Future Internet **2013**, *5*, 591-610; doi:10.3390/fi5040591

OPEN ACCESS

future internet

ISSN 1999-5903

www.mdpi.com/journal/futureinternet

Article

Framework for Analyzing Android I/O Stack Behavior: From Generating the Workload to Analyzing the Trace*

Sooman Jeong [1], Kisung Lee [2], Jungwoo Hwang [1], Seongjin Lee [1] and Youjip Won [1,*]

[1] Department of Computer and Software, Hanyang University, Seoul 133-791, Korea; E-Mails: 77smart@hanyang.ac.kr (S.J.); tearoses@hanyang.ac.kr (J.H.); insight@hanyang.ac.kr (S.L.)

[2] Samsung Electronics, Suwon 443-742, Korea; E-Mail: kiras.lee@samsung.com

* Author to whom correspondence should be addressed; E-Mail: yjwon@hanyang.ac.kr; Tel.: +82-2-2220-0579; Fax: +82-2-2281-9912.

Received: 18 October 2013; in revised form: 12 November 2013 / Accepted: 3 December 2013 / Published: 13 December 2013

Abstract: The existing I/O workload generators and trace capturing tools are not adaptable to generating nor capturing the I/O requests of Android apps. The Android platform needs proper tools to capture and replay real world workload in the Android platform to verify the result of benchmark tools. This paper introduces Android Storage Performance Analysis Tool, AndroStep, which is specifically designed for characterizing and analyzing the behavior of the I/O subsystem in Android based devices. The AndroStep consists of Mobibench (workload generator), MOST (Mobile Storage Analyzer), and Mobigen (workload replayer). Mobibench is an Android app that generates a filesystem as well as SQLite database operations. Mobibench can also vary the number of concurrent threads to examining the filesystem scalability to support concurrency, e.g., metadata updates, journal file creation/deletion. MOST captures the trace and extracts key filesystem access characteristics such as access pattern with respect to file types, ratio between random *vs.* sequential access, ratio between buffered *vs.* synchronous I/O, fraction of metadata accesses, *etc.* MOST implements reverse mapping feature (finding an inode for a given block) and retrospective reverse mapping (finding an inode for a deleted file). Mobigen is a trace capturing and replaying tool that is specifically designed to perform the user experiment without actual human intervention. Mobigen records the system calls generated from the user behavior and sanitizes the trace for replayable form. Mobigen can replay this trace on different Android platforms or with different I/O stack configurations. As an

*Primitive version of this paper has been presented at the 1st European Workshop on Mobile Engineering (ME13) [1].

example of using AndroStep, we analyzed the performances of twelve Android smartphones and the SQLite performances on five different filesystems. AndroStep makes otherwise time consuming I/O stack analysis extremely versatile. AndroStep makes a significant contribution in terms of shedding light on internal behavior of the Android I/O stack.

Keywords: Android; workload generator; analyzer; replayer

1. Introduction

Recently, the number of Internet accesses through smartphones has surpassed the number of accesses through desktop PCs [2]. This implies that a smartphone is no longer an office assistance tool but an essential device in everyday life. It is important to note that, as of 2012, the market share of Android platform has reached about 60% of the global smartphone operating systems [2]. This fact asserts that improving the performance of Android based smartphones is not trivial but a meaningful effort.

Kim *et al*. recently have shown that the storage I/O is one of the key factors that govern the overall system performance in smartphones [3]. There is much value in generating accurate workload to properly measure and analyze the I/O performance of Android based smartphones.

Android I/O stack consists of DBMS, file system, I/O daemon, and I/O scheduler. The Android platform uses SQLite [4], EXT4 [5], `mmcqd`, and CFQ scheduler [6] as DBMS, file system, I/O daemon, and I/O scheduler, respectively. Android applications use SQLite to maintain information in persistent manner. The default DBMS of Android, SQLite, generates 90% of the entire write operations in the Android platform [7,8]. Jeong *et al*. revealed that Android I/O behavior suffers from excessive inefficiency, which is caused by uncoordinated interaction between EXT4 and SQLite, known as Journaling of Journal [8]. They achieved 300% improvement in SQLite DB performance through proper optimization. Some of representative characteristics of I/O workload in Android are as follows: (i) 4 KB random write followed by `fsync()` and (ii) frequent creation and deletion of small (less than 12 KB) short-lived files.

The applications, often referred as "apps", in Android based smartphones, generate unique I/O requests. Existing I/O workload generators and trace capturing tools are not designed to generate nor to capture this unique I/O workload of Android apps. This paper introduces the Android Storage Performance Analysis Tool, *AndroStep*, which is specifically tailored for characterizing and analyzing the behavior of the I/O subsystem in Android based devices. AndroStep consists of workload generator (Mobibench), workload analyzer (MOST: Mobile Storage Analyzer), and workload replayer (Mobigen).

The remainder of the paper is organized as follows: Section 2 explains the existing workload generation tools and presents analysis of Android I/O workload. We introduce AndroStep, Android Storage Performance Analysis Tool, in Section 3. Section 4 presents the results of experiments with AndroStep. Section 5 concludes the paper.

2. Problem Assessment

2.1. I/O Characteristics of Android Based Devices

I/O characteristics of an Android based device [7] are different from those of a server [9] or a desktop [10,11]. In order to characterize the I/O behavior of Android I/O stack, we need right tools that (i) properly capture various attributes of the I/O behavior; (ii) generate the synthetic workload that represents essential behavior of the Android I/O subsystem; and (iii) replay the I/O trace to compare the performance of I/O subsystem in various Android devices. This section investigates whether existing tools are capable of capturing the Android I/O characteristics and reproducing them. This section further analyzes the limitations of the existing benchmarks. Through thorough analysis of the existing benchmark tools, we not only differentiated our tool from the existing ones, but also created a basis for implementing a benchmark for the Android-based platform.

There are many benchmark tools available for measuring the performance of file systems or storage devices; however, the existing benchmark tools for Android devices, such as IOzone [12], Bonnie++ [13], or AndroBench [14], can only measure limited level of Android resources.

One notable work on workload analysis of the Android based platform is done by Kim *et al*. [15]. They investigated which system services are used by an Android application and tried to allocate sufficient I/O bandwidth to the corresponding application, using the I/O bandwidth usage model. They classified applications into three classes: bursty, time-sensitive, and plain, and applied their I/O management usage model to media applications. However, their approach has two issues. First, they chose to model the I/O characteristics of a well-known system service instead of analyzing the I/O behavior of each application. Second, they neglected to model various scenarios within running an application and also did not consider that an application utilizes multiple system services.

There are four essential features that have to be considered in generating I/O workload for Android based devices. First, the workload generator must have a function to generate `write()` followed by `fsync()`, which is an essential characteristic of Android I/O. Second, the workload generator should be able to measure not only the performance of file I/O but also the performance of SQLite. Third, the workload generator must be able to run as an Android app, as well as a shell command. Finally, the tool must provide various options for file and database in measuring the performance of file I/O and DBMS.

sync函数同步内存中所有已修改的文件数据到储存设备。

2.2. Issues with Existing Tools

IOzone [12] and Bonnie++ [13] are widely used benchmark tools for measuring the performance of file systems. There are a few issues in IOzone and Bonnie++ that make them unsuitable for the Android I/O subsystem.

First, IOzone and Bonnie++ do not provide an option to change the synchronization mode of a write operation, e.g., `fsync()` or `fdatasync()`, which is one of the most important I/O operations performed by SQLite [8]. SQLite calls `fsync()` or `fdatasync()` to make rollback journal update and database update persistent. IOzone offers an option to `fsync()` the page cache only when the file is `closed()` or the buffered write completes.

Second, IOzone and Bonnie++ cannot measure the performance of SQLite, a default DBMS in the Android system. One needs to use separate application to measure the performance of database operations such as insert, update, or delete. We provide two versions of the workload generator: as an Android app and as a shell command.

The third issue concerns the testing method. In IOzone, any type of benchmark test, e.g., random write, random read *etc*., is preceded by creating a file of a given size, which not only is time consuming but also reduces the NAND flash life time significantly. To warrant the validity of the benchmark result, the same benchmark is often repeated multiple times. The overhead of creating the file in each run is redundant and can be significant especially when the file size is large, e.g., a few tens of GBytes.

AndroBench [14] is a file benchmark tool that runs in the Android system. It only offers `O_SYNC` as a synchronization option and does not support multi-threading benchmark environment.

2.3. Requirements for Characterizing Android I/O

Traditional I/O characterization study defines the I/O characteristics on four dimensions: I/O type (read *vs*. write), I/O size (KB), spatial locality (sequential *vs*. random), and temporal locality (hot *vs*. cold). To properly understand the I/O characteristics of Android platform, one needs to acquire the above mentioned four characteristics under various different contexts: subject to file types, filesystem block types, e.g., metadata, data, or journal, and Android apps, e.g., Facebook, Youtube, *etc*. For example, we need to understand what fraction of writes is synchronous for filesystem journal writes.

However, as far as we are aware of, the existing tools that capture and analyze workload are not suitable to study the details of I/O behavior in the Android platform because the tools do not identify the file type nor the application that issued an access for a given block. To provide remedy for these issues, we implemented the following two features:

- Identify the file type for a given block. We define six file types: SQLite DB, SQLite Journal, executable, multimedia, resources, and other files. From the logical block address captured by block access trace tool [16], we identify the type of a file to which the respective block belongs. This process consists of two steps. First, from a logical block address, the proposed tool identifies the inode number of the file to which the logical block address belongs. Then, from the inode number, it identifies the file type of the respective file;
- Identify the application for a given block. The proposed tool can identify the application that originally issued an I/O request. Different from legacy Linux kernel, Android I/O subsystem delegates the handling of the I/O requests to special daemon, `mmcqd`. From the block device layer, `mmcqd` is viewed as an application for a given I/O request. We need to find the original application for a given I/O, not `mmcqd`.

2.4. Replaying the Real World Traces

It is difficult to have complete confidence in the performance results obtained from using the synthetic workload. There are a number of works that address the limits of the synthetic workload [17–21]. These works exploit two approaches in capturing and replaying the system calls of an application. First approach is to capture the system calls at user-level and replaying them with timing accuracy [17,19–21].

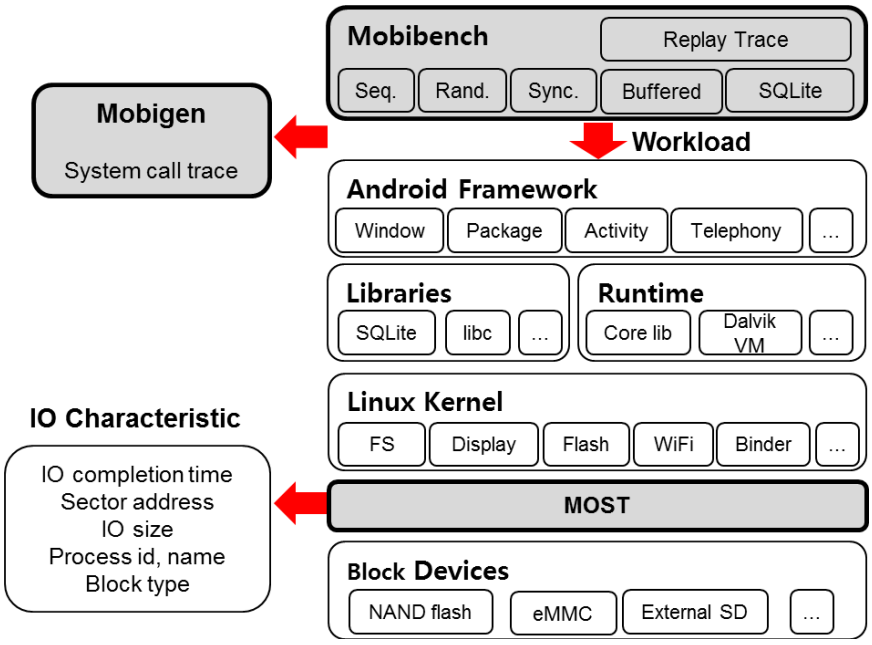
The other approach is to capture and replay the system calls at kernel level, such as VFS layer [18]. Capturing and replaying the system calls at user-level significantly reduces the programming overhead of implementing an additional I/O stack in VFS layer. However, it requires fastidious work to issue I/O requests in the exact time of the trace. In contrast to the user-level approach, the kernel level approach reduces processing overhead but it requires an additional layer to process the trace. It also has an inherent issue of kernel dependency.

Two most important factors that should be considered in implementing a tool that captures and replays real world workload are time accuracy in replaying the trace and proper handling of I/Os generated in multi-thread environment. I/Os issued in multi-thread environment need special attentions in locks and scheduling overhead while replaying the trace. Anderson *et al.* [17] proposed to exploit bypassing locking and pre-spin scheme to increase the timing accuracy. Mesnier *et al.* [19] addressed the dependency issues between resources while replaying the trace captured in multi-thread environment. In order to properly address the performance of real world workload, one needs to take aforementioned factors into account in capturing and replaying the I/Os of real world applications.

3. AndroStep

Android Storage Performance Analysis Tool consists of a workload generator called Mobibench (Mobile Benchmark) [22], trace capture tool called MOST (Mobile Storage Analyzer) [23], and Mobigen (Mobile Real Workload Generator) [24]. Figure 1 illustrates the structure of AndroStep.

Figure 1. Structure of AndroStep (Mobibench, MOST, and Mobigen).



3.1. Mobibench

Mobibench is a benchmark tool specifically designed for simulating the I/O characteristics of the Android applications. Mobibench is capable of measuring the performance of file I/Os and SQLite operations. Mobibench is implemented in two versions, one as a shell application and the other as an Android application ("the app"). Both versions use the same measurement engine written in C language. We use JAVA Native Interface (JNI) for app based Mobibench to call the functions implemented in C. As an Android app, the benchmark can use the shared SQLite library which is prelinked by the phone vendor. To run the benchmark as a shell command, we need to statically link the SQLite library to the benchmark program. Smartphone vendors put significant effort in optimizing the SQLite. SQLite is published under Apache License and the vendors do not have an obligation to disclose the modified source code. We are not aware of any phone vendors who disclose production SQLite code. Shell command based benchmark can only use stock SQLite which is publicly available. It should be noted that in most smartphones, the performance results of the app based benchmark and shell command based one rarely coincide.

If an application is compiled with shared library, the application exploits SQLite shared library that is preloaded by the smartphone vendor. Since the manufacturers provide the optimized SQLite library, it is possible to acquire an accurate and optimized performance of SQLite operations using the app. On the other hand, if the application is compiled with static library, the application exploits SQLite static library. Since Mobibench is a unified measurement engine, it allows to test and compare the performances in diverse systems.

Table 1 illustrates comparisons of three benchmarks: Mobibench, IOzone, and AndroBench. Mobibench provides detailed benchmark configuration options such as choice of a partition, the number of threads, and workload characteristics. Mobibench uses one of `/data` or `/sdcard` in internal storage and `/extSdcard` in external memory card. Mobibench allows configuring the number of threads in order to provide an environment similar to smartphones, where multiple threads execute I/Os and SQLite operations simultaneously.

In Mobibench, one can generate two types of workloads: file operations and database operations. Mobibench specifies spatial locality (random *vs.* sequential), I/O mode (read *vs.* write), file size, I/O unit size, and synchronization mode. There exist five synchronization modes: buffered, synchronous, direct, mmap, and `write()+fsync()`.

Database operations include insert, update, and delete. Performances of these operations vary widely depending on compile and PRAGMA options of SQLite. PRAGMA command is used to set the operation modes of the SQLite library such as SQLite synchronization and journal modes. The performance of SQLite varies significantly subject to its journal mode. Mobibench can be built to use either stock SQLite library or prelinked SQLite library provided by the phone manufacture.

Mobibench generates three performance values: throughput, CPU utilization, and the number of context switches. In file I/O test, units of throughput are "KB/s" for sequential operations and "IOPS" for random operations. Unit of throughput in SQLite operation is "transaction/ sec". Utilization of CPU distinguishes ACTIVE, IDLE, and IO-WAIT to understand how the test utilizes the CPU. Mobibench also counts the number of context switches to measure the context switch overheads.

Table 1. Functional comparison.

	Function	Mobibench	IOzone [12]	AndroBench [14]
Workload	sequential write	O	O	O
	sequential read	O	O	O
	random read	O	O	O
	random write	O	O	O
	`write()+fsync()`	O	X	X
	multi-thread	O	O	X
	SQLite operation	O	X	O
Output	throughput	O	O	O
	CPU utilization	O	O	X
	number of context switch	O	X	X
Options	exe environment	shell/app	shell	app
	separate file I/O operation	O	X	X
	separate SQLite operation	O	X	X
	file sync mode	O	O	X
	SQLite journal mode	O	X	X

Figure 2. Mobibench application. (**a**) Measure tab; (**b**) Setting tab.

(**a**) (**b**)

Figure 2 illustrates screen snapshots of Mobibench. There are three tabs in Mobibench. In Measure tab, there are four buttons: ALL, File I/O, SQLite, and Custom. File I/O runs sequential or random read/write operations. SQLite runs transactions of insert, update, and delete DB operations. "All" option executes both file operations and SQLite operations. "Custom" option allows the user to tailor various options for workload generation. Mobibench displays a progress bar to show the status of the running test and illustrates the results upon completion of the test.

We maintain Mobibench server which collects the result of Mobibench runs from the smartphone through the Internet. Mobibench server keeps a record of ranks of submitted performance measurements;

this allows comparisons between various Android devices deployed worldwide. Figure 3a shows a screen snapshot of the performance statistics of different smartphones. The results are submitted to the server only when the user agrees. Mobibench also can share the performance results with others via SMS/Mail/Facebook/Twitter. Figure 3b illustrates the rank of sequential writes performances among the seven experiment modes. Each mobile device is identified by the device name and the Android version. If there were same devices with different Android versions, we considered them different devices.

Figure 3. Function of data collection. (**a**) Result screen; (**b**) Ranking.

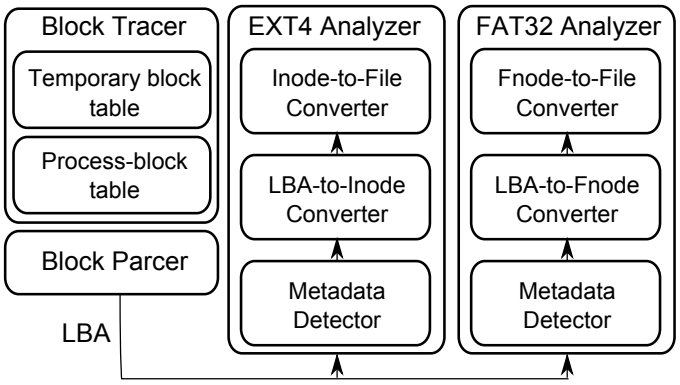
(**a**) (**b**)

3.2. MOST: Mobile Storage Analyzer

Mobile Storage Analyzer (MOST) consists of (i) modified Linux kernel; (ii) a block analyzer which is used to extract the semantic information of a given block; and (iii) block trace utility, `blktrace`. To obtain correlational information among different attributes of an I/O, *i.e.*, file type *vs.* randomness, block type *vs.* synchronization mode, it is mandatory that we acquire comprehensive I/O attributes, *i.e.*, process ID, type of a block in the filesystem, type of a file, *etc.*, for a given LBA. Modern I/O subsystem consists of a set of layers and each layer communicates with others via very well-defined narrow interface. It is not possible to access the information in the other layers. For example, it is not possible to determine whether a given block belongs to SQLite rollback journal or SQLite database table by examining the logical block address. Figure 4 illustrates the overall organization of MOST.

MOST provides three features: (i) finding a file type for a given block; (ii) finding an application which accesses a given block; and (iii) finding a file for a given block even when the file has been deleted. The capability of finding a "deleted" file for a given block is essential in studying the I/O characteristics in Android since in "Delete" journal mode, SQLite deletes its rollback journal file when the transaction completes.

Figure 4. Mobile storage analyzer.



The first feature is block-to-file mapping. MOST can reverse-map the disk block to the respective file to which it belongs. It accepts a logical block number as an input and generates a file name. MOST uses `debugfs` [25] to reverse-map the block in the EXT4 file system and we developed a module to reverse-map the block for FAT32 filesystem [23].

The second feature is block address to process ID mapping. MOST can identify the original process that issued a given I/O. In Android, `mmcqd` daemon manages the mmc card device driver and is responsible for issuing all block I/Os. For each block access, `blktrace` records the process ID for each block. When `blktrace` is used in Android I/O stack, `blktrace` designates `mmcqd` as the process for all I/O accesses. The goal of MOST is to identify the application, app, that has issued a given I/O. For this purpose, we modified the Linux kernel. We added a block address to process ID table in the kernel which maintains a process ID for a given LBA. The entries in the table are <LBA, process ID>. When the I/O scheduler inserts the I/O request into the queue, MOST inserts the <LBA, process ID> information into an entry of the mapping table. MOST examines table posthumously to retrieve ID of the original process that issued a given LBA access.

MOST allows retrospective LBA mapping. In Android, we found that many files are short-lived; they are created and quickly deleted by SQLite. This is due to the ”DELETE” rollback journal mode of SQLite. Although they have short life span, these files are `fsync()` to NAND storage. When MOST initiates analysis procedure for a given LBA, a temporary file to which the block belonged may have been deleted and cannot be found. To address this issue, MOST maintains a table for logical block address and inode number pair in the Linux kernel. When the I/O scheduler inserts the I/O request to the queue, MOST inserts <LBA, inode number> entry to the table. MOST examines this table to obtain the file information for a given logical block address. To reduce the table size, MOST maintains an <LBA, inode number> entry only for temporary files, that is, when the file extension is either `.db-journal`, `.db-mjxxxx`, `.bak`, or `.tmp`.

MOST categorizes logical blocks into three types: *metadata*, *journal*, and *data*. In the EXT4 file system, *metadata* blocks include a superblock, group descriptor, data block bitmap, inode bitmap, and inode table. In the FAT32 file system, *metadata* blocks include a boot record and File Allocation Table (FAT). *journal* is a journal block of the EXT4 file system. *data* blocks are file data and directory entries. Table 2 illustrates the output format from MOST.

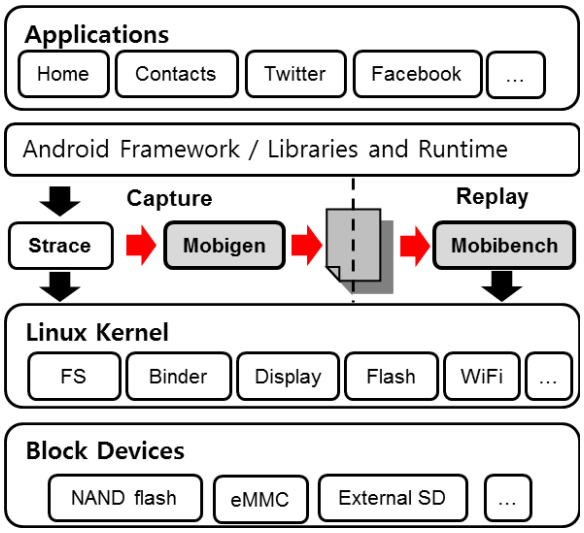
Table 2. Output of Mobile Storage Analyzer.

1	I/O completion time
2	Flags for read and write
3	Sector address and I/O size
4	Process ID and process name
5	Block type: *metadata*, *journal*, or *data* block
6	File name in case of *data* block

3.3. Mobigen: Mobile Real Workload Generator

Although Mobibench is a powerful micro-benchmark tool that allows generating I/O workload of Android based smartphones, its workload is *synthetic*. In order to properly analyze the performance behavior of a given system configuration, it is mandatory to analyze the system behavior under human generated workload. One of the main difficulties in using human generated workload is the ability to replay. Reproducing identical human behaviors across different sets of experiments is not a trivial task. We developed a tool called *Mobigen* to address this issue. Mobigen and Mobibench work in a collaborative manner. Mobigen records system calls using `strace` [26] and processes them to be replayed by Mobibench. Figure 5 shows how these two tools operate. Mobigen is written in ruby [27].

Figure 5. Structure of Mobigen.



There have been a number of works on capturing and replaying the I/O traces [17–21]. To the best of our knowledge, our work is the first to attempt capturing as well as replaying the I/O trace of Android applications.

Mobigen is designed to record and to replay the system call trace issued by Android applications. Mobigen consists of two components: trace collector and trace cleaner. Mobigen uses existing *strace* [26] in acquiring the system call trace. Trace cleaner is the key component of Mobigen; it creates the system call trace file in a format which can be recognized and can be replayed by Mobibench.

Trace cleaner performs three major roles. First, it transforms the acquired trace into *closed* set of system call trace. Trace cleaner scans the system call trace and identifies the read/write system calls that

are not preceded by `open()` system call. Then, it inserts the `open()` system call before the `write()` system call. Second, in an effort to reduce the processing overhead of Mobigen, trace cleaner simplifies the parameters of the workload such as file path, open flag, and removal of unnecessary parameters. Through the simplification, we were able to issue I/Os in a time accurate manner. As a result of multi-threaded environment, a single system call can be divided into two system calls due to the interruption by another system call. Trace cleaner merges these system calls and reduces time of parsing redundant system calls in replay engine.

The trace cleaner not only analyzes thread information from the acquired system call trace but it also removes platform specific system calls to improve the timing accuracy of the issued I/Os and to allow synchronizing the I/Os in multi-thread environment. We used pthread library to replay the I/Os generated in multi-thread environment. We designed a replay engine that allows each thread to issue respective I/Os of captured trace according to its respective time.

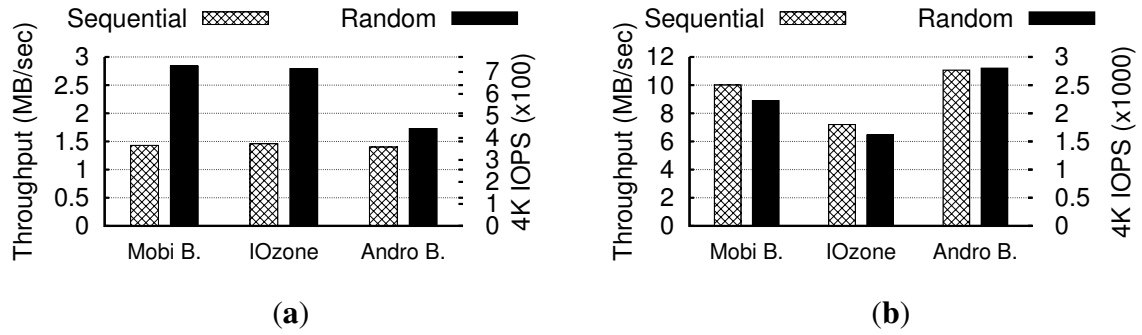
4. Experiment

We verified the performance result of Mobibench against IOzone and AndroBench. We used MOST to analyze the I/O characteristics of real world Android applications. All experiments were conducted on the Galaxy S4 [28] (Samsung 1.6 GHz Octa-core, 2 GB RAM, 32 GB eMMC, Android 4.2.2 with Linux kernel 3.4.5).

4.1. Accuracy

The primary feature of Mobibench is to measure the performance of file operation and the performance of database operation. The accuracy of performance measurement should be guaranteed. We extracted the performance of file I/O using Mobibench, IOzone, and AndroBench. Figure 6 illustrates the result.

Figure 6. I/O on internal eMMC. (**a**) Synchronous write; (**b**) Direct read. Filesize: 512 MB; I/O-size: 4 KB [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].

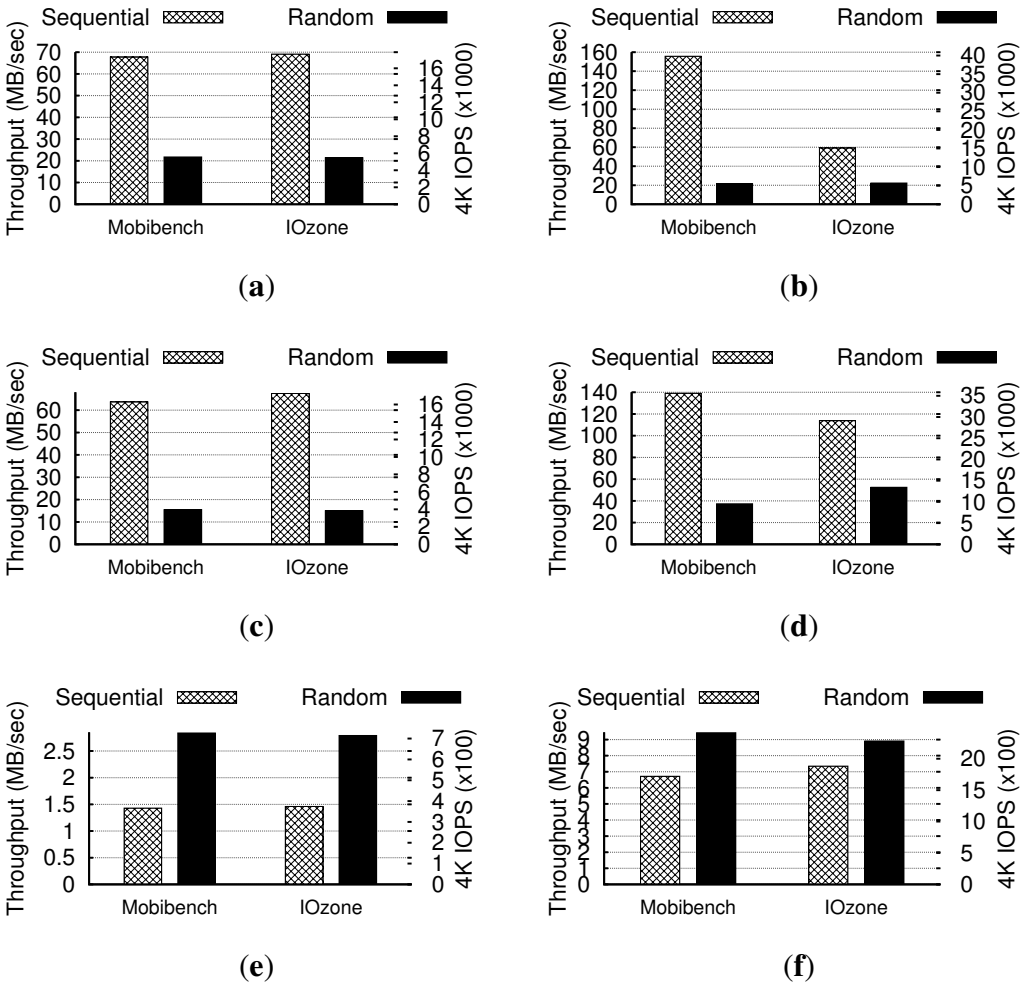


(**a**) (**b**)

File size and I/O size in the experiment were set to 512 MB and 4 KB, respectively. Mobibench and IOzone show similar throughput and IOPS on sequential and random operations, respectively. Random write performances observed in Mobibench and IOzone differ only by 2%.

Mobibench and IOzone support various file open modes. We compared the two benchmarks using six different I/O modes: buffered read/write, mmap read/write, synchronous write, and direct write. File size and I/O size were set to 512 MB and 4 KB, respectively. We measured the performances of sequential and random I/O on all six modes. The performance results from Mobibench are similar to the results from IOzone (Figure 7). However, sequential read performances of Mobibench in buffered and mmap mode are about 20% to 160% higher than those of IOzone. It is because IOzone calls `fsync()` at the end of the run.

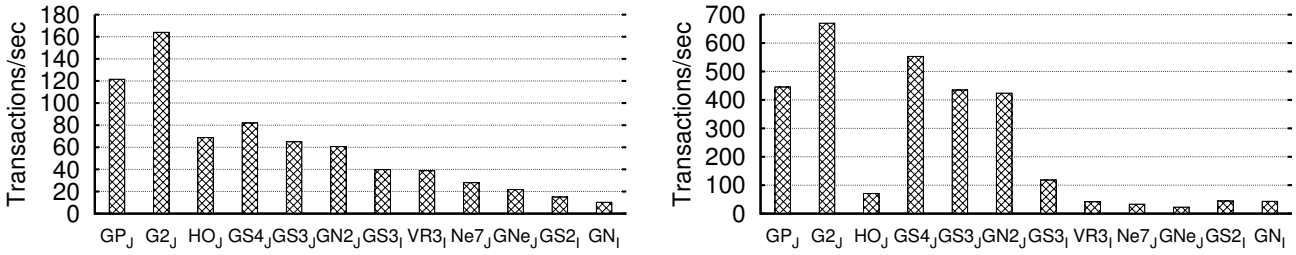
Figure 7. Mobibench *vs.* IOzone, I/O on internal eMMC. (**a**) Buffered write; (**b**) Buffered read; (**c**) mmap write; (**d**) mmap read; (**e**) Synchronous write; (**f**) Direct write. Filesize: 512 MB; I/O-size: 4 KB [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].



Next, we measured the performance of SQLite operation using Mobibench with various smartphone models and Android versions. We used TRUNCATE journal mode for SQLite to measure the performance. The presented results are averages of 10,000 runs. Figure 8 shows the insertion and update performances of SQLite. We have tested a total of twelve smartphone models. We used two different versions of Android (Jelly Bean and Ice Cream Sandwich) on Samsung Galaxy S3. Most of Jelly Bean loaded devices exhibited better performance than Ice Cream Sandwich loaded devices. There are two reasons for the performance difference. The first reason is the hardware difference. Jelly Bean loaded

smartphone models are equipped with faster CPU and faster eMMC (NAND storage for smartphone). The second reason is the software optimization. We loaded the Jelly Bean and Ice Cream Sandwich on the same smartphone device, Galaxy S3 and examined the performance. Jelly Bean loaded GS3 shows approximately 150% better performance than Ice Cream Sandwich loaded GS3 (63 insertions/sec *vs*. 40 insertions/sec).

Figure 8. SQLite insert and update performance on twelve smartphone models from different smartphone manufacturers. (**a**) Insert; (**b**) Update. SQLite journal mode: TURNCATE. GP: LG Optimus Pro, G2: LG Optimus G2; HO: HTC ONE; GS4: Samsung Galaxy S4; GS3: Samsung Galaxy S3; GN2: Samsung Galaxy Note2; VR3: Pantech Vega R3; Ne7: Google Nexus 7; GNe: Samsung Galaxy Nexus; GS2: Samsung Galaxy S2; GN: Samsung Galaxy Note; Subscript i and j denote the versions of Android: Ice Cream Sandwich (4.0.4) and Jelly Bean (4.2.x, 4.3.x), respectively. (`/data` partition, EXT4).



(**a**) (**b**)

On the other hand, two Jelly Bean loaded smartphones, Nexus 7(Ne7$_j$), and Galaxy Nexus(GNe$_j$) show 20 insertions/sec and 10 insertions/sec, respectively. These performance numbers are much lower than the performance number of Ice Cream Sandwich loaded Galaxy S3(GS3$_i$), 40 insertions/sec. Mobibench analysis brings us two important findings. First, Android software stack does evolve from SQLite performance' perspective. Each upgrade in Android is known only for its new features, new UI/UX, and security enhancements compared to its predecessor but not much is being said on the performance optimization effort. We found that Linux kernel and SQLite are becoming more efficient as they proceed and SQLite exhibits 1.5× performance gain compared to its predecessor. Second, hardware inefficiencies can be offset by software optimization. On the same token, smartphone vendors need to use better hardware, e.g., CPU, DRAM, and eMMC, to overcome the inefficiencies in software.

4.2. New Features

Mobibench provides eight different synchronization options and can vary the number of threads. These features allow us to effectively tailor the benchmark workload for the target environment. We performed three different experiments in file I/O, SQLite, and multithreading degree. We examined the performance of 4 KByte random `write()` followed by `fsync()`. We varied journal and synchronization modes of SQLite and ran experiments on multithreading environment. We used the same devices and partitions as the previous section.

Figure 9 compares the results of `write()+ fsync()` against synchronous and direct write using file and I/O size of 512 MB and 4 KB, respectively. `write()+ fsync()` and synchronous write show similar performances on sequential workload. With random write workload, on the contrary, the number of I/Os caused by textttfsync() is increased. As a result, the performance of `write()+ fsync()` is about twice as slower than that of synchronous write.

Figure 9. `write()+fsync()`, I/O on internal eMMC. Filesize: 512 MB; I/O-size: 4 KB [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].

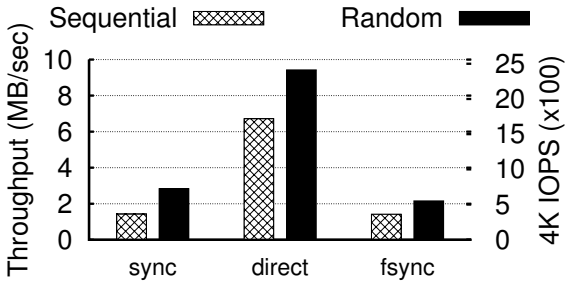
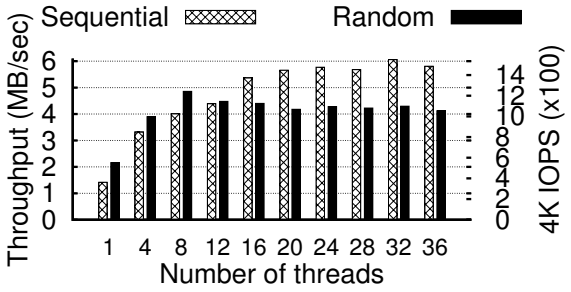


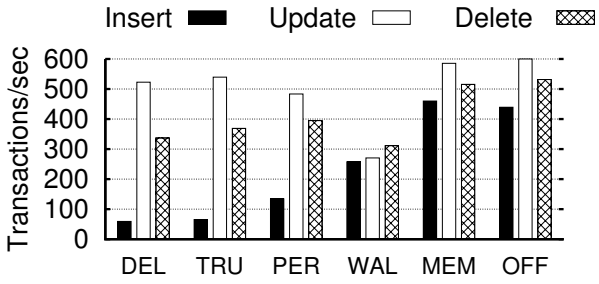
Figure 10 shows the performances of `write()+ fsync()` under varying number of threads. Note that all threads, which have equal file size, in an experiment sums up to the file size of 512 MB. As the number of threads increases, the performance also increases because Galaxy S4 is equipped with octa-core CPU; however, when there are more than 16 threads, the performance saturates because all H/W resources are exploited.

Figure 10. `write()+ fsync()` with multi-thread, I/O on internal eMMC. Filesize: 512 MB; I/O-size: 4 KB [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].



Mobibench offers an option to change SQLite journal and sync mode. There are six journal modes in SQLite: DELETE, TRUNCATE, PERSIST, WAL, MEM, and OFF. Excluding MEM and OFF journal modes, SQLite suggests that WAL journal mode shows the best performance [29]. In MEMORY mode, SQLite stores the journal information in system memory. OFF mode does not keep account of the journal. Figure 11 shows the results of average TPS of running 10,000 insert, update, and delete operations. Insert in WAL mode shows about 4 times better performance than in other journal modes. On the other hand, update in WAL mode shows slightly lower performance than in the other modes. Main reason behind this result is modified SQLite library in Galaxy S4. The manufacturer modified the library to use OFF mode in update operations on DELETE, TRUNCATE, and PERSISTENT modes. Using strace, we have verified that all except WAL mode operate like OFF mode in update operations.

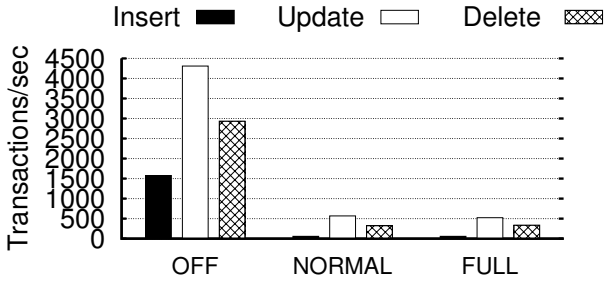
Figure 11. SQLite operation with various journal modes on internal eMMC. Sync mode: FULL; SQLite version: 3.7.5 [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].



Another mode that has noteworthy effect on performance of SQLite is sync mode. There are three sync modes in SQLite: FULL, NORMAL, and OFF. The number of times SQLite calls `fsync()` is determined by the mode. FULL mode calls `fsync()` on each transaction to guarantee that all the data is written to a storage device. NORMAL mode still calls `fsync()` at the most critical point, but less often than FULL mode. OFF mode does not call `fsync()` while processing a transaction.

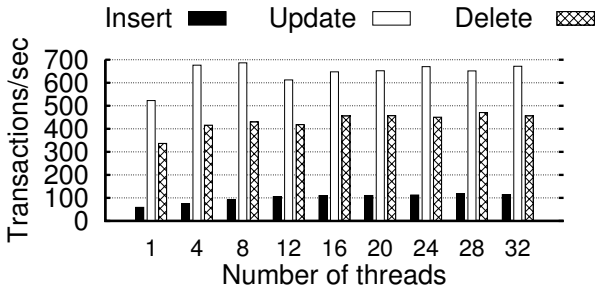
Figure 12 illustrates the results of three sync modes while running insert, update, and delete SQLite operations. It shows that using OFF mode speeds up the transactions greatly compared to NORMAL or FULL modes in all SQLite operations.

Figure 12. SQLite operation with various sync mode on internal eMMC. Journal mode: DELETE [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].



Mobibench also supports multi-threading. Figure 13 illustrates the effect of increasing the number of threads up to 32. We observed no further performance gain when the number of threads exceeds 12.

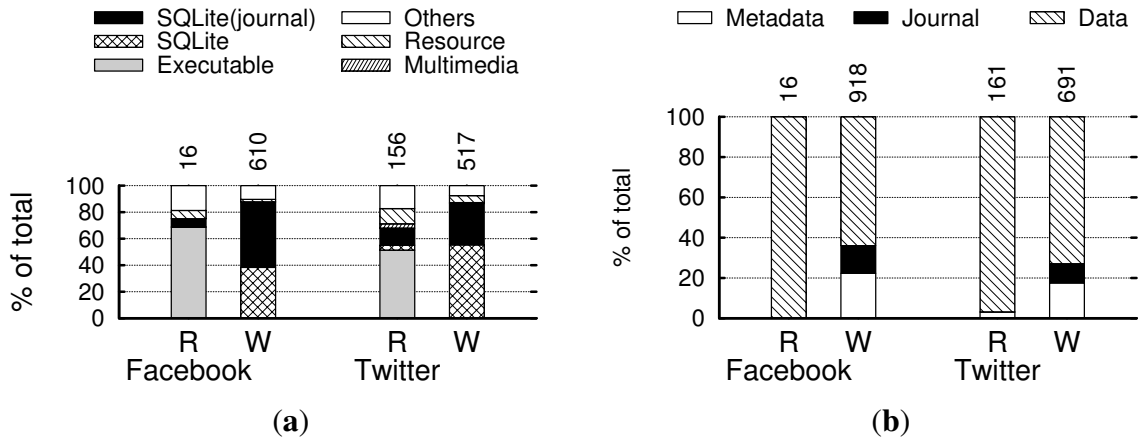
Figure 13. SQLite operation with multi-thread on internal eMMC. sync mode: FULL; journal mode: DELETE [Samsung Galaxy S4, Android 4.2.2 (JB), `/data` partition, EXT4].



4.3. I/O Characterization of Android Applications with MOST

In this section, we describe the analysis of I/O characteristics of real world Android applications. Among the applications we have analyzed, we describe the I/O characteristics of Facebook and Twitter. Figure 14a shows the distribution of file types for the I/Os directed to eMMC partition, which is analyzed with MOST. We grouped the files into six categories: SQLite database (`.db`), SQLite journal (`.db-journal`), executables (`.so`, `.apk`, and `.dex`), resources (`.dat` and `.xml`), multimedia, and others.

Figure 14. I/O distribution of file types and block types. (**a**) File types; (**b**) Block types. The number on the top of each bar indicates the number of respective block I/O for R (Read) and W (Write), respectively. [Samsung Galaxy S4, Android 4.2.2 (JB), EXT4].



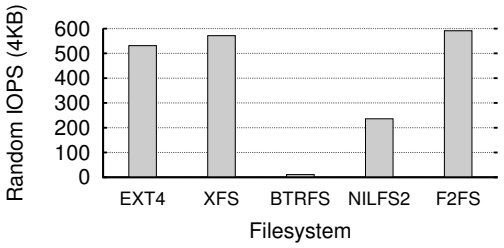
We found that SQLite and its journal files are responsible for approximately 90% of the write I/Orequests in both Facebook and Twitter applications. The reason for such high number of write I/O requests to the local storage is because Facebook and Twitter cache the timeline records in their SQLite table, which is common approach in modern SNS applications to improve the processing speed and to reduce the latency for user requests [30]. We also found that resources (`.dat` and `.xml`) and executable files are responsible for about 60% of the read I/O requests. Next, we categorized the blocks in the filesystem partition into three types: metadata, journal, and data. Figure 14b illustrates the result of file type distribution. Metadata and journal account for 10% and 20% of all write I/Os, respectively.

4.4. Trace Replay

Mobigen can be especially useful when one wants to re-examine the performance of a workload after making modifications to the file system or the block device layer.

We measured the performance of `write()` followed by `fsync()` on five different filesystems. We used EXT4, XFS, BTRFS, NILFS2, and F2FS filesystems. Figure 15 illustrates the performance results. F2FS yields approximately 10% performance gain against EXT4.

Figure 15. Random write using `fsync()` on internal eMMC. File size: 100MB; I/O size: 4 KB. [Samsung Galaxy S4, Android 4.2.2 (JB), `/cache` partition, EXT4].



Using Mobigen, we examined the performance of the different Android platforms in executing Facebook and Twitter. In this study, human user used the Facebook and Twitter for two minutes each in baseline platform (GS4 with EXT4). With Mobigen, we recorded the system calls generated while using these applications and replayed the captured trace under four different filesystems. For each two-minute run, we extracted the total I/O latency. In EXT4, the total I/O time was 3 s. With F2FS, the total I/O time reduced to 2.3 s. Similar degree of improvement was observed in Twitter experiment. Figure 16 illustrates the results. The objective of this experiment does not lie in analyzing the filesystem behavior. Rather, it is to show that Mobigen can greatly facilitate the performance experiment based on human generated workload without actual human intervention. Table 3 describes the basic statistics of trace acquired by post processing the I/O trace using Mobigen.

Figure 16. Comparison on execution-time of replaying script using Mobigen/Mobibench. (**a**) Facebook; (**a**) Twitter. [Samsung Galaxy S4, Android 4.2.2 (JB), `/cache` partition, EXT4].

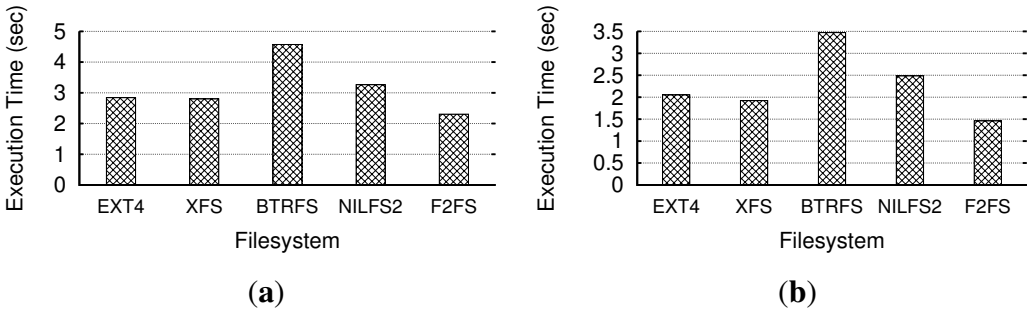


Table 3. Characteristics of captured trace using Mobigen.

Attributes	Facebook	Twitter
Runtime(sec)	120	120
Lines (strace out)	25,516	22,349
Lines (Mobigen out)	23,231	20,641
Number of threads	107	81
Write size(MB)	2.81	1.91
Read size(MB)	1.52	3.11

5. Conclusions

In this work, we developed a framework to analyze the behavior of the Android I/O stack. The framework has three major features: (i) generate I/O patterns that effectively capture the physical characteristics of Android I/O; (ii) capture the context dependent I/O characteristics in Android I/O stack (application, file system, and block device); and (iii) capture and replay real world applications using AndroStep.

AndroStep is a comprehensive set of tools which enable the user to overhaul the entire I/O stack of Android OS. It consists of workload generator called Mobibench which exports variety of options to generate workloads with different nature, trace analyzer called MOST which captures the block IO trace and analyzes its characteristics at filesystem level and application level, and Mobigen which collects the system call trace and replays them. Mobigen is designed to perform the human centered experiment without actual human intervention. AndroStep makes the study on the Android I/O stack extremely versatile. Mobibench is available at Google playstore. The source code of AndroStep is publicly available.

Acknowledgements

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

References

1. Jeong, S.; Lee, K.; Hwang, J.; Lee, S.; Won, Y. AndroStep: Android Storage Performance Analysis Tool. In Proceedings of the First European Workshop on Mobile Engineering ME13, Aachen, Germany, 26–28 February 2013; Volume 215.
2. Meeker, M. *2013 Internet Trends Report*; Kleiner Perkins Caufield & Byers: Snyder, TX, USA, 2013.
3. Kim, H.; Agrawal, N.; Ungureanu, C. Revisiting Storage for Smartphones. In Proceedings of the 10th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, 14–17 February, 2012.
4. SQLite Homepage. Available online: http://www.sqlite.org/ (accessed on 11 December 2013).
5. Mathur, A.; Cao, M.; Bhattacharya, S.; Dilger, A.; Tomas, A.; Vivier, L. The New ext4 Filesystem: Current Status and Future Plans. In Proceedings of the Linux Symposium, Ottawa, Canada, 27–30 June 2007.
6. Axboe, J. CFQ IO Scheduler. Presented at linux. conf. au, Sydney, Australia, 15–17 January 2007.
7. Lee, K.; Won, Y. Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone. In Proceedings of EMSOFT 2012 International Conference on Embedded Software, Tampere, Finland, 7–12 October 2012.
8. Jeong, S.; Lee, K.; Lee, S.; Son, S.; Won, Y. I/O Stack Optimization for Smartphones . In Proceedings of the USENIX Annual Technical Conference, San Jose, CA, USA, 26–28 June 2013.

9. Hsu, W.W.; Smith, A.J. Characteristics of I/O traffic in personal computer and server workloads. *IBM Syst. J.* **2003**, *42*, 347–372.
10. Zhou, M.; Smith, A.J. Analysis of Personal Computer Workloads. In Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS, College Park, MD, USA, 24–28 October 1999; pp. 208–217.
11. Harter, T.; Dragga, C.; Vaughn, M.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. A File is not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *SOSP*; Wobber, T., Druschel, P., Eds.; ACM: New York, NY, USA, 2011; pp. 71–83.
12. IOzone Filesystem Benchmark. Available online: http://www.iozone.org/ (accessed on 11 December 2013).
13. Coker, R. Bonnie++ File-System Benchmark. Available online: http://www.coker.com.au/bonnie++ (accessed on 11 December 2013).
14. Kim, J.M.; Kim, J.S. AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices. In *Frontiers in Computer Education*; Sambath, S., Zhu, E., Eds.; Springer: Berlin, Germany, 2012; Volume 133, pp. 667–674.
15. Kim, H.; Lee, M.; Han, W.; Lee, K.; Shin, I. Aciom: Application Characteristics-Aware Disk and Network I/O Management on Android Platform. In Proceedings of the International Conference on Embedded Software (EMSOFT), Taipei, Taiwan, 9–14 October 2011; pp. 49–58.
16. Axboe, J.; Brunelle, A.D. Blktrace User Guide. Available online: http://www.cse.unsw.edu.au/~aaronc/iosched/doc/blktrace.html (accessed on 11 December 2013).
17. Anderson, E.; Kallahalla, M.; Uysal, M.; Swaminathan, R. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In Proceedings of the 3rd USENIX Conference on File and Storage Technologies, San Fransisco, CA, USA, 31 March–2 April 2004; pp. 45–58.
18. Joukov, N.; Wong, T.; Zadok, E. Accurate and Efficient Replaying of File System Traces. In Proceedings of the 4th USENIX Conference on File and Storage Technologies, San Fransisco, CA, USA, 13–16 December 2005; p. 25.
19. Mesnier, M.P.; Wachs, M.; Sambasivan, R.R.; Lopez, J.; Hendricks, J.; Ganger, G.R.; O'Hallaron, D. Trace: Parallel Trace Replay with Approximate Causal Events. In Proceedings of the 5th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, 13–16 February 2007; p. 24.
20. Agrawal, N.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. Towards realistic file-system benchmarks with CodeMRI. *SIGMETRICS Perform. Eval. Rev.* **2008**, *36*, 52–57.
21. May, J. Pianola: A Script-Based I/O Benchmark. In Proceedings of the Petascale Data Storage Workshop, Austin, TX, USA, 17 November 2008; pp. 1–6.
22. Mobile Benchmark. Available online: https://github.com/ESOS-Lab/mobibench (accessed on 11 December 2013).
23. Mobile Storage Analyzer. Available online: https://github.com/ESOS-Lab/MOST (accessed on 11 December 2013).
24. Mobile Real Workload Generator. Available online: https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen (accessed on 11 December 2013).

25. Ts'o, T. Debugfs. Available online: http://linux.die.net/man/8/debugfs (accessed on 11 December 2013).
26. Strace: System Call Tracer. Available online: http://sourceforge.net/projects/strace/ (accessed on 11 December 2013).
27. Ruby: Ruby Script. Available online: https://www.ruby-lang.org (accessed on 11 December 2013).
28. Samsung Galaxy S4. Available online: http://www.samsung.com/global/microsite/galaxys4/ (accessed on 11 Decermber 2013).
29. Write-Ahead Logging. Available online: http://www.sqlite.org/wal.html (accessed on 11 Decermber 2013).
30. Huang, Q.; Birman, K.; van Renesse, R.; Lloyd, W.; Kumar, S.; Li, H.C. An Analysis of Facebook Photo Caching. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, PA, USA, 3–6 November 2013; pp. 167–181.