Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone

Kisung Lee and Youjip Won

Dept. of Electronics and Computer Engineering
Hanyang University, Republic of Korea
{kisunglee|yjwon}@hanyang.ac.kr

ABSTRACT

In this paper, we offer an in-depth IO characterization of the Android-based smartphone. We analyze the IO behaviors of a total of 14 Android applications from six different categories. We examine the correlations among seven IO attributes: originating application, file type, IO size, IO type (read/write), random/sequential, block semantics (*Data*/*Metadata*/*Journal*), and session type (buffered vs. synchronous IO). For the purposes of our study, we develop Mobile Storage Analyzer (MOST), a framework for collecting IO attributes across layers. Let us summarize our findings briefly. SQLite, which is the most popular tool for maintaining persistent data in Android, puts too much burden on the storage. For example, a single SQLite operation (update or insert) results in at least 11 write operations being sent to the storage. These are for creating short-lived files, updating database tables, and accessing EXT4 *Journal*. From the storage point of view, more than 50% of writes are for EXT4 *Journal* updating. Excluding *Metadata* and *Journal* accesses, 60-80% of the writes are random. More than 50% of the writes are synchronous. 4KB IO accounts for 70% of all writes. In the Android platform, each SQLite and EXT4 filesystem requires a great amount of effort to ensure reliability in supporting transactions and journaling, respectively. When they are combined, the results are rather dumb. The operations of SQLite and EXT4, when combined, generate unnecessarily excessive write operations to the NAND-based storage. This not only degrades IO performance but also significantly reduces the lifetime of the underlying NAND flash storage. The results of this study clearly suggest that SQLite, EXT4, and the underlying NAND-based storage need to be completely overhauled and vertically integrated so as to properly and effectively incorporate their respective characteristics.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.4.3 [**OPERATING SYSTEMS**]: File Systems Management—*Access methods*; D.4.8 [**OPERATING SYSTEMS**]: Performance—*Measurements*

General Terms

Measurement, Performance

Keywords

Android, Smartphone, IO Characterization, NAND flash

1. INTRODUCTION

The smartphone is growing ever more popular. It is expected that over 800,000,000 smartphones will be sold in 2016 [8]. The smartphone has changed and continues to change the way people live. It provides a variety of extremely accessible functionalities, for example, social networking, picture management, music, and movies, to list only a few. Application programs in the smartphone have made all of these possible. While we are well familiar with the functions and features that these applications provide, little is known about how they interact with the underlying Operating System and underlying storage. IO access characteristics in enterprise servers [22], OLTP servers [17], Web servers [10], and desktop PCs [26, 13] are relatively well understood, but not in smartphones.

In this work, we aim to obtain a comprehensive understanding of how smartphone applications utilize and stress underlying storage. To that end, we analyze the interaction across software layers: applications, smartphone platform (Android OS), filesystem, and underlying storage device. For this purpose, we study the behaviors of the 14 representative smartphone applications. We categorize these into six groups: Web, legacy phone applications, SNS, Multimedia, System, and Game; further, we select one or more representative commodity applications from each of the categories. In our cross-layer IO characterization, we carefully select seven IO attributes to investigate the correlations among the IOs; The attributes are originating application, file type (`.apk`, `.db`), block type (*Metadata*, *Data*, and *Journal*), IO type (read/write), session type (buffered vs. synchronous), randomness, and IO size.

The well-defined layered structure of the modern Operating System makes it impossible to collect these IO attributes at a single point of observation. For example, at the block device layer, neither the file type nor the session type of a given IO operation is available. Thus, in this work, we develop Mobile Storage Analyzer (MOST), a framework for

EMSOFT'12, October 7-12, 2012, Tampere, Finland.
Copyright 2012 ACM 978-1-4503-1425-1/12/09 ...$15.00.

collecting and analyzing block IO operations of the Android smartphone. MOST, significantly, enables collection of all seven attributes at a block device level. It addresses three major issues: LBA-to-file mapping, LBA-to-process mapping, and retrospective mapping. First, MOST obtains file type information from the LBA. Second, MOST obtains the process information that *originally* triggered a given IO. The Android system delegates all IO requests to the dedicated kernel process. The kernel daemon, when observed at the block device level where the trace is captured, looks like the original process that generated the given IO request. Therefore, we modify the Linux kernel of Android to keep the process ID for a given IO. The third issue, which is the most intricate of the three, is the retrospective LBA-to-file mapping. The Android platform creates a large number of ephemeral files due to the SQLite operation. In MOST, LBA-to-file mapping is performed posthumously, so that it does not interfere with ongoing system behavior. However, posthumous analysis, if the owner file of the respective blocks has been deleted, inevitably yields orphan blocks. We modify the Linux kernel to capture IO operations to these short-lived files. MOST can perform LBA-to-file mapping retrospectively: this way, it can find the owner file even though that file had been deleted. MOST uses `blktrace` [11] and `debugfs` [24] to collect block level IO traces and to map LBA to files in the EXT4 filesystem, respectively. MOST is available at [1]. Let us summarize our findings as follows.

Smart layers and dumb result: Android OS exports the SQLite database to allow applications to manage their own persistent data in a structured manner. Most Android platforms use the EXT4 filesystem to manage underlying NAND flash-based storage devices. Both SQLite and EXT4 adopt sophisticated techniques for reliable maintenance of data. SQLite creates temporary files (logs) for each database operation to support transactions [5], and EXT4 adopts journaling [23]. But when they work in combination, the result is rather dumb. SQLite calls *fsync()* three times in every database operation. In EXT4, each *fsync()* entails two to three IOs for *Journal* writes. Combining all of these, we observe that a single SQLite operation (update/insert) results in at least 11 block IOs to the storage device! This excessive IO behavior not only negatively affects IO performance, but also significantly shortens the endurance of NAND storage devices, which is of the utmost concern in adopting NAND flash for computing devices.

Most write operations are 4KB: In all applications, more than 70% of write operations are 4KB. They consist mostly of database updates from SQLite, file creation and deletion by SQLite, and *Journal* writes. NAND storage for mobile devices (e.g. eMMC [9] and UFS [6]), therefore, should focus on efficient handling of 4KB random write operations.

Buffered IO is rare: We observe that in smartphone applications, for example Contact, SMS, Browser and SNS applications, buffered IO represents less than 30% of write. This is an IO characteristic unique to the smartphone. This phenomenon, in fact, has not been observed in the desktop or server environment, because smartphone applications use SQLite to manage persistent data, exploiting *fsync()* to preserve the atomicity of database operations. This phenomenon dictates that the filesystem and NAND flash storage for mobile devices should devote greater efforts to optimizing themselves for delivery of more synchronous IO performance.

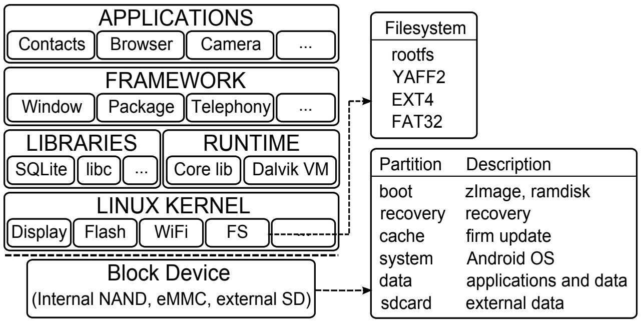

Figure 1: Android Architecture and Storage Partition

Table 1: Storage Partition of typical Android smartphones: Nexus One, Nexus S, and Galaxy S2.

	Internal Flash		eMMC		External SD	
Nexus One	boot	rootfs	None		sdcard	FAT32
	recovery	rootfs			-	-
	cache	yaffs2			-	-
	system	yaffs2			-	-
	data	yaffs2			-	-
NexusS	boot	rootfs	system	EXT4	None	
	recovery	rootfs	data	EXT4		
	cache	yaffs2	sdcard	FAT32		
Galaxy S2	None		boot	rootfs	sdcard	FAT32
			recovery	rootfs	-	-
			cache	EXT4	-	-
			system	EXT4	-	-
			data	EXT4	-	-
			sdcard	FAT32	-	-

Each Storage Partition has unique access characteristics: Android OS devotes great care to harboring files with similar characteristics at the same partition. Because IO requests to individual partitions exhibit unique IO characteristics, this approach makes partition management easier and simpler. However, since the physical and logical addresses of a block do not coincide in NAND flash, the blocks in each partition are not likely to be clustered together in the storage device. Therefore, efforts to maintain files with similar characteristics in the same partition should be exploited in the NAND flash.

The results of this study provide insights into and directions for designing future smartphone filesystems and storage subsystems.

The remainder of this paper is organized as follows. In Section 2, we briefly describe the architecture of Android OS along with the storage configurations of Android smartphones. Section 3 discusses the measurement environment, and Section 4 introduces Mobile Storage Analyzer (MOST), a framework for tracing IO operations in Android smartphones. Sections 5 and 6 provide the results of an analysis of the IO characteristics of applications and daily usage. Section 7 discusses the future of smartphone filesystem design. Section 8 acknowledges prior efforts, and Section 9 concludes the paper.

Table 2: Applications and their use. [ ] denotes short names in figures. * denotes pre-installed in Nexus S. Others are installed from the market. All applications are executed for one minute and repeated ten times.

Category	Application	Scenarios
Web	Dolphin Browser [Br]	1. Execute browser and open “www.google.com.” 2. Web search by any keyword. 3. View results and repeat web searching two more times.
Basic	Contact* [Con]	1. Execute Contact and scroll lists. 2. Search a person by name. 3. Delete the item. 4. Create a new item. [Precondition: Contact is filled with 200 addresses.]
	SMS* [Sms]	1. Execute SMS. 2. Write a message. 3. Send a message. 4. Receive a message.
SNS	Twitter [Twi]	1. Execute Twitter and view new tweets. 2. Write a status.
	Facebook [Fb]	1. Execute Facebook and view new messages. 2. Write a status.
	Kakao-Talk [Kt]	1. Execute Kakao-Talk and choose a counterpart. 2. Start to exchange messages.
Multimedia	Camera* [C]	1. Execute Camera and take a picture. 2. Take four more pictures.
	Camcorder* [Cc]	1. Execute Camcorder and start to record. 2. After 50 seconds, stop recording.
	Media player* [Me]	1. Execute Media player and select a movie file. 2. Play the movie and change volume.
	Music player* [Mu]	1. Execute Music player and select a music file. 2. Play the music and change volume.
	Gallery* [Gal]	1. Execute Gallery and scroll thumbnails. 2. Select a picture and view. 3. Repeat this five times. [Precondition: Gallery is filled with 500 pictures.]
	Youtube [You]	1. Execute Youtube and search a video by any keyword. 2. View the video.
System	Install* [Ins]	1. Execute Android market and select one application. 2. Install the application. (We select 3-5MB applications that can be completely installed in 1 minute.)
Game	Angry birds [Gam]	1. Execute Angry birds. 2. Select a level and play the game.

2. BACKGROUND

2.1 Brief note on Android OS

Android [7] is an open-source software stack for mobile devices. It consists of the Operating System (Linux kernel), Java Virtual Machine (Dalvik) and various libraries. Figure 1 shows the Android architecture. Android applications are written in Java and packaged to the Android application package file (`.apk`). Android includes a set of libraries used by various components: SQLite, libc, Media libraries, and others. The Android application runs on its own process, with its own instance of the Dalvik virtual machine. The Dalvik VM executes files in the Dalvik Executable (`.dex`) format, which is optimized for a minimal memory footprint.

2.2 Storage Configuration for Smartphones

Android manages several filesystem partitions: `/boot, /recovery, /cache, /system`, and `/data`. Table 1 summarizes the storage configuration and partition information for three smartphones recently deployed to the market: Nexus One [2] (Jan. 2010), Nexus S [3] (Dec. 2010), and Galaxy S2 [4] (May 2011). The storage configuration of the smartphone evolves with time. Nexus One, the first-generation Android reference phone, has a 512MB internal raw NAND flash and an external SD slot. It uses the YAFFS2 filesystem to manage the storage partitions (`/cache`, `/system`, and `/data`) in the internal NAND flash. Beginning from Android 2.3 (Nexus S), EXT4 filesystem is used to manage `/system`, and `/data` on an eMMC block device (16GB). Galaxy S2, the most recent among the three phones, does not have any internal flash storage. It uses eMMC to harbor all storage partitions. The EXT4 filesystem is used to manage storage partitions on eMMC; YAFFS2 filesystem is no longer used. In this paper, we focus on the `/system`, `/data`, and `/sdcard` partitions because Android applications only access these partitions. The rest of the partitions are only used for firmware updates (`/cache`) and boot image maintenance (`/boot, /recovery`).

3. MEASUREMENT ENVIRONMENT

3.1 Device: Nexus S

We select an Android smartphone, Nexus S [3] which runs Android OS 2.3 (Gingerbread) based on Linux Kernel 2.6.35.9. This phone is a reference model that represents the standard architecture of Android OS. As most Android-based smartphones have a similar storage configuration, the results of this study should be sufficiently representative. Table 1 shows the partition information of Nexus S. The `/system` partition (512MB) is mounted with READ-ONLY. It contains Android-executable files and pre-installed applications. The `/data` partition (1GB) is mounted with READ-WRITE and contains the user’s data. These data can include contacts, messages, settings, and applications. Both `/system` and `/data` are formatted with EXT4. The `/sdcard` partition (13.3GB) is formatted with FAT32, and can be used to store external data such as media files, documents, and other types.

3.2 Applications

Smartphone is a literally multipurpose device. It is used, additionally to its phone functionality, as a Camera, MP3 player, Web browser, and SNS. For any comprehensive study, selection of a sufficiently representative set of applications is mandatory. We define six application categories (Basic, Web, SNS, Multimedia, System, and Game) and select 14 applications from among them.

Basic: Contact is an address book that stores people’s names, phone numbers, and other identifying information. Contact operations such as search are frequently shared by many basic applications, for example SMS, VOICE CALL, and others. SMS is a means of communicating with others in traditional mobile phones, but its usage is gradually decreasing with the popularity of SNS applications.

Web: Web browser is an extremely popular application on the Internet, and is included in the smartphone as well. We select the Dolphin browser for our test, which is very popular in the Android OS.

SNS: Social Networking Service (SNS) is another very popular application for the smartphone. We select Facebook, Twitter, and Kakao-Talk, which latter is one of the most rapidly growing IP-based multi-party chatting services.

Multimedia: We select six applications in this category: Camera, Camcorder, Media player, Music player, Gallery, and Youtube. These applications are enabling the smartphone to replace traditional handheld multimedia devices such as portable media players (PMP), MP3 players, and digital cameras. Youtube provides an interactive means of sharing videos; users can easily upload and play videos through this website. This represents a very different mode of access from that of traditional Media players, which store and play video files locally in the device.

System: The installing application is another new feature of the smartphone. It enables the user to install new applications in the Application market, and greatly expands the usability of smartphones.

Game: Advanced computing ability makes the smartphone a powerful video game console. We select Angry birds, which is a very popular puzzle game in Android and also in Apple's iOS.

3.3 Collecting Data

Table 2 summarizes the application scenarios. All of the applications were executed for one minute and repeated ten times. To verify the generality of the application-specific IO traces, we also collected IO traces from daily usage. We installed the Mobile Storage Analyzer to the smartphone being tested, and collected IO traces for seven different 24 hour periods from Dec. 1 to Dec. 14, 2011. The purpose was to examine the aggregate storage access pattern of the smartphone. We used the smartphone in the normal ways, without any specific scenarios. Web surfing, exchanging messages, listening to music, and playing games are among the typical functionalities tested. As the smartphone is a personal and private device that users are reluctant to share with others, installing a modified Android kernel on the smartphones tested, and collecting traces in the regular manner, is neither an easy task nor a major part of this work. Collecting IO traces from just one user served our purpose adequately well in verifying the results of our analysis.

4. MOST: MOBILE STORAGE ANALYZER

For the purposes of the present study, we develop Mobile Storage Analyzer (MOST). It consists of (i) a modified Linux kernel that maintains processes and file-related information for IOs; (ii) a block analyzer that enables identification of a file for a given block, and (iii) `blktrace` utility. Figure 2 provides a schematic illustration of MOST. We make MOST publicly available at [1].

Due to the layered structure of a modern IO subsystem, it is not possible to identify session-related information at the block device level. When an IO request is passed across layers, for example, from the filesystem to the block device layer, the session-related information (i.e., file id and process id), are lost. In order to be able to analyze the relationships among blocks, respective files, and processes, the information needs to be collected from the different layers. MOST collects the IO trace at the block device driver level and deduces the file information and the process information for each respective block. MOST addresses three reverse-mapping issues: LBA-to-file mapping, LBA-to-process mapping, and retrospective LBA mapping.

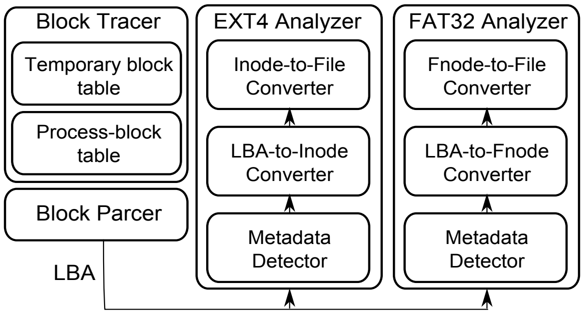

Figure 2: Mobile Storage Analyzer

Table 3: Output of Mobile Storage Analyzer

1	IO completion time
2	Flags for read and write
3	Sector address and IO size
4	Process id and process name
5	Block type: *Metadata*, *Journal*, and *Data* block
6	File name in case of the *Data* block

For LBA-to-file mapping, MOST can reverse-map the disk block to the respective file where it belongs. It accepts a logical block number as an input, and generates a file name. MOST uses `debugfs` [24] to reverse-map the block in the EXT4 filesystem, and an in-house module for the FAT32 filesystem.

MOST identifies the original process that issued a given IO. In Android, *mmcqd* daemon manages the mmc card device driver and is responsible for issuing all block IOs. Without any modification, `blktrace` reports that all block IOs are initiated by the *mmcqd* daemon, which is not the information we are interested in. We create the process-block table in the Android Kernel. The entry of the table is <LBA, process id>. When the IO scheduler inserts the IO request into the queue, MOST inserts the <LBA, process id> information into process-block table. MOST references the process-block table later in order to retrieve the process id with a given LBA.

MOST allows retrospective LBA mapping. In Android, we find that many files are short-lived and are created and rapidly deleted by SQLite. Proper understanding how these files are utilized is very significant. Although they are short-lived, these files are all *fsync()*ed to NAND storage, which greatly affects system performance. We need the file information for a given LBA when the trace is recorded, not when posthumously analyzed. When MOST initiates the analysis procedure for a given LBA, the temporary file where the block belonged might have been deleted and therefore cannot be found. To address this issue, MOST creates a file-block table in the Android kernel. A file-block table is an array of <LBA, file>. When the IO scheduler plugs in the LBA to the scheduler queue, MOST inserts <LBA, file> entry to file-block table. Later, MOST references this table to obtain the file information for a given LBA. To reduce the table size, MOST inserts an <LBA, file> entry only for temporary files, that is, when the file extension is `.db-journal`, `.db-mjxxxx`, `.bak`, or `tmp`. When `blktrace` creates a log for

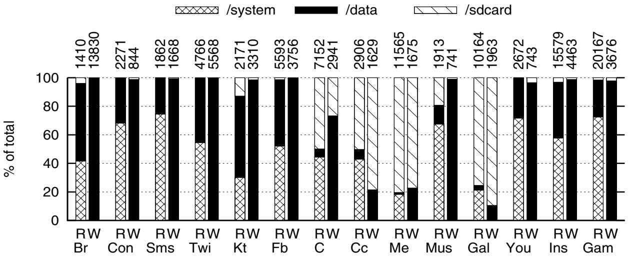

Figure 3: IO distribution on each partition. The number at the end of each bar indicates the number of IO for R (Read) and W (Write), respectively.

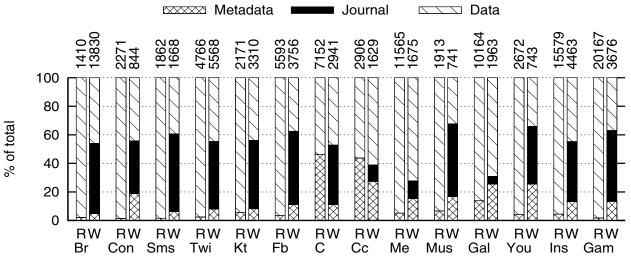

Figure 4: IO distribution on block type. The number at the end of each bar indicates the number of IO for R (Read) and W (Write), respectively.

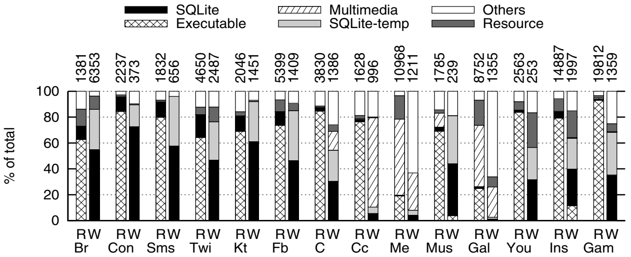

Figure 5: IO distribution of each file type. The number at the end of each bar indicates the number of *Data* block IO for R (Read) and W (Write), respectively.

the trace file, it consults the temporary block table to determine if the given block belongs to the temporary files that triggered the respective IO.

One of the important objectives of this study is to find IO characteristics based on the block type. MOST categorize logical blocks into three types: *Metadata*, *Journal*, and *Data*. In the EXT4 filesystem, *Metadata* blocks are blocks harboring a superblock, group descriptor, data block bitmap, inode bitmap, and inode table. In the FAT32 filesystem, *Metadata* blocks correspond to blocks harboring a boot record and File Allocation Table (FAT). *Journal* is a journal block of the EXT4 filesystem. *Data* blocks are those harboring file data and directory entries. Table 3 illustrates the entry format of MOST output.

5. ANALYSIS OF APPLICATION USAGE

5.1 Accesses on Filesystem Partition

We first examine how the individual applications access each of the partitions. Figure 3 illustrates the results. The labels on the X-axis denote the short names of the 14 applications. The label RW denotes Read and Write. The number on the top of each bar denotes the number of the respective operations. We find that multimedia applications and the rest exhibit very different partition usage patterns. In non-multimedia applications, accesses on the `/data` partition are mostly write, and `/sdcard` is rarely accessed. Multimedia applications, for example Camera, Camcorder, Gallery, and Media player, access the `/sdcard` partition much more frequently. These access characteristics reflect the Android OS partition management strategy, which aims to effectively exploiting the very limited storage capacity.

The `/sdcard` partition is used mostly by multimedia applications that read and write very large files such as MP3, pictures and movies. The underlying filesystem should be optimized to effectively accommodate this workload. FAT32, which maintains file blocks as a linked list, leaves much to be desired in its handling of large files.

5.2 Access Characteristics of Block Type

We define the three block types in NAND storage devices: *Metadata*, *Journal*, and *Data*. We examine how the individual applications utilize each type. This is critical information for both the filesystem and the storage controller. It can be used in devising a hot/cold identification algorithm for the NAND storage controller. The filesystem also can use this to design an efficient layout and caching strategy. Figure 4 illustrates the results.

In legacy text-based applications such as Browser, Contact, and SMS, most read operations are for *Data* blocks. Read operations for *Metadata* constitute less than 5% of the total. Interestingly, in Camera and Camcorder, 50% of read operations are for *Metadata*. We find that these applications aggressively allocate *Data* blocks to accommodate incoming data (jpeg images or video recordings). This aggressive allocation behavior results in heavy accesses of the File Allocation Table of FAT32 for location or allocation of such *Data* blocks.

One notable point in Figure 4 is the IO operation on EXT4 *Journal*. In most applications, *Journal* block accesses represent 40-50% of write operations. Given that write is approximately ten times slower than read in NAND flash, and given also the limited cell duration of NAND flash, such excessive journaling activity not only can seriously aggravate system performance but also can seriously curtail NAND device lifetimes. We herein dedicate a separate section (section 5.4) to an in-depth discussion of EXT4 journaling activity in Android OS.

5.3 Access Characteristics of File Type

We examine the correlation between each file type and the ways in which individual applications utilize them. We categorize the files into six groups: executable, SQLite (database tables), SQLite-temp (ephemeral database files), multimedia (image, video, and music), resources (application properties), and others. The file type is determined based on the file extension: executable files (`.apk`, `.dex`, `.odex`, `.so`), SQLite (`.db`), SQLite-temp (`.db-journal`, `.db-mjxxxx`), multimedia (`.3gp`, `.jpg`, `.mp3`, etc), resources (`.dat`, `.xml`, `.cache`, etc), and others (including directory entry).

Figure 5 illustrates the results. In most cases, executable files are accessed in READ-ONLY mode. Installation is an exception to this rule. Among all of the 14 applications tested, more than 60% of read operations are for accessing executable files. A number of studies have proposed prefetching of executable files to reduce application launch latency [15]. These techniques manifest themselves further in the smartphone environment, since most read operations are on executable files.

Excepting some multimedia applications, the dominant fractions of the write operations relate to SQLite database tables. Android OS provides several options for managing persistent data, but SQLite is the most popular storage method because application developers can easily make structured and private databases for individual applications. We find that even multimedia applications use SQLite to store information. Media player and Music player use *AudioService* to adjust the audio volume level with respect to the user's volume control, and *AudioService* records the volume level to `setting.db`. In Camera and Camcorder, `external.db` is used to manage media files.

We also find that there are many ephemeral files, most of which, surprisingly, are created by SQLite. SQLite, in order to implement atomic commit and rollback capabilities, does make use of many temporary files in the course of database processing. These temporary files have `.db-journal` and `.db-mjxxxx` extensions. Severe performance degradation occurs due to *fsync()* calls for each creation and update in these files.

Our analysis shows that each file type exhibits very unique access characteristics. In most applications, 80% of write operations are on SQLite database, SQLite-temp, and resource files. Seventy percent (70%) of read operations are for executable files. Significantly, the strong correlation between the file type and its access characteristics can be effectively exploited by the NAND storage controller. For example, FTL can exploit file type information in making hot/cold decisions on a given logical block. This makes the hot/cold identification algorithm more accurate, and, subsequently, the performance of garbage collection for page-mapping FTL, and of log block merge operations for hybrid FTL, can improve significantly.

5.4 Analysis of Excessive Journaling

In section 5.2, we observe that the number of *Journal* writes accounts for 40-50% of all write operations. Given that EXT4 is mounted with *Metadata only* journaling (Ordered mode), the number of *Journal* writes should constitute a much smaller fraction of the entire write operation, which phenomenon we regard as not only excessive but also anomalous. This phenomenon is observed in all applications that use SQLite. We perform an in-depth analysis of this issue using the Facebook application to find a root cause. Specifically, we examine the IO patterns generated when SQLite performs write operations on a Facebook database table: `fb.db`. Figure 6 illustrates the LBA write accesses over a 60 msec period. The accesses are clustered in two distinct LBA regions: 100,000 and 300,000, the former being the location of the EXT4 *Journal* and latter, the locations of the SQLite database tables and the temporary files, respectively. In the Figure, there are five dashed rectangles, each denoting the IOs generated by a single database operation (insert or update). The bottom graph of the Figure is a magnified image of the IOs involved in this single operation.

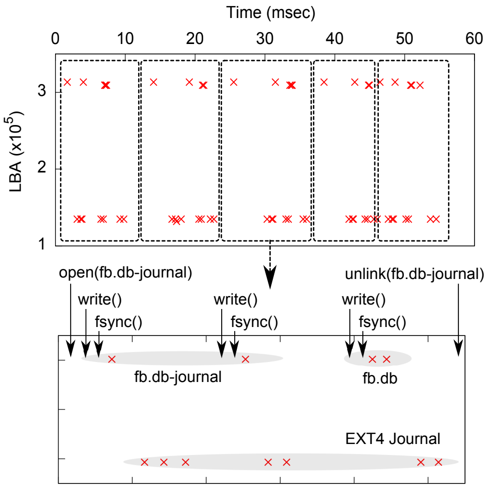

Figure 6: Block IO accesses of Facebook database table (fb.db) for 60 msec.

Let us explain the details of the operations in one of the single dashed rectangles in Figure 6. The first IO (in the $3x10^5$ LBA region) is for creating and updating a `fb.db-journal` file. The second IO (in the $3x10^5$ LBA region) is a commit log write to this journal file. The third IO (in the $3x10^5$ LBA region) consists of actually two IOs, which are visible in the magnified image. In each of these steps, SQLite forces the results to storage via *fsync()*. The IO overhead compounds according to the EXT4 filesystem. The IOs at the bottom are for EXT4 *Journal* writes. In EXT4, a *fsync()* call accompanies two or three *Journal* writes, each of which accounts for a writing journal descriptor and a metadata. There are a total of 7 IOs in the EXT4 *Journal*. In summary, a single SQLite operation triggers at least 11 writes to the storage device when used with the EXT4 filesystem. In this example, the database update to `fb.db` consists of two of the 11 IOs. Eighty percent (80%) of the write operations are for purely managerial purposes!

Here, we suggest an improvement. *fsync()* forces both the *Metadata* and *Data* of a file to storage. However, if write does not cause any removal from or addition to the *Data* block, storing *Metadata* might be unnecessary. In this case, *fdatasync()*, instead of *fsync()*, is a good alternative for mitigation of the burden of excessive journaling. *fdatasync()* forces only *Data* block. In the magnified bottom graph of Figure 6, the second write operation to `fb.db-journal` updates the header portion of a file of 12Bytes. It does not cause any removal from or addition to the *Data* block. In this case, *fdatasync()* can be a better choice, as it can significantly improve the excessive journaling phenomenon.

5.5 IO Size Distribution

We examine the IO size distribution. We group the IOs into five categories according to size: $\leq$ 4KB, $\leq$ 16KB, $\leq$ 64KB, $\leq$ 256KB, and $>$ 256KB. We perform analyses on the IO and Byte counts, respectively.

Let us first examine the IO count. In all applications, the 4KB IO is dominant (top graph in Figure 7), accounting

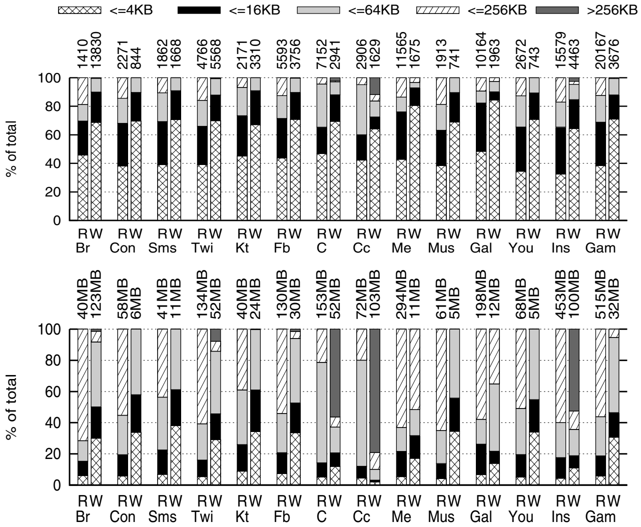

Figure 7: IO size distribution. At the top is the distribution by IO count, and at the bottom, the distribution by Byte count.

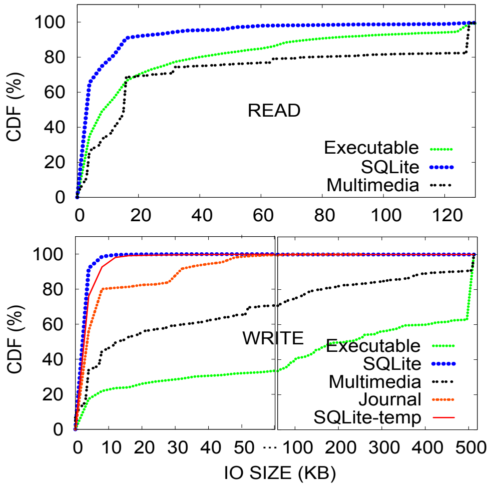

Figure 8: Cumulative IO size distribution

for 40% and at least 65% of read and write operations, respectively. Even in Camera and Camcorder, which handle large data files such as video clips and pictures, 4KB constitutes more than 60% of writes. We find two reasons for this phenomenon. The first is excessive journaling, explained above. The second, which is very interesting and important, is the updates on the File Allocation Table (FAT) of the `/sdcard` partition. While multimedia applications normally deal with large IO (e.g. copying images, mp3 files, video files), the FAT32 filesystem updates its FAT object very frequently when a new block is allocated. Therefore, via a reduction of the number of FAT synchronization operations (e.g. delayed write or periodic synchronization), we can greatly prolong the cell lifetime. This approach is even more useful in dealing with the sdcard, since sdcard normally uses an inexpensive MLC (or TLC, Tri-Level Cell) flash device, due to its strict cost requirements. These devices have a very limited Erase/Write cycle. The success of the storage and filesystem of Android-based smartphones critically relies on efficient handling of small random writes.

The bottom graph in Figure 7 shows the Byte count statistics. For read, whereas 4KB IO constitutes the dominant fraction of all IO requests (40%), the resultant IO volume is not significant (less than 5%). For write, 4KB IOs account for 35% and 17% of text-based applications (Browser, SMS, and SNS) and multimedia applications (Camera, Media player, Gallery), respectively. Contrary to our expectation, 4-64KB IOs constitute a significant fraction of entire IO volumes. The reason for this is the EXT4 *Journal* IO size (Figure 8). Half of *Journal* writes are accessed in 4KB units, but the rest are accessed in much larger sizes (8-50KB). In Camera and Camcorder, IOs larger than 256KB constitute 60% and 80% of the entire writes, respectively. These applications create new video (or image) files; the maximum write IO is as large as 512KB.

We examine the IO size distribution as subject to individual file types and EXT4 *Journal*. Figure 8 illustrates the cumulative IO size distribution, which we determine by aggregating IO traces from the 14 Android applications. Resources and other files show a pattern similar to SQLite, and so are not shown in the Figure. In read, the 4KB IOs are dominant in SQLite. There are no read accesses to SQLite temporary files, because those files are accessed only to recover from a crash. Executable and multimedia files are accessed in much larger units. One interesting phenomenon is that among all of the file types, the maximum read IO is 128KB; the eMMC interface has a maximum IO size of 512KB, which is four times larger than that of the SAT interface. Few applications exploit this larger IO size in reading files.

In write, half of *Journal* writes are accessed in 4KB units, but the rest are accessed in much larger sizes (8-50KB). SQLite and its temporary files are accessed in 4KB units. The actual updated data in SQLite and its temporary files are much smaller than 4KB (not shown in the graph). Most SQLite files request data updates of 1KB size. Half of the IOs from SQLite temporary files are for updating about 3KB of data, but the rest are only for updating 12Bytes commit logs. This throws light on an important design guideline for future mobile storage design: the FTL mapping unit should be smaller than a page. This mapping technique generally is called sub-page mapping.

5.6 Access Characteristics: Spatial Aspect

We examine IO randomness for individual smartphone applications. Random writes are considered to be very harmful to NAND flash storage in the performance and reliability aspects, and thus in-depth investigation is required. The top half of Figure 9 illustrates the sequential and random IO volume of read operations for all types of blocks (A) and for only *Data* blocks (D). For all of the applications, reads are mostly sequential, which is to say that 80% of read operations are sequential. We find that most read operations are for executable files or multimedia files, and that the read operations for these files are mostly fully loaded (128KB).

The bottom half of Figure 9 shows the spatial characteristics of write operations. In the 14 applications excepting Multimedia applications (Camera, Camcorder, Media player, and Gallery), most of the *Data* block writes (60-80%) are random. These accesses are for SQLite database tables

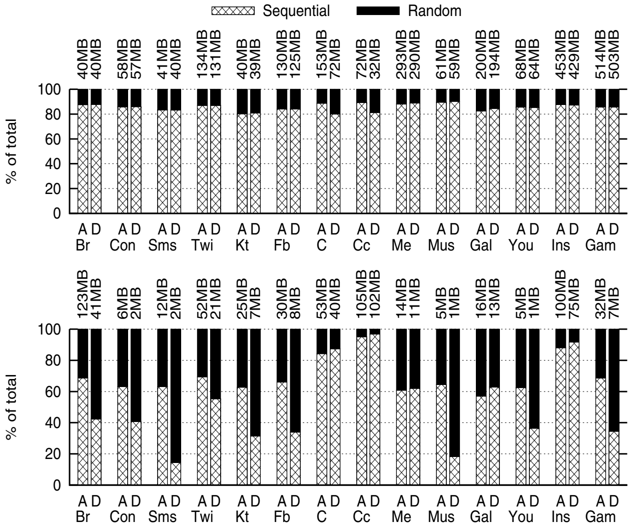

Figure 9: Randomness of IO traffic: Read (top) and Write (bottom). A denotes all blocks; D denotes *Data* blocks.

and their temporary files. If we consider all of the block types, sequential write constitutes rather a significant portion. This is because more than half of all *Journal* writes in the EXT4 filesystem fall in the 10-50KB range. This analysis provides us with important guidelines for performing IO characterization studies. If the spatial pattern of write is examined without consideration of the block type, the dominant fraction of writes will be sequential, which result can be misleading. When we focus our analysis on *Data* block accesses, writes are mostly (more than 60% in terms of `volume`) random. These *Data* block IOs are not only random, but also are frequently accessed and forced to storage.

5.7 Buffered write vs. Synchronous write

Buffered IO and synchronous IO stress the system in different ways, and thus their optimization should be approached from different perspectives. We examine the fraction of writes that are buffered and synchronous IO from applications, respectively. *Journal* and *Metadata* IOs are not included in this case study. We find that a buffered IO is *rare*. Most smartphone applications are found to update data in a synchronous manner, due to the fact that they use SQLite to manage information. However, multimedia applications such as Media player, Music player, and Gallery exploit buffered IO to download contents.

Synchronous write is no longer a supplemental option for updating data. In Android-based smartphones, synchronous write constitutes a significant portion of all write IO, and thus efficient handling of it is critical. IO subsystems of the modern Operating System adopt interrupt-driven IO to effectively share CPU cycles among multiple threads and, thereby, cope with the large IO latency (longer than a few msec) of a storage device. Polling-based IO is being revisited for IO subsystems for high-end SSD and storage class memory [25]. We argue that polling-based IO subsystems should be carefully studied as an alternative IO subsystem for future smartphone storage.

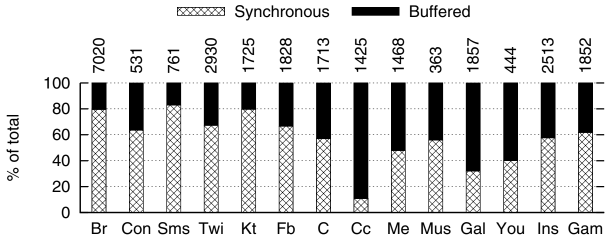

Figure 10: Buffered write IO distribution. The number at the end of each bar indicates the number of *Data* block IOs.

6. ANALYSIS OF NORMAL DAILY USAGE

It is important to verify that the IO characteristics we have observed in the individual applications properly incorporate the IO characteristics of the normal daily use of Android-based devices. Accordingly, we installed a Mobile Storage Analyzer in a volunteer's smartphone (Nexus S) and collected IO traces for seven periods of 24 hours each. The objective of this user study is not to perform any extensive user survey on smartphone usage; rather, the purpose is to verify that the IO characteristics we have observed from individual applications are not groundless but have practical implications.

Figure 11 plots the results. We examine both IO count and Byte count characteristics. In summary, the IO characteristics observed from daily usage are very similar (if not identical) to what we observed in an IO characteristics study for individual applications.

- **Partition Accesses**: From the IO count point of view, 90% of the writes are for `/data` and 60% of the reads are for `/system`. The IOs for each partition exhibits very different characteristics. This phenomenon suggests that the FTL of NAND-based storage should be able to effectively handle IOs for both write-dominant partitions and read-dominant partitions, which requires precise hot/cold detection and support for multiple-address-mapping granularity.
- **Block types**: The EXT4 *Journal* IO accounts for more than 60% of all writes. In contrast, *Metadata* write constitutes only 10% of all writes. The overhead of *Journal* IOs, significantly, are very expensive.
- **File types**: SQLite and its temporary files constitute 70% of write IOs. Most applications maintain their persistent data using SQLite. Accesses to the executable files are mostly read.
- **IO SIZE**: 4KB IO is dominant in both read and write. It accounts for 40% and 50% of the IO counts for read and write, respectively. The 4KB IO does not constitute a significant fraction of IO volume. However, overall system performance will critically rely on the performance of 4KB IO, since 4KB is mostly synchronous IO, which blocks the application until it completes.
- **Randomness**: Sequential IO constitutes 80% of all writes. This is because IOs from multimedia files and half of EXT4 *Journal* are of very large IO size.
- **Buffered write**: Synchronous IOs from applications account for 70% of all writes. However, buffered IOs are much larger, because these usually are IO accesses of large multimedia files.

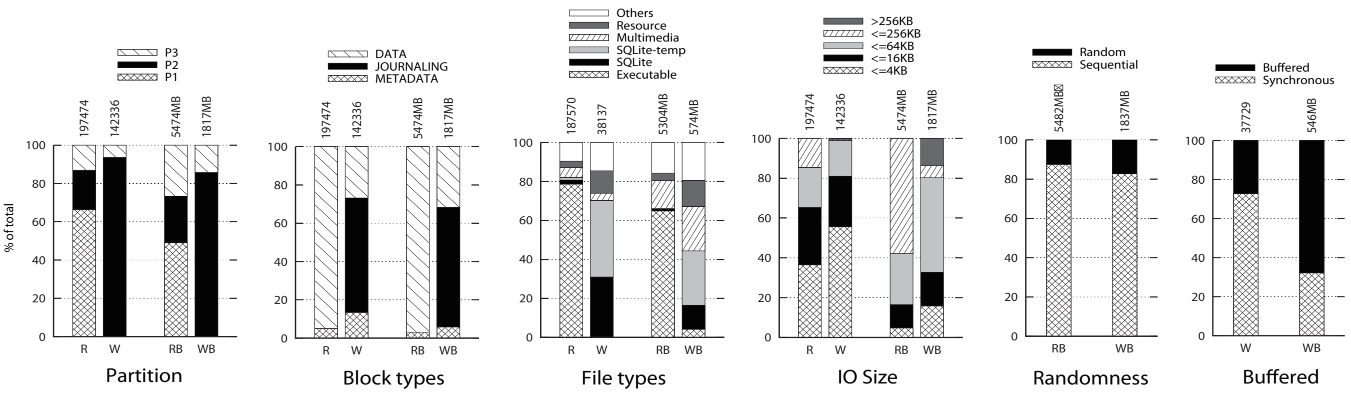

Figure 11: Daily block IO characteristics. The numbers at the R and W bars indicate the number of IOs, and the numbers at the RB and WB bars indicate the total bytes accessed for R (Read) and W (Write), respectively.

7. DISCUSSION

We summarize the technical issues relevant to the current IO subsystems and possible directions for improvement of future Android-platform filesystem and storage design.

Smart layers and Dumb result: SQLite adopts journaling to preserve the integrity of information. It creates a temporary journal file for each transaction. The EXT4 filesystem adopts journaling to maintain filesystem consistency and to achieve fast crash recovery. Each of these techniques is sophisticated and mature. However, as we observed, when these two are combined, they interact in unexpected ways, generating excessive EXT4 *Journal* block accesses. SQLite and EXT4 should be integrated in a rational manner so that duplicate operations can be eliminated.

Efficient handling of Synchronous write is critical: To improve IO performance, a number of cache-replacement algorithms have been developed for NAND flash storage devices [18, 14]. These algorithms work via exploitation of the temporal locality and asynchronous nature of IO operations. In the Android platform, however, neither of these hold: Reads are mostly for cold blocks, and writes are mostly synchronous. Further, write is at least ten times slower than read. In the Android platform, efficient handling of synchronous write operation, certainly, demands more attention. The recent proposal for a polling-driven IO [25] represents a good alternative for improvement of synchronous IO performance.

IO access pattern in smartphone is potpourri: The smartphone is a multi-purpose device. In addition to legacy phone functions, which include management of contacts, schedules and text messaging, it is used as a camera, music player, web browser and game console, among others. Accordingly, the IO access pattern in smartphones is a mixture of a wide variety of different access characteristics. IO accesses to the `/system` are mostly read, while IO accesses to the `/data` are mostly write. Accesses to the `/sdcard` entails large IOs, which accompany significant numbers of random writes caused by FAT updates.

File type information should be exploited by underlying storage and filesystem: Different types of files have unique characteristic accesses. Accesses to executable files are mostly large reads. Accesses to SQLite are mostly 4KB random. SQLite temporary files are only *written* with 4KB IO, and are short-lived. By examining the file type, that is, the extension, we can easily predict the characteristics of incoming IO. This information can be effectively exploited by prefetch strategies [15]. Interestingly, the recently proposed eMMC interface standard [9] allows the host to inform the eMMC of details on data blocks being transferred.

8. RELATED WORK

IO characterization studies in desktop and enterprise server environments have been conducted for several decades and have attained sufficient maturity. Despite the fact that some papers are now decades old, they still provide important guidelines to the understanding the intrinsic behavior of IO workloads. Riska et al. [20] studied the characteristics of disk-drive workloads in three different computing environments: enterprise, desktop, and consumer electronics. They showed that the access pattern for an enterprise server is more random than for a desktop one. Zhou et al. [26] found that the read/write ratio in the filesystem is 80%/20%, and that the majority of IO operations are random. Ruemmler et al. [22] analyzed disk IO in three different HP-UX systems. Their research showed that a majority of IO operations are writes and that the majority of writes (67-78%) are to *Metadata*, user-data IOs representing only 13-41% of all accesses. Roselli et al. [21] analyzed filesystem traces in a variety of different environments, including both UNIX and NT systems. They found that file access has a bimodal distribution pattern: some files are written repeatedly without being read, whereas other files are almost exclusively read.

The recent and rapid proliferation of NAND flash-based storage devices necessitates thorough understanding of the block level access characteristics of SSD [19, 12]. The following studies examined the temporal, spatial, and frequency aspects of the block-access trace, and exploited findings for devising various FTL algorithms (e.g. hot/cold identification, wear-leveling, a hybrid FTL log block management scheme, etc.).

Harter et al. [13] studied the IO behavior of the Mac OS filesystem. They showed that due to the complex XML-based document format, sequential IO on a file rarely results in sequential IO on a block device. Whereas it has generally been believed that in smartphones, the speed of the air link interface is a bottleneck to overall performance, it was recently found that the performance of smartphone applications are governed not by the communication speed of the air link but rather by storage performance [16]. It is of the utmost importance that a firm understanding of the ways in which newly emerging applications in smartphones use storage devices, which is to say, application-specific block-access characteristics. The result of our study provide explanations for the phenomenon observed in Kim et al. [16] and an important direction for the future filesystem development for smartphones.

9. CONCLUSIONS

We studied the Android smartphone's storage IO characteristics using Mobile Storage Analyzer (MOST). Our analysis revealed unique smartphone IO characteristics. These include the partition management strategy, the dominance of SQLite files, the excessiveness of *Journal* block accesses, the limited number of file types incurring block IO, and the low usage of buffered IO. We discovered that the IO subsystem and filesystem designs of the current state-of-the-art smartphone leave much to be desired in terms of fully exploiting the potential of the underlying NAND storage. Guaranteeing integrity from SQLite and EXT4 is a very complex undertaking. However, they need to be optimized in an integrated manner, so that redundant efforts are eliminated. The filesystem affords in-depth knowledge on the access characteristics of individual block IOs, which can be effectively exploited by the NAND storage controller. It is important that any modern NAND storage controller interface adopts a rich set of interfaces for sharing of valuable information between host and storage devices.

10. ACKNOWLEDGEMENTS

This work was supported by IT R&D program MKE/KEIT [No. 10035202, Large Scale hyper-MLC SSD Technology Development] and [No. 10041608, Embedded System Software for New-memory based Smart Device].

11. REFERENCES

[1] Mobile storage analyzer (most). `http://dmclab.hanyang.ac.kr/sub/main_most.htm`.

[2] Nexus one (google/htc). `http://en.wikipedia.org/wiki/Nexus_One`.

[3] Nexus s (google/samsung). `http://www.google.com/phone/detail/nexus-s`.

[4] Samsung galaxy s2. `http://www.samsung.com/global/microsite/galaxys2/html/`.

[5] Sqlite's use of temporary disk files. `http://www.sqlite.org/tempfiles.html`.

[6] Universal flash storage (ufs). `http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs`.

[7] What is android? `http://developer.android.com/guide/basics/what-is-android.html`.

[8] World mobile phone market 2010 to 2011. `http://www.yanoresearch.com/press/pdf/709.pdf`.

[9] Embedded multi-media card(e-mmc), electrical standard (4.5 device), June 2011.

[10] M. Arlitt and C. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. on Networking (ToN)*, 5(5):631–645, 1997.

[11] J. Axboe and A. D. Brunelle. Blktrace user guide, 2007.

[12] F. Chen, D. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 181–192. ACM, 2009.

[13] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of apple desktop applications. In T. Wobber and P. Druschel, editors, *SOSP*, pages 71–83. ACM, 2011.

[14] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. Fab: Flash-aware buffer management policy for portable media players. *Consumer Electronics, IEEE Trans. on*, 52(2):485–493, 2006.

[15] Y. Joo, J. Ryu, S. Park, and K. G. Shin. FAST: Quick application launch on solid-state drives. In *Proc. of the 9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February, 2011.*

[16] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proc. of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February, 2012.*

[17] S. Lee, B. Moon, and C. Park. Advances in flash memory ssd technology for enterprise database applications. In *Proc. of the 35th SIGMOD international conference on Management of data*, pages 863–870. ACM, 2009.

[18] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. Cflru: a replacement algorithm for flash memory. In *Proc. of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241. ACM, 2006.

[19] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pages 1 –7, 17-17 2008.

[20] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proc. of the USENIX Annual Technical Conference, General Track*, pages 97–102. USENIX, 2006.

[21] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the 2000 USENIX Annual Technical Conference*, pages 41–54, Berkeley, CA, June 18–23 2000.

[22] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proc. of Winter USENIX*, pages 405–20, 1993.

[23] K. Sovani. Linux: The journaling block device. `http://kerneltrap.org/node/6741`, June 20, 2006.

[24] T. Ts'o. Debugfs. `http://linux.die.net/man/8/debugfs`.

[25] J. Yang, D. Minturn, and F. Hady. When poll is better than interrupt. In *Proc. of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February, 2012.*

[26] M. Zhou and A. Smith. Analysis of personal computer workloads. In *Proc. of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS*, pages 208 –217, 1999.