AndroStep: Android Storage Performance Analysis Tool

Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee and Youjip Won

Dept. of Electronics and Computer Engineering
Hanyang University, Korea
$\{77smart|kisunglee|tearoses|insight|yjwon\}$@hanyang.ac.kr

Abstract: The applications in Android based smartphones generate unique IO requests; however, existing IO workload generators and trace capturing tools are designed to neither generate nor capture the IO requests of Android apps. In this paper, we introduce the Android storage performance analysis Tool (AndroStep) which is specifically designed for characterizing and analyzing the behavior of the IO subsystem in Android based devices. AndroStep consists of workload generator, called Mobibench, and workload analyzer, called Mobile Storage Analyzer (MOST). Mobibench is an android App, which generates typical filesystem workloads, e.g., Random vs. Sequential and Synchronous vs. Buffered IO, as well as the most dominant workload in Android platform: SQLite insert/update and a write followed by `fsync()` call. Mobibench can also vary the number of concurrent threads to examine the storage and filesystem overhead to support concurrency, e.g., metadata updates, journal file creation/deletion. MOST capture the trace and extracts key filesystem access characteristics: access pattern with respect to file types, ratio between random vs. sequential access, ratio between buffered vs. synchronous IO, 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). We explain the structure and usage of AndroStep in detail. We verify performance result of Mobibench on eight smartphone models.

1 Introduction

NAND Flash based storage device such as eMMC card [eMM11] and μ-sdcard, is the most popular storage media for Android based devices, such as smartphone, smartTV, Smartpad, etc. Android IO stack consists of DBMS, file system, IO daemon, and IO scheduler. Android 4.0.4 (ICS) uses SQLite [SQL], EXT4 [MCB+07], mmqcd, and CFQ scheduler [Axb07] as DBMS, file system, IO daemon, and IO scheduler, respectively. Android application uses SQLite to maintain information in persistent manner. SQLite generates 80% of entire write operations in Android platform [LW12]. Some of characteristics of IO 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.

In smartphone, the storage IO is one of the key factors that governs the overall system performance [KAU]. The applications, often referred as "app", in Android based smartphones generate unique IO requests. Existing IO workload generators and trace capturing tools are designed to neither generate nor capture the IO requests 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 IO subsystem in Android based devices. AndroStep consists of workload generator, called Mobibench and workload analyzer called, Mobile Storage Analyzer (MOST).

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

2 Problem Assessment

2.1 IO characteristics of Android based Device

IO characteristics of Android based device [LW12] is different from server [HS03] or desktop [ZS99, HDV$^+$11] IO characteristics. In order to measure the performance of Android based device properly, we need a tool that can reproduce IO behaviors of Android platform. This section investigates whether existing tools are capable of capturing the Android IO characteristics and reproducing them. This section further analyzes limitation of existing tools. Through thorough analysis of existing benchmark tools, we not only differentiate our tool from existing tools but also create a basis for implementing a benchmark tool for Android-based platform.

There are many benchmark tools available for measuring the performance of file systems or storage devices; however, existing benchmark tools, such as IOzone [ioz] or Androbench [KK12], can only measure limited amount of Android resources. In this section, we briefly explain pros and cons of IOzone and Androbench and their limitation in measuring performance of Android devices. One notable work on workload analysis of Android based platform is done by Kim et al. [KLH$^+$11]. They investigate which system services are used by Android applications and try to allocate sufficient IO bandwidth to corresponding application via computing IO bandwidth usage model. They classified applications into three classes, bursty, time-sensitive, and plain; and showed a result of using their classification and IO management usage model in media application. However, their approach has two issues. First, they chose to model the IO characteristics of well-known system service instead of analyzing IO behavior of each application. Second, they neglected to model various scenarios within running an application, and also did not consider the fact that an application utilizes multiple system services.

2.2 Workload generation of Android based Device

IOzone is a widely used benchmark tool for measuring the performance of file system as well as IO subsystem [ioz]. There are a few issues in IOzone that makes it not suitable as a benchmark tool for Android IO subsystem. First is that IOzone does not include `fsync()` followed by a 4 KB random write operation. It is not only a frequently repeated IO operation in Android but also one of most important IO operations that journal of SQLite performs. Upon receiving a data, journal in SQLite forces to commit the data via `fsync()`. Two to three `fsync()` calls are made to the storage depending on the journal modes of SQLite. Note that IOzone affords an option to include `fsync()` in the benchmark; however it does not simulate journal of SQLite. The option includes time of `fsync()` operation in two cases, which is after finishing the whole buffered IO and file close operation.

Second is that IOzone cannot measure performance of SQLite3, a default DBMS in Android system. Therefore, one needs to implement an Android app to measure performance of basic operations such as DB insert, update, and delete in SQLite3. Another way to measure the performance of the basic operations is to implement a shell based test tool which exploits SQLite3 APIs and cross-compiler. To build a shell based test tool, SQLite3 library must be statically linked to the application in order to run such a test tool in an Android platform. A problem in such a method is that statically linked SQLite3 library is different from shared library of the Android platform. As a result, performance acquired using the testing tool significant differs from optimized shared SQLite3 library. Thus, it is impossible to acquire reliable performance. It has to be noted that SQLite3 uses Apache license, which forbids to ask for optimized version of the open source.

The third issue concerns testing method. IOzone must run sequential write benchmark operation before performing any other benchmarks. Mandatory write benchmark in IOzone incurs significant overhead in measuring large sized IO and in repeated runs of tests.

Androbench [KK12] is a file benchmark tool that runs in Android system. What is special about Androbench is that it includes options to measure SQLite operations. Although Androbench is a simple, and yet a powerful benchmark tool to measure performance of file and SQLite operations in mobile device, there are three limitations in measuring various IO characteristics of Android based devices. First, Androbench do not allow changing synchronization options such as O_SYNC, O_DIRECT, and mmap. Only available option is O_SYNC. Second, it cannot measure performance effect of `fsync()` calls. Finally, Androbench does not support multi-threading benchmark environment.

2.3 Workload Analysis of Android based Device

Traditional IO characterization study is defined on four dimensions : IO type (Read vs. Write), IO size (KB), spatial locality (Sequential vs. Random), and temporal locality (Hot vs. Cold). All these characteristics can be analyzed by examining the block access trace. To properly understand the IO characteristics of Android Apps, one needs to acquire the four defined IO characteristics under various different contexts: subject to file types, block types, application processes, etc. For example, we need to understand how many IO writes are for metdata and how many writes are for synchronous when the IO is for journal file writes. Existing file system analysis tools does not provide the correlation information between the IO characteristics and the different file system attributes (block type, file type, and application process).

For detailed study, one has to acquire IO access characteristics, e.g., Read vs. Write, IO size; however, as far as we are aware of, existing workload capture and analysis tools are not suitable to study the details of application behavior in Android IO. It is not suitable because the tools does not identify the file type for a given block and identify the application for a given block. To provide remedy to the issues, we implement two features in MOST.

- Identify the file type for a given block: From the logical block address captured by block access trace tool [AB07], we identify the type of file which the respective block belongs to. This process consists of two steps. First, from a logical block address, MOST identifies the inode number of a file where the logical block address belongs to. Then, from the inode number, MOST identifies the file type of the respective file.
- Identify the application for a given block: It is important not to confuse application dependent IO characteristics with file type dependent access characteristics. MOST allows to distinguish the two different IO characteristics.

3 AndroStep

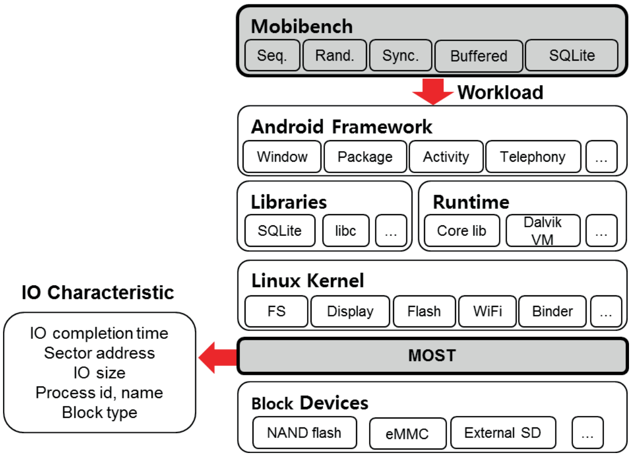


Figure 1: Structure of AndroStep (Mobibench and MOST)

Android storage performance analysis tool consists of a workload generator, Mobibench[1] [2] (Mobile Benchmark), and trace collection tool, MOST[3] (Mobile Storage Analyzer). Figure 1 illustrates structure of AndroStep, which consists of Mobibench and MOST, in the Android platform. Mobibench generates read and write IO requests which mimics the IO pattern of Android based storage device. MOST is a tool to collect IO statistics and other useful information in comprehending the IO from Android based device.

[1]Mobibench is available at google playstore.
[2]Mobibench, `https://github.com/ESOS-Lab/mobibench`
[3]Mobile Storage Analyzer, `https://github.com/ESOS-Lab/MOST`

3.1 MOBIBENCH

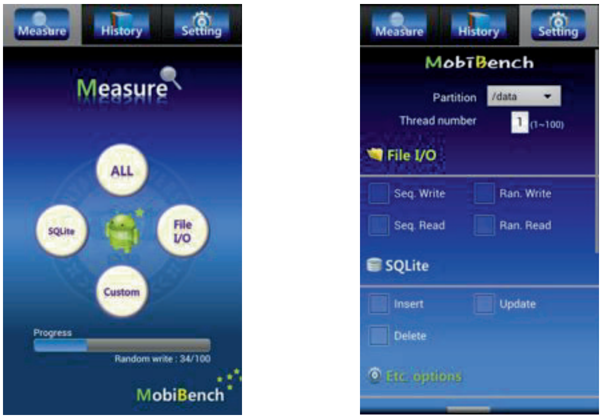
Mobibench is a benchmark tool especially designed for simulating IO characteristics of Android system; Mobibench is capable of measuring the performance of file IO and DB operations using SQLite. Mobibench is implemented in two versions, one as a shell application and the other as an Android Application (the app.) Both versions use same measurement engine written in C language. Since JAVA cannot call the app in C directly, we use JAVA Native Interface (JNI) to run the app. The shell application supports both Android device and desktop system. If the application is compiled in Android device, the application exploits SQLite shared library. Since manufacturers provide optimized SQLite library, it is possible to acquire accurate performance of SQLite operations using the app. On the other hand, on a desktop system, the application exploits SQLite static library. Since Mobibench provides unified measurement engine, it allows to test and compare the performance in diverse system.

Table 1: **Functional comparison**

	Function	Mobibench	IOzone[ioz]	Androbench[KK12]
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 IO operation	O	X	X
	separate SQLite operation	O	X	X
	file sync mode	O	O	X
	SQLite journal mode	O	X	X

Table 1 illustrates comparison of three benchmarks, Mobibench, Iozone, and Androbench, on performance measurements, feedback, and miscellaneous options. Note that Mobibench combines all of the functions that are only available in Iozone or in Androbench. Mobibench provides detailed benchmark configuration options such as choice of a partition, number of threads, and workload characteristics. Once a partition and number of threads is configured, the setting is applied to all test cases. Mobibench uses one of `/data` and `/sdcard` in internal storage, or `/extSdcard` in external memory card. Differentiating the partition is important because available file system mount options for each partition are not the same. Mobibench allows configuring the number of threads for a test in order to provide similar environment as smartphones, where multiple threads execute IO and SQLite operations simultaneously.

There are two categories of workload, File IO and Database Operations. Test cases avail-



Partition /data
Thread number 1 (1~100)

(a) measure tab. (b) setting tab.

Figure 2: Mobibench application

able in File IO category is choice of sequential and random, and read and write. Mobibench further specifies file size, IO size, and synchronous mode. The synchronous mode supports five different modes, buffered, synchronous, direct, mmap, and `write()+fsync()`. Since synchronous mode cannot be changed in file `open()` call of JAVA environment, Mobibench implements synchronous mode through C/C++ and JNI.

Database operation measures performance of insert, update, and delete in SQLite, the default DBMS in the Android system. Performance of these operations varies greatly depending on compile and PRAGMA options of SQLite. PRAGMA command is used to modify the operation of the SQLite library, which Mobibench uses to change SQLite synchronous or journal modes. According to our test, performance of SQLite varies significantly depending on the choice of synchronous and journal modes. Although modified and optimized source code of SQLite is not publically available, manufacturers provide SQLite shared library which is optimized to Android device. Mobibench uses the shared library to measure the performance of SQLite operations.

Mobibench generates three results, Throughput, CPU Utilization, and Number of Context Switches. In the case of File IO test, unit of throughput is “KB/s” for sequential operation and “IOPS” for random operation. 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.

One must have superuser permissions to flush buffer cache explicitly in shell version of Mobibench. Mobibench explicitly flushes buffer cache at initialization phase of the application via writing to `vm.drop_caches`.

Figure 2 illustrates user interface of the app version of Mobibench. There are three tabs in Mobibench. In Measure tab, there are four buttons, ALL, File IO, SQLite, and Custom.

File IO runs sequential or random read/write operations, and SQLite runs transactions of insert, update, and delete DB operation. ALL runs both File IO and SQLite operation and Custom runs only the tests user have selected. In Setting tab, there are customizable options which are also available in the shell version of Mobibench. When a measurement process is running Mobibench displays a progress bar to show status of a running test, and illustrates the result upon completion of the test. Mobibench shows three results, Throughput, CPU Utilization, and Number of Context Switches. Since Android applications cannot run in Superuser, Mobibench cannot flush buffer cache explicitly; instead it opens a file with `O_DIRECT` to remove the effect of buffer cache in the measurement.

3.2 MOST: Mobile Storage Analyzer

Mobile Storage Analyzer (MOST) 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 3 provides a schematic of MOST.

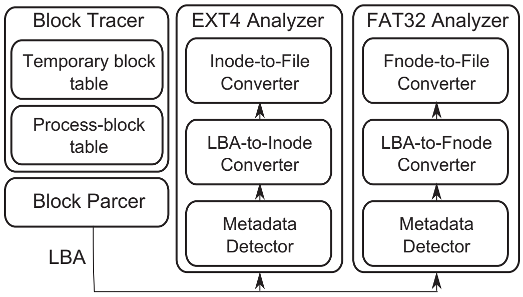


Figure 3: Mobile Storage Analyzer

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 file system 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 different layers. MOST collects the IO trace at the block device driver level to deduces the file information and then it processes information for each respective block. MOST addresses three reverse-mapping issues: LBA-to-file mapping, LBA-to-process mapping, and retrospective LBA mapping.

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` [Ts'] to reverse-map the block in the EXT4 file system, and an in-house module for the FAT32 file system.

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 all block IOs that are initiated by the *mmcqd* daemon, which is not the information we are interested in. We create a process-to-block mapping 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 the process-to-block mapping table. MOST references the process-to-block mapping 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. These temporary files are created by SQLite for managerial purposes, for example, creating a temporary database file for update and committing the changes to the storage device. It is very important to have proper understanding of how these files are utilized. Although they have short-life span, each files are *fsync()*ed to NAND storage, which greatly affects system performance. We need file information for a given LBA when a trace is recorded, not when posthumously analyzed. When MOST initiates analysis procedure for a given LBA, a temporary file where the block belonged might have been deleted and therefore cannot be found. To address this issue, MOST creates a file-to-block mapping table in the Android kernel. A file-to-block mapping 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-to-block mapping 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 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.

Table 2: **Output of Mobile Storage Analyzer**

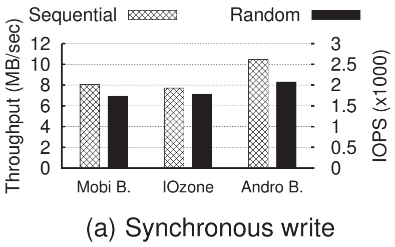
1	IO completion time
2	Flags for read and write
3	Sector address and IO size
4	Process id and process name
5	Block type: *Metadata*, *Journal*, and *Data* block
6	File name in case of the *Data* block

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

4 Experiment

We verified performance result of Mobibench against similar benchmark tools, IOzone and Androbench. We tested common features in the benchmark tool to verify accuracy of Mobibench, and show performance result of additional features of Mobibench. The results of the study presented here are based on the Galaxy S3 [GAL] (running Android 4.0.4 with Linux Kernel 3.0.15).

4.1 Common Features



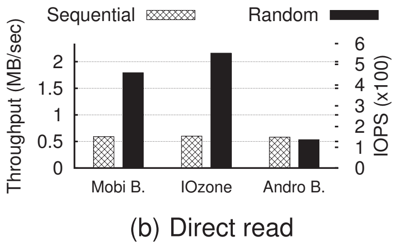


Figure 4: I/O on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

We compared common features of three benchmarks in three different experiments. First experiment compares common features present in all benchmarks. Second experiment compares features present only in Mobibench and IOzone. Third, we compare features present only in Mobibench and Androbench. We used `/data` partition in internal eMMC. The partition is formatted in EXT4 file system.

We have validated that the result of Mobibench is accurate through measuring performance of common features available in different benchmark applications. The common feature provided in all three benchmarks are sequential and random read/write operations. We tested synchronous write and direct read performance of the three benchmarks because Androbench supports synchronous write and direct read operations only. File and IO size in the experiment is set to 512 MB and 4 KB, respectively. Figure 4(b) and Figure 4(a) shows the results from each benchmark. Mobibench and IOzone shows similar throughput and IOPS on both sequential and random operations. On the other hand, all performance result of Androbench except sequential read differs from the result of the other two benchmarks. Random write performance is regarded as critical performance measure in eMMC and NAND Flash based SSD and it is important to measure them accurately. Random write performance observed in Mobibench and IOzone shows about 10% difference; however, result of Androbench shows about 80% slower IOPS than that of IOzone.

Mobibench and IOzone supports various file open options for measuring the performance of random and sequential read/write operations. We compare the two benchmarks using six different IO operations including buffered read/write, mmap read/write, synchronous write, and direct write. File and IO size is set to 512 MB and 4 KB, respectively. We test sequential and random IO on all six modes. Result of Mobibench is not far different from the result of IOzone. On Buffered read and mmap read Mobibench shows about 25% higher sequential performance. The difference is incurred by inclusion of execution time of `fsync()` at the end of read test. IOzone includes execution time of `fsync()` in total run time of buffered and mmap read/write; however, it is unnecessary to include execution

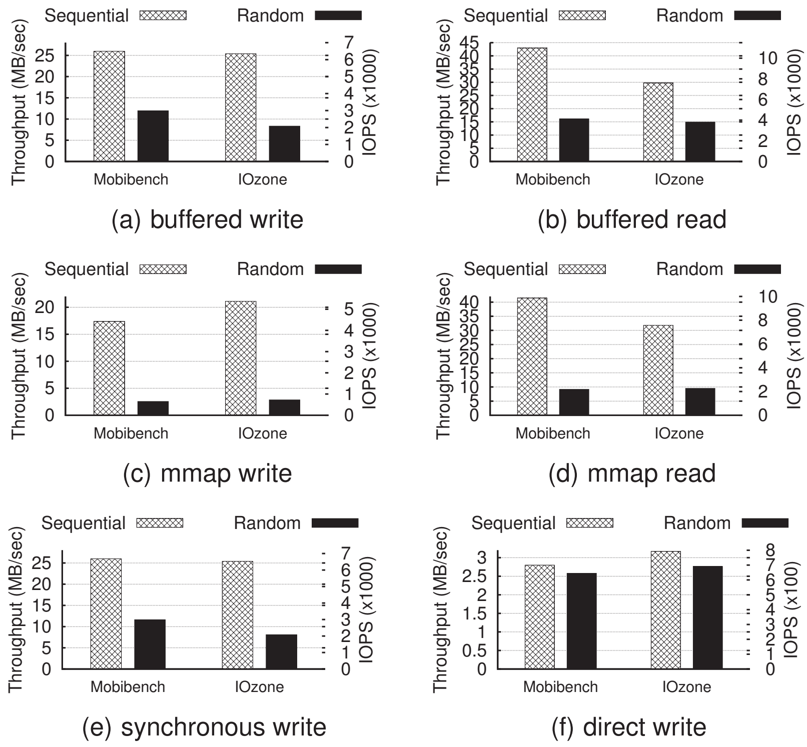


Figure 5: Mobibench vs. IOzone, IO on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

time of `fsync()` on buffered and mmap read. We believe that buffered and mmap read result of IOzone is biased because it includes execution time of `fsync()` at the of the experiment.

Next, we measure performance of SQLite operation using Mobibench with various smartphone models and android version. We use TRUNCATE journal mode, and FULL sync mode to measure the performance and the result is average of 1000 runs. Figure 6 shows the result of the benchmark. Galaxy S3 and Galaxy Note2, which has Jelly Bean installed, show the highest SQLite performance compared to other devices. The performance seems to be affected by version of Android used in devices because same device, i.e. Galaxy S3, with difference Android version shows performance difference of more than 150%. It shows that optimization applied to Android platform, especially on SQLite, seems to produce great difference in the performance. However, it cannot be said that version of Android platform alone matters to the performance because Nexus7 and Galaxy Nexus, which also uses Jelly Bean, shows very low performance. The two devices have very low hardware specifications than Galaxy S3 or Galaxy Note. Hardware performance of a device has obvious effect on the performance.

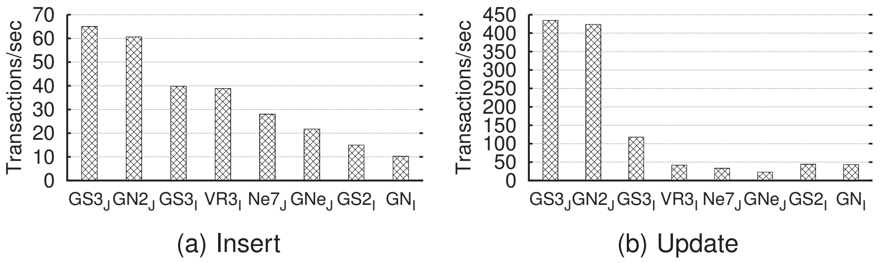


Figure 6: SQLite insert and update performance on various smartphone models. SQLite journal mode: TURNCATE. GS3: Samsung Galaxy S3, GN2: Samsung Galaxy Note2, VR3: Pentech Vega R3, Ne7: Google Nexus 7, GNe: Samsung Galaxy Nexus, GS2: Samsung Galaxy S2, GN: Samsung Galaxy Note; Subscript i and j denotes the version of Android, Ice Cream Sandwich(4.0.4) and Jelly Bean(4.1), respectively. (`/data` partition, EXT4)

4.2 New Features

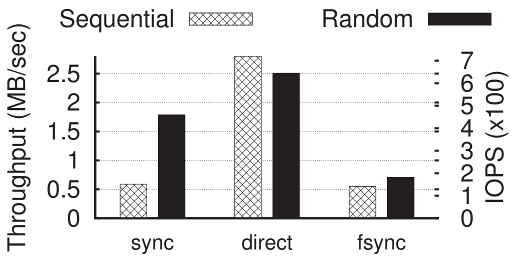


Figure 7: `write()+fsync()`, IO on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

There are some other features that are not available in IOzone or Androbench. We devote this section to explore features available only in Mobibench. We show three different experiments in File IO, SQLite, and multithreading environment. We test File IO using `write()+fsync()`, which is dominant workload in Android system. We vary journal and sync mode of SQLite and run experiments on multithreading environment. We use the same device and partition as previous section.

Figure 7 compares the result of `write()+fsync()` against synchronous and direct write using file and IO size of 512 MB and 4 KB, respectively. IO behavior of `write()+` `fsync()` and synchronous write is similar on sequential IO; however in random write workload, number of IOs caused by `write()+fsync()` increase and thus it shows about twice as slower performance than synchronous write.

Figure 8 shows performance of `write()+fsync()` while increasing number of threads to 32. As number of threads increases the performance also increases because Galaxy S3 is equipped with quad core CPU; however, when there are more than 16 threads the

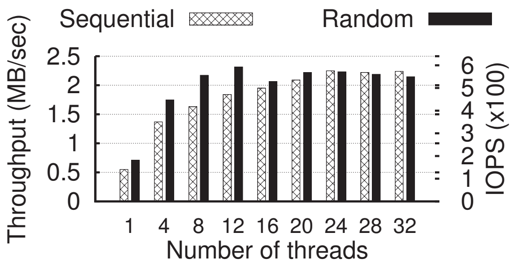


Figure 8: `write()+fsync()` with multi-thread, IO on internal eMMC. Filesize: 512 MB, IO-size: 4 KB (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

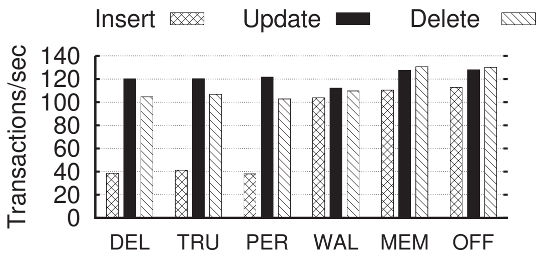


Figure 9: SQLite operation with various journal modes on internal eMMC. sync mode: FULL, SQLite version: 3.7.5 (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

performance saturates because scheduling overhead becomes the bottleneck.

Mobibench provides various options in measuring the performance of SQLite operation. Mobibench allows to change SQLite journal and sync mode. There are six Journal modes in SQLite and they are as follows: DELETE, TRUNCATE, PERSIST, WAL, MEM, and OFF. SQLite suggests that WAL Journal mode shows the best performance other than MEMORY and OFF mode [WAL]. In MEMORY mode, SQLite stores the journal information in system memory and OFF mode does not keep account of the journal. Figure 9 shows the result of average TPS of running 1000 insert, update, and delete operation. Insert in WAL mode shows about 2.5 times better performance than other Journal modes. On the other hand, update mode in WAL mode shows slightly lower performance than the other modes. Main reason behind the result is modified SQLite library in Galaxy S3. The manufacturer modified the library to use OFF mode in update operation on DELETE, TRUNCATE, and PERSISTENT mode. Using strace, we have verified that indeed all except WAL mode operates like OFF mode in update operation.

Another mode that have noteworthy effect on performance of SQLite is sync mode. There are three sync modes in SQLite which are FULL, NORMAL, and OFF mode. Number of times SQLite calls `fsync()` is determined by the mode. FULL mode calls `fsync()` on every transaction to guarantee that all the data is written to storage device. NORMAL

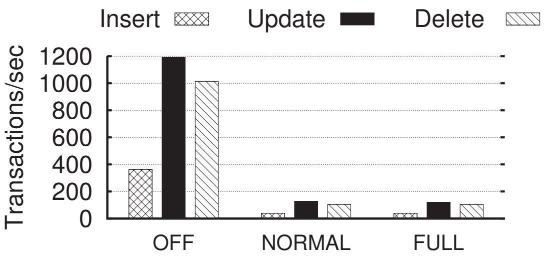


Figure 10: SQLite operation with various sync mode on internal eMMC. journal mode: DELETE (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

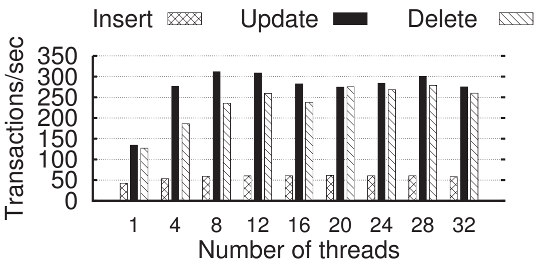


Figure 11: SQLite operation with multi-thread on internal eMMC. sync mode: FULL, journal mode: DELETE (Samsung Galaxy S3, Android 4.0.4 (ICS), `/data` partition, EXT4)

and OFF mode reduces number of `fsync()` calls to produce better performance in return of sacrificing the data integrity. Figure 10 illustrates the result of three sync modes while running insert, update, and delete SQLite operations. It shows that using OFF mode speeds up greatly than using NORMAL or FULL mode in all SQLite operations.

Mobibench supports multi-threading in measuring the performance of SQLite operations as well. Figure 11 illustrates the effect of increasing number of threads up to 32. We observe no further performance gain when the number of threads is more than 12. The result is in accordance with file IO experiment and can be interpreted as scheduling overhead.

5 Conclusion

In this work, we develop a suite of software for analyzing the IO performance of the Android based device. This suite intends to generate the IO patterns which uniquely exist in the Android based storage device and which can capture the context dependent IO characteristics in various levels of the IO stack: application, file system, and block device.

6 Acknowledgements

This work is sponsored by IT R&D program MKE/KEIT. [No.10035202, Large Scale hyper-MLC SSD Technology Development].

References

[AB07] Jens Axboe and Alan D. Brunelle. Blktrace User Guide, 2007.

[Axb07] J. Axboe. CFQ IO Scheduler. In *presentation at linux. conf. au, Jan*, 2007.

[eMM11] EMBEDDED MULTI-MEDIA CARD(e-MMC), ELECTRICAL STANDARD (4.5 Device), June 2011.

[GAL] Samsung Galaxy S3. `http://www.samsung.com/ae/microsite/galaxys3/en/index.html`.

[HDV$^+$11] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In Ted Wobber and Peter Druschel, editors, *SOSP*, pages 71–83. ACM, 2011.

[HS03] W. W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347 –372, 2003.

[ioz] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[KAU] 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*.

[KK12] Je-Min Kim and Jin-Soo Kim. AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices. In Sabo Sambath and Egui Zhu, editors, *Frontiers in Computer Education*, volume 133 of *Advances in Intelligent and Soft Computing*, pages 667–674. Springer Berlin Heidelberg, 2012.

[KLH$^+$11] H. Kim, M. Lee, W. Han, K. Lee, and I. Shin. Aciom: Application characteristics-aware disk and network I/O management on Android platform. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 49–58, 2011.

[LW12] Kisung Lee and Youjip Won. Smart Layers and Dumb Result: IO Characterization of an Android-based Smartphone. In *EMSOFT 2012: In Proc. of International Conference on Embedded Software*, Oct. 7-12 2012.

[MCB$^+$07] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proc. of the Linux Symposium, Ottawa*, 2007.

[SQL] SQLite Homepage. `http://www.sqlite.org/`.

[Ts'] Theodore Ts'o. Debugfs. `http://linux.die.net/man/8/debugfs`.

[WAL] Write-Ahead Logging. `http://www.sqlite.org/wal.html`.

[ZS99] Min Zhou and Alan Jay 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.