game itself, as the features of the game may both read the game files and write back progress data to allow for settings or game saving. The Camera workload was the most affected by I/O latency, which accounts for nearly 70% of the run time, as high resolution pictures are written.

In general, our findings indicate that the significance of storage I/Os to the end-to-end performance of mobile apps varies across workloads.

V. System Implications

With these key observations, we are now in a position to present several important system implications. Additionally, this section also provides an executive summary of our answers to the questions we raised at the beginning of this paper.

I/O writes are very small and of varying I/O latency. In all of our mobile applications, we see excessive small write I/Os. The write I/Os exhibit strong locality; some blocks are heavily rewritten while most are only written once. These I/O writes can be of a highly variable latency. Regardless of how large or small the write, we saw a range between 1 ms and 10 ms in nearly every workload. This suggests that write performance, overall, is quite poor. This is mostly because NAND flash does not handle random and small writes well. Consequently, applications which write a lot of data to storage will be the biggest culprits of I/O latency. Because of these small and latent writes, the camera workload saw a near 70% I/O overhead. Even with the Dropbox sync workload's heavy reliance on a network connection to download data, small writes contributed to this workload being the 5th most affected workload from I/O latency. This implies that mobile app designers should pay specific attention to write operations, especially frequent small writes. Buffering and clustering small writes into large ones can effectively reduce the effect.

Aggressive flushing is a common practice in all applications. In nearly every case, a pattern of very short page cache flushes occur (e.g., fsync()). These flushes contain typically less than 40 I/Os of small size and happen very frequently. This is caused by applications which are constantly triggering a flush operation in order to ensure data safety and persistence in the event of some failure. This, in turn, requires the system to stop and wait for the data to be written to the storage. Such a blocking effect further magnifies the effect of slow writes on NAND flash. This aggressive flushing has a large impact to overall system performance: because each interval between flushes is so short, little data is being written per operation, implying that more time is spent waiting than may be necessary. As a result, the user is left waiting for all of the I/Os to complete before they can proceed further. Because these flushes trigger sequences of short and random writes, they inadvertently contribute to the overall problem of storage latency and reduce the possibility of organizing more favorable large writes as well. A potential solution to reducing latency would be to use these flush operations conservatively, thus reducing the frequency of data being written to storage. Also, mobile app developers should understand the impact of such flushes to ensure I/O operations are not issued arbitrarily and should also avoid abusing the SQLite library for randomly storing small data items - the main source of frequent, small synchronized writes.

I/O reads are mostly one-time access and of relatively predictable latencies. In our experiments, reads exhibit a rather linear behavior between latency and request size in nearly every workload. This finding is consistent with our understanding about reads in NAND flash, in that they are

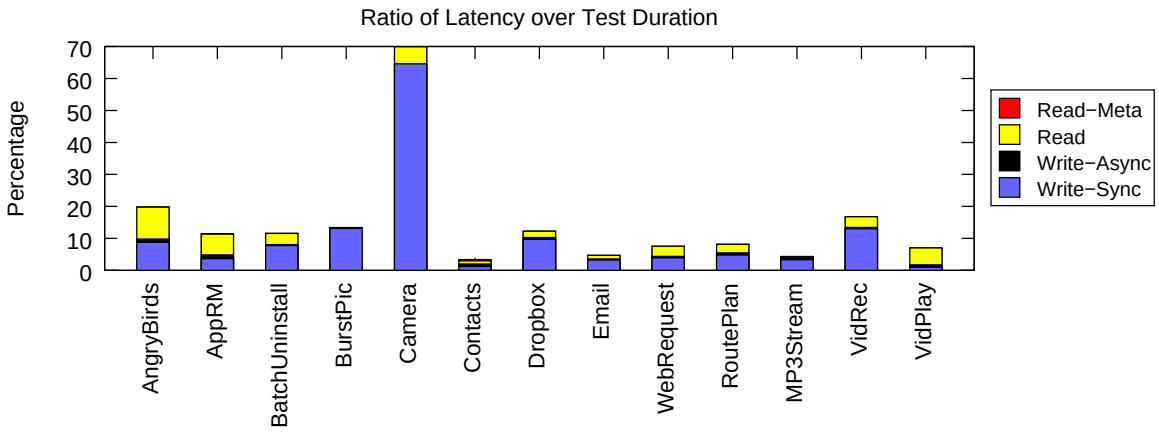


Fig. 7. Ratio of Latency over Test Duration for a given workload

Application Name	Number of I/Os	Number of Reads	Number of Writes	Test Duration (ms)	Percentage of I/O Latency
Angry Birds	846	567	279	7414.26	19.8508
App Removal	511	294	217	5667.04	11.5023
Batch Uninstall	1238	326	912	13346.3	11.5861
Burst Mode Camera	2257	11	2246	17880.6	13.3792
Camera	2319	229	2090	5429.02	69.8771
Contacts	348	113	235	18736.6	2.97313
Dropbox Sync	1983	299	1684	18313.9	12.2474
E-mail Sync	2177	414	1763	61163.4	4.69179
Web Request	173	70	103	4484.59	7.51333
Route Plotting	1950	550	1400	27424.3	8.16327
MP3 Streaming	728	17	711	17606.3	4.33635
Video Playback	256	165	91	7961.2	7.01778
Video Recording	1304	248	1056	10301.7	16.7913

TABLE III
I/OS BY WORKLOAD

fast and predictable. In addition, these reads are, almost in all cases, one-time access. This is directly a result of the memory being capable of fully containing the workload with a fairly small working-set size. As long as the memory is capable of containing the working-set, the system will not see a significant amount of latency from read I/Os, as it only needs to read storage blocks once. Accordingly, reads do not contribute as much of an impact to I/O related latency. This also is in part due to the read benefits that come from using NAND based flash storage, as these reads will be able to be completed efficiently. This, again, implies that mobile system and app developers need to focus more on optimizing writes.

Synchronous Writes make up a majority of the I/O latency. Of the four different specific types of reads and writes, we see primarily read-aheads and synchronous writes by quantity. By percentage of latency over test duration, however, we find that synchronous writes contribute an overwhelming amount of the total latency from storage I/Os. This latency is compounded by the aggressive flushing experienced in each workload. Because applications are constantly flushing at short intervals (e.g., 200 ms), the device has to spend a large amount of time writing back the data to storage and the applications remain in a state of being blocked. This finding implies that we need to either reduce the amount of synchronous writes or speed them up. The former can be achieved through less use of flushes, while the latter could be realized by adopting a faster storage medium, such as persistent memory [6].

The impact of storage I/Os to the end-to-end application performance is workload dependent. In total, storage I/O based performance degradation appears to be reliant on the type of application being used. The typical percentage of latency due to storage I/Os falls between 7 and 20 percent. We found that network-heavy applications, such as the E-mail application or the MP3 Streaming application, had much less latency due to storage I/Os at just over 4% each. Conversely, multimedia based apps saw typically more latency attributed to storage I/Os, such as the camera workload with nearly 70% latency. These trends are not necessarily strict, as the Dropbox workload indicated that an application may be both heavily reliant on a network and affected by storage I/Os. Comparing Camera and its burst mode version, we can find that leveraging local buffering can effectively reduce the impact of storage I/O. It also implies that other mobile apps with intensive writes should consider such a simple technique to optimize performance.

Compared to desktop applications, mobile apps show several unique and distinct characteristics. In our experiments, we have observed several interesting properties of mo-

bile apps. First, we see a large volume of synchronous writes and frequent use of flushes in mobile apps. This is related to the mobile apps' heavy reliance on the SQLite library, and we rarely see such patterns in desktop applications. Second, most mobile apps have a more relatively small working set than typical desktop applications. On one hand, it allows most reads to be comfortably accommodated in memory. On the other hand, it makes mobile apps more write I/O intensive and the write performance issues even more obvious. Desktop applications, in contrast, are typically more read intensive and most writes are asynchronous. Thus, the optimization goals for mobile and desktop applications are very different.

In general, through our experimental studies, the implications of the answers show that there is a definite space for optimization with respect to storage I/Os. A solution that could overcome the need to constantly commit data to storage at a short interval, thus only writing small amounts of data, would in turn reduce the overall latency which the user must experience. By reducing the amount of flushing, much of the storage I/O latency can be negated which will consequently optimize application performance on a much larger scale. In the meantime, we should also note that storage I/Os account for a moderate portion of the overall mobile app performance, which indicates that when optimizing mobile app performance, we must consider all system components in a whole package.

VI. Related Work

In recent years, inefficiencies of mobile device optimization have received interests in academia. These prior studies cover various aspects of mobile systems, from power management (e.g., [7], [9], [19], [21]), privacy and security (e.g., [8], [10], [12]), applications (e.g., [11], [22], [24], [25]), and many others. In this section, we will focus on the prior work that is most related to this paper.

Storage I/Os are important to mobile system performance and user experience. Kim et al. have presented benchmarks of Android performance, showing that storage may contribute more of an effect on system performance than previously thought [14]. In contrast to this early work, which focused primarily on SD-card based external storage, our studies are based on the internal eMMC flash storage and show that the impact of storage to the overall user-perceivable performance is moderate for most mobile apps. Lee and Won first noticed several inefficiencies within the various layers of the Android stack [18]. They found that although these layers have been well designed, several issues with journaling still existed and were causing issues with device performance. This anomaly, later known as *Journaling of Journal*, was determined as the result of the innate competition of the journaling actions of the SQLite database which unexpectedly triggered the high-cost journaling in the Ext4 file system [13]. Due to these unexpected interactions between SQLite and Ext4, solutions to address these problems have been proposed, though at this time have not been adopted into Android, as our results suggest that small sync write issues still remain a problem within Android. Jeong et al. investigated the results of changing various features of the Android operating system and discovered that by making changes to the file system and changing the operating mode of the SQLite database, they were able to remove this journaling of journal effect and achieve a 300% performance upgrade from SQLite [13]. Recently, Kim et al. developed an algorithm known as LS-MVBT (multi-version B-tree with lazy split) to reduce the number of `fsync` calls which trigger the journaling in Ext4 [16]. By maintaining the recovery information within the database instead of a journal log, they were able to achieve a 1,220% performance improvement over the default SQLite logging modes. Shen, Park, and Zhu have also identified several implementation limitations of the Android operating system and through applying a custom journaling mode they were able to improve overall database performance by 7% [23]. Additional work in storage areas contributes to the understanding of the uniqueness of mobile storage to traditional desktop systems. Chen et al. performed several tests on flash based SSDs and identified several performance issues that can appear with writes with regards to flash storage, indicating a need to study flash storage uniquely of traditional storage understandings of HDDs [5]. To overcome these innate problems, and those caused by the Journaling of Journal anomaly, Lee et al. show that F2FS, a file system designed to perform for devices using flash storage, outperforms the existing EXT4, effectively removing Journaling of Journal by using append only logging [17]. Recognizing the impact of writes to reads, Nguyen et al. proposed a scheduling scheme to reduce application delay by prioritizing read over write I/Os and grouping them based on priorities [20]. Kim et al. developed a buffer cache replacement scheme to influence the I/O performance on flash [15]. In this paper, our main focus is to characterize the I/O behavior of mobile apps and its interaction with the underlying flash memory. Besides observing frequent flushes, we have also found that the end-to-end impact of these I/O latencies varies depending on applications, which deserves further research.

VII. Conclusions

Mobile devices have become increasingly important in our daily computing. In this paper, we present a comprehensive study on the storage I/O behavior of mobile applications and their interaction with the underlying flash-based storage. By carefully selecting 13 workloads from 5 categories, we perform extensive experimental studies in an attempt to discover the trends that would allow for overall device performance optimization. Our analysis shows that although the number of storage based I/Os comprises a smaller number of the overall I/Os for a running application, the mobile apps exhibit unique I/O patterns. I/O writes, especially those from synchronous writes and fast flushing, contribute most to latency in these storage I/Os. As a result, a large window of optimization exists at the system level for Android OS design. On the other hand, we have also acknowledged that the end-to-end impact of I/O latencies to application performance depends on

workloads, meaning that optimizations must be customized to applications.

Acknowledgment

The authors thank anonymous reviewers for their constructive comments to improve this paper. This work was supported in part by Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, National Science Foundation under grant CCF-1453705, and generous support from Intel Corporation.

References

[1] Yet Another Flash File System. http://www.yaffs.net.

[2] Without Much Fanfare, Apple Has Sold Its 500 Millionth iPhone. http://www.forbes.com/sites/markrogowsky/2014/03/25/without-much-fanfare-apple-has-sold-its-500-millionth-iphone/, 2014.

[3] Market Share: Devices, All Countries, 4Q14 Update. http://www.gartner.com/newsroom/id/2996817, 2015.

[4] Android. The Android Source Code.

[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, pages 181–192, New York, NY, USA, 2009. ACM.

[6] F. Chen, M. P. Mesnier, and S. Hahn. A Protected Block Device for Persistent Memory. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*, Santa Clara, CA, June 2-6 2014.

[7] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom'15)*, Paris, France, September 7-11 2015. ACM.

[8] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Istanbul, Turkey, March 14-18 2015. ACM.

[9] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, San Francisco, CA, June 15-18 2010. ACM.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, Canada, October 4-6 2010. USENIX.

[11] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu. Reliable, Consistent, and Efficient Data Sync for Mobile Apps. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, Feburary 16-19 2015. USENIX.

[12] S. Guha, M. Jain, and V. N. Padmanabhan. Koi: A Location-Privacy Platform for Smartphone Apps. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, April 25-27 2012. USENIX.

[13] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 309–320, San Jose, CA, 2013. USENIX.

[14] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, February 14-17 2012.

[15] H. Kim, M. Ryu, and U. Ramachandran. What is a Good Buffer Cache Replacement Scheme for Mobile Flash Storage? In *Proceedings of the 2012 ACM SIGMETRICS Conference (SIGMETRICS'12)*, London, UK, June 11-15 2012. ACM.

[16] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 273–285, Santa Clara, CA, 2014. USENIX.

[17] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 273–286, Santa Clara, CA, Feb. 2015. USENIX Association.

[18] K. Lee and Y. Won. Smart Layers and Dumb Result: IO Characterization of an Android-based Smartphone. In *Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT'12)*, pages 23–32, New York, NY, USA, 2012. ACM.

[19] J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang. On the Energy Overhead of Mobile Storage Systems. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Santa Clara, CA, Feburary 17-20 2014. USENIX.

[20] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang. Reducing Smartphone Application Delay through Read/Write Isolation. In *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, Florence, Italy, May 18-22 2015. ACM.

[21] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the 6th Conference on Computer Systems (EuroSys'11)*, Salzburg, Austria, April 10-13 2011. ACM.

[22] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, October 8-10 2012. USENIX.

[23] K. Shen, S. Park, and M. Zhu. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 287–293, Santa Clara, CA, 2014. USENIX.

[24] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are Web Browsers Slow on Smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*, Phoenix, AZ, March 1-2 2011. ACM.

[25] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. In *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*, Taipei, Taiwan, June 25-28 2013. ACM.