I/O Stack Optimization for Smartphones

Sooman Jeong[1], Kisung Lee[2,*], Seongjin Lee[1], Seoungbum Son[2,*], and Youjip Won[1]
[1] Hanyang University, Seoul, Korea
[2]Samsung Electronics, Suwon, Korea

Abstract

The Android I/O stack consists of elaborate and mature components (SQLite, the EXT4 filesystem, interrupt-driven I/O, and NAND-based storage) whose integrated behavior is not well-orchestrated, which leaves a substantial room for an improvement. We overhauled the block I/O behavior of five filesystems (EXT4, XFS, BTRFS, NILFS, and F2FS) under each of the five different journaling modes of SQLite. We found that the most significant inefficiency originates from the fact that filesystem journals the database journaling activity; we refer to this as the JOJ (Journaling of Journal) anomaly. The JOJ overhead compounds in EXT4 when the bulky EXT4 journaling activity is triggered by an `fsync()` call from SQLite. We propose (i) the elimination of unnecessary metadata journaling for the filesystem, (ii) external journaling and (iii) polling-based I/O to improve the I/O performance, primarily to improve the efficiency of filesystem journaling in the SQLite environment. We examined the performance trade-offs for each combination of the five database journaling modes, five filesystems and three optimization techniques. When we applied three optimization techniques in existing Android I/O stack, the SQLite performance (inserts/sec) improved by 130%. With the F2FS filesystem, WAL journaling mode (SQLite), and the combination of our optimization efforts, we improved the SQLite performance (inserts/sec) by 300%, from 39 ins/sec to 157 ins/sec, compared to the stock Android I/O stack.

1 Introduction

Smart devices, e.g., smartphones, tablets, and smart TVs, have become mainstream computing devices and are quickly replacing their predecessor, PCs. Smartphones and tablets have become the dominant source of DRAM consumption [17] and account for 45% of Internet web browsing [18]. They are becoming *the* personal computing devices for a variety of applications, including social network services, games, cameras, camcorders, mp3 players, and web browsers.

The application performance of a smartphone is not governed by the speed of its airlink, e.g., Wi-Fi, but rather by the storage performance, which is currently utilized in a quite inefficient manner [11]. Furthermore, one of the main sources of this inefficiency is an excessive I/O activity caused by uncoordinated interactions between EXT4 journaling and SQLite journaling [14]. Despite its significant implications for the overall smartphone performance, the I/O subsystem behavior of smartphones has not been studied nearly as thoroughly as those in enterprise servers [26, 23], web servers [4, 10], OLTP servers [15], and desktop PCs [34, 8].

In this work, we present extensive measurement results to understand Android's I/O behavior and propose techniques to optimize the individual layers so that the overall Android I/O stack behaves much more efficiently when the layers are integrated. The Android I/O stack is a collection of elaborate and mature software layers (SQLite, EXT4, the interrupt-driven I/O of the Linux kernel, and NAND-based storage), each of which has gone through several years of development and refinement. When the layers are integrated, the resulting I/O behavior is not well-orchestrated and leaves a substantial room for an improvement. We overhaul the I/O stack of the Android platform from DBMS to a storage device and propose several techniques to improve the performance. Our contributions are as follows:

• Starting from EXT4, we performed an extensive performance study of five filesystems (BTRFS, XFS, NILFS, and F2FS [1]) in one of the most recent Android-based smartphones and examined how they interact with each journaling mode of SQLite. We found that SQLite journaling interacts with the EXT4 journaling layer in an unexpected way and, the EXT4 filesystem stresses the storage device in a way that was rarely seen before.

[*] This work was done while the author was a graduate student at Hanyang University.

We found that recently introduced F2FS can be a good remedy for Journaling of Journal anomaly which current stock Android I/O stack suffers from.

- Examining five journal modes of SQLite, we found that Write-Ahead-Logging mode(WAL) yields the best performance since it generates smallest amount of the synchronous random writes amongst all SQLite journal modes.
- We propose the use of external journaling, in which the filesystem journal is maintained in a separate storage device to explicitly preserve the access locality induced by the filesystem journal file access. This approach enables the FTL layer of the NAND storage to more effectively exploit the locality of the incoming I/O stream so that it can reduce the NAND flash management overhead, e.g., garbage collection in page mapping and the log block merge operation in hybrid mapping.
- We found that SQLite triggers a significant amount of synchronous random write traffic when it commits its journal file to the filesystem, a significant fraction of which is not required. We tuned SQLite to eliminate this random write I/O by employing `fdatasync()` in place of `fsync()`.
- NAND-based storage is sufficiently fast, and state-of-the-art smartphones are equipped with a sufficient number of CPU cores. We developed a polling-based I/O system for Android storage devices and studied its effectiveness.

Combining these optimization efforts with the optimal choices for the filesystem and database journaling mode of SQLite (i.e., by F2FS, WAL journaling mode in SQLite, using external journaling, eliminating unnecessary metadata commits, and polling-based I/O), we achieved a 300% improvement in the SQLite performance (inserts/sec) compared to the stock Android platform.

The remainder of the paper is organized as follows: Section 2 presents the background. The I/O characteristics of Android is briefly described in Section 3, and the current Android I/O stack is examined in Section 4. Section 5 explores various filesystem choices for Android. Section 6 provides optimization techniques for the Android I/O stack. Section 7 presents the results of our integration of the proposed schemes. Section 8 describes other works related to this study. Our conclusions are presented in Section 9.

2 Background

2.1 Android I/O Stack

Google Android is an open-source Linux-based operating system designed for mobile devices. Figure 1 illustrates the architecture of Android. Android applications are written in Java and packaged as `.apk` (Android Application Package) files. Android provides a set of libraries used extensively by various system components and applications; some of the most widely used libraries are SQLite, libc, and the media libraries. The Linux kernel provides core services, such as memory management, process management, security, networking, and a driver model. Android uses the Dalvik virtual machine (VM) with just-in-time compilation to run `.dex` (Dalvik Executable) files, and an application runs on top of the Dalvik VM.

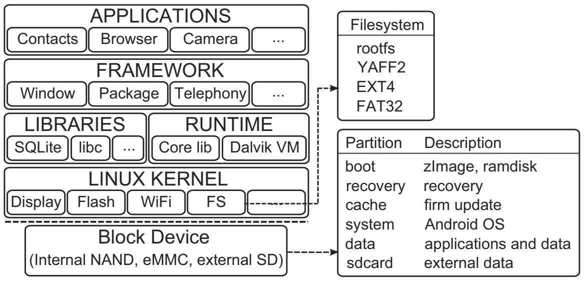


Figure 1: Android Architecture and Storage Partition

We define the Android I/O stack as a set of software and hardware layers used by applications for persistent data management. The I/O stack of the Android platform consists of the DBMS, filesystem, block device driver, and NAND flash based storage device. SQLite and EXT4 are the default DBMS and filesystem, respectively. The Android platform uses interrupt driven I/O with a CFQ I/O scheduling scheme. The eMMC and SD card are used as internal and external storage devices, respectively.

Most Android applications use SQLite to manage data in a persistent manner. SQLite interacts with the underlying filesystems through the usual system calls, such as `open()`, `unlink()`, `write()`, and `fsync()`. SQLite uses journaling for recovery. It records rollback information at `.db-journal` file. The database file and journal file are frequently synchronized with the storage device using `fsync()`.

Since the release of Android 4.0.4 (Ice Cream Sandwich), Android only uses EXT4 to manage its internal storage, eMMC.

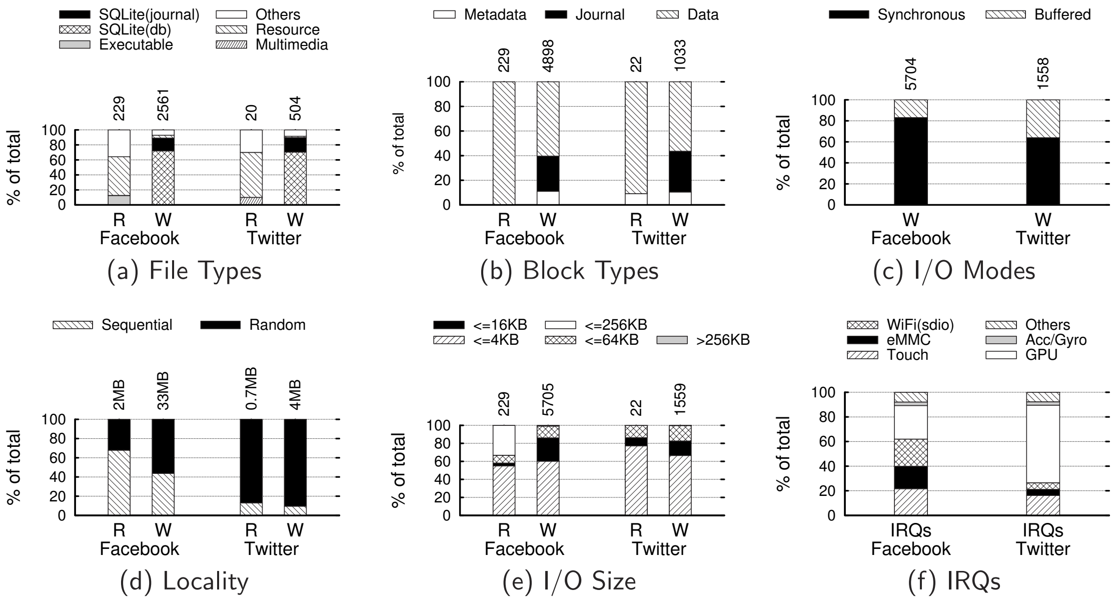
2.2 AndroStep: Android Storage Analyzer

We use Androstep [9] to collect, analyze and replay the trace in this study. AndroStep is a collection of tools developed for analyzing the storage stack behavior of Android. It consists of Mobibench[1], MOST[2], and Mobigen[3]. Mobibench (mobile benchmark) is an I/O workload generator which is specifically designed for An-

[1] `https://github.com/ESOS-Lab/Mobibench`, available at Google playstore

[2] `https://github.com/ ESOS-Lab/MOST`

[3] `https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen`



(a) File Types (b) Block Types (c) I/O Modes

(d) Locality (e) I/O Size (f) IRQs

Figure 2: I/O distribution of file types, block types, I/O modes, randomness, and I/O size. The number on the top of each bar indicates the number of respective block I/O for R (Read) and W (Write), respectively.

droid workload generation. It can generate SQLite workload (insert, update, and delete) and filesystem workload (read, write, create, unlink etc). User can configure SQLite journaling option, filesystem journaling operation, and various filesystem I/O options, e.g., direct vs. buffered I/O, synchronous I/O, and etc. The accuracy of Mobibench is validated against existing widely used benchmark IOZONE [9].

MOST (mobile storage analyzer) is a tool to collect and to analyze the block level trace. From the block trace, MOST can identify the block type (metadata, journal, and data), and the file type of the respective block such as SQLite journal/database, apk, and etc. The salient feature of MOST is that it keeps track of this information for deleted files.

In addition to Mobibench and MOST, Androstep has a tool to record and to replay the system call trace, Mobigen (Mobile Workload Generator). Mobigen is used to collect the system calls generated from the human user behavior for using a given application. By replaying the system call trace, Mobigen can reproduce the human driven I/O activities without actual human intervention.

3 I/O Characteristics of Android Applications

Prior works performed extensive study of Android I/O characterization and found that a significant fraction of the I/O's are generated by SQLite operation [11, 14]. Kim et.al. [11] found that most I/Os in Android platform are related to SQLite database operations. Lee et.al [14] performed extensive I/O characterization study and found that dominant fraction of Andriod I/O is synchronous random write caused by misaligned interaction between SQLite and EXT4 filesystem. We analyzed the I/O behavior of Facebook and Twitter apps, both of which are highly popular smartphone applications. We present the analysis results only for Facebook and Twitter apps because the results are well aligned with our prior study on fourteen popular Android apps and exhibits similar characteristics [14].

The results of the study presented here are based on the Galaxy S3 (Samsung Exynos 4412 1.4 GHz Quad-core, 2 GB RAM, 32 GB eMMC, Android 4.0.4 with Linux kernel 3.0.15)[4]. We use MOST (Mobile Storage Analyzer) [9] to collect and analyze the I/O trace. Figure 2 illustrates the results of the analysis. The numbers on the top of each bar represent the number of I/O requests. We briefly summarize our findings as follows:

- *90% of the write requests are to the SQLite database and journal.* We categorize the files into six categories: database file (`.db`), journal file (`.db-journal`), executables (`.so`, `.apk`, and `dex`), resources (`.dat` and `.xml`), and others. We found that SQLite and its journal file are responsible for approximately 90% of the write I/O requests in both Facebook and Twitter apps (Figure 2(a)).
- *Writes to the EXT4 journal block constitute 30% of all writes.* We categorized the blocks in the filesystem partition into three types: metadata, journal,

[4] We have also tested earlier smartphone models, the Nexus S (Android 2.3 "Gingerbread", 2010 Nov.) and Galaxy S (Android 2.1 "Eclair", 2010 Mar.). We only show the Galaxy S3 results to save space. I/O behaviors of earlier smartphone models are similar to that of the Galaxy S3.

and data. 10% and 30% of all writes are for the metadata and journal, respectively (Figure 2(b)).

- *Of all writes, 70% are synchronous.* Figure 2(c) shows the number of buffered and synchronous writes. 70% of all writes are synchronous I/O operations, initiated primarily by SQLite.
- *75% of all writes are random.* Figure 2(d) shows the spatial characteristics of the write operations. In general, random writes are unfavorable for NAND storage devices and are considered to be a source of performance degradation.
- *64% of the I/O operations involve data with size less than 4 KB.* Figure 2(e) shows the I/O size distribution. A dominant fraction (64%) of the I/O requests has sizes of 4 KB. This is because in SQLite on EXT4, every update to the database table and the respective journaling activity are synchronized with the storage device.
- *The interrupt requests issued by the eMMC comprise 18% of all interrupts.* Figure 2(f) shows the interrupt requests from each device driver. We found that the eMMC is responsible for 18% of the interrupt requests on average.

4 Analysis of the Android I/O Stack

In this section we examine SQLite journaling and EXT4 file system journaling. We focus on how Android storage system is affected subject to the SQLite journaling mode, especially SQLite journaling and EXT4 journaling are both active.

4.1 Journaling in SQLite

SQLite is the most popular persistent data management module on the Android platform. Even multimedia players use SQLite to store configuration options such as the speaker volume. SQLite uses journaling to provide transactional capabilities for its database operations. There are six journaling modes in SQLite: DELETE, TRUNCATE (default in Android 4.0.4), PERSIST, MEMORY, write-ahead logging (WAL), and OFF. The differences among these modes are both subtle and profound.

In DELETE, SQLite creates a journal file at the start of a transaction and deletes it upon completion of the transaction. After the journal file is created, SQLite inserts journal records and calls `fsync()` to make the journal file persistent.

In TRUNCATE mode, SQLite truncates the journal file to zero instead of unlinking it when the transaction completes. This truncation is performed to relieve the burden of updating the metadata (for example, the directory block and inode bitmap) involved in creating and deleting the database journal file.

PERSIST mode takes a more aggressive approach than TRUNCATE mode to more efficiently reduce the journaling. In PERSIST mode, SQLite writes zeros at the beginning of the database journal when the transaction completes instead of truncating the file. When inserting a new record into journal file, PERSIST mode uses the existing blocks (zero-filled block), whereas TRUNCATE mode allocates a new block. The amount of metadata committed to filesystem journal is smaller in PERSIST mode compared to TRUNCATE mode.

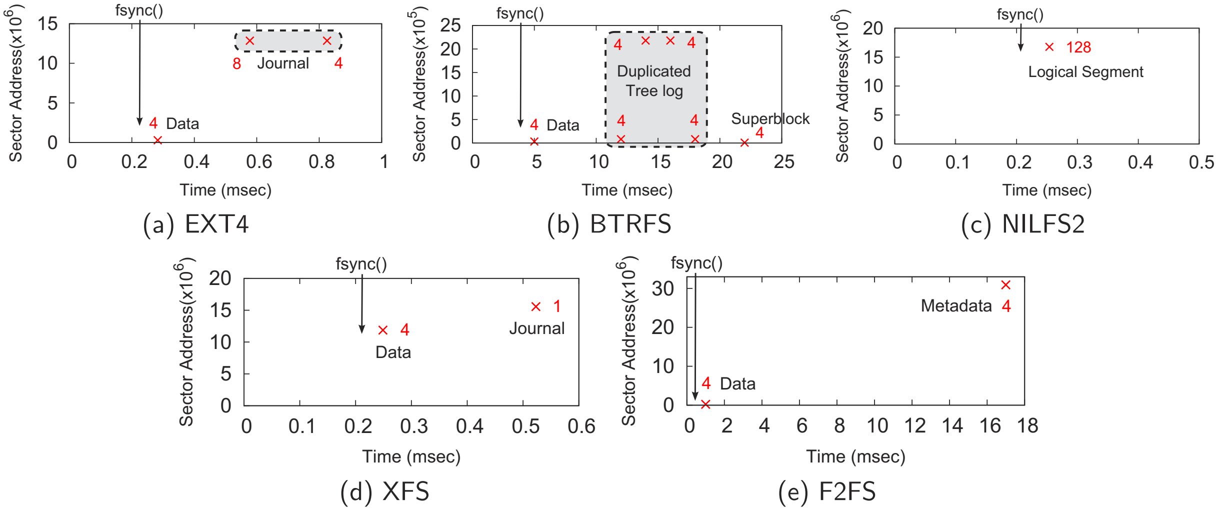
In MEMORY mode, the journal records are kept in memory. Since MEMORY mode does not rely on filesystem service to maintain journal records, MEMORY mode does not have any variants different from the filesystem based journal modes.

WAL journaling mode creates a separate WAL file (`.wal`) and logs the database updates to the log file. When the `.wal` file reaches a specified threshold size, the outstanding logs of the `.wal` file are checkpointed to the database file (`.db`). In WAL mode, I/O operations tend to be sequential; therefore, this mode is good for exploiting the nature of NAND flash storage. The OFF journaling mode does not use journaling.

4.2 EXT4 Journaling and fsync()

EXT4 has long been the default filesystem on the Android platform. For efficiency, EXT4 journaling maintains the log records for multiple system calls as a single unit called a *journal transaction* and uses this as the unit at which the log records in memory are committed to filesystem journal. Normally, the overhead of journaling is negligible in EXT4 because a journal transaction consists of a large number of log records and a journal transaction is committed to the EXT4 journal file at relatively long intervals, e.g., every 5 sec. In the Android platform, however, the journaling overhead of EXT4 becomes rather significant because of its frequent `fsync()` calls. As we show in Section 4.3, the insert operation in SQLite issues two or more `fsync()` calls within 2 msec. Each `fsync()` call triggers the commit of a journal transaction in which the journal transaction consists of very few (often one or two) log records that represent the updated in-core metadata for the respective file. Consequently, EXT4 journaling becomes very inefficient when it is triggered by `fsync()`.

Let us physically examine the effect of `fsync()` in EXT4 journaling (ordered mode). We generated a 4 KB write followed by an `fsync()` call. Figure 3(a) illustrates the results. In ordered mode, filesystem first updates the file and commits the respective file metadata to filesystem journal. As a result of `fsync()`, there occurs three write operations to the storage. The first write in the lower range of LBA is the data update. The second and the third writes are for committing updated metadata to filesystem journal; writing the journal descriptor, inserting the log record(s) and writing a journal commit mark. In Figure 3(a), the journal descriptor and log record are



(a) EXT4

(b) BTRFS

(c) NILFS2

(d) XFS

(e) F2FS

Figure 3: Block I/O pattern: 4 KB `write()` followed by `fsync()` EXT4, BTRFS, NILFS2, XFS, and F2FS. Number at each block I/O denotes I/O size in KB.

written in a single write operation. A single `write()` system call entails two additional block write operations, which are for updating the filesystem journal. The journaling overhead is 200% in this experiment.

`fsync()` not only creates additional write operations but also disintegrates the locality of the underlying workload. `fsync()` introduces more randomness to the underlying traffic because of frequent journal commits; consequently, `fsync()` significantly degrades the performance as well as lifetime of NAND-based storage.

4.3 Journaling of Journal: Interaction between SQLite and EXT4

Previous study [14] reported that the excessive I/O behavior of Android-based smartphones is due to the uncoordinated interaction between SQLite and EXT4, but the detailed mechanism has not been studied. We performed an in-depth analysis of the block-level I/O activity caused by SQLite and EXT4 (ordered mode). The application inserted one record (100 Byte) into the SQLite database table in this experiment. For comprehensiveness of the study, we examined four journaling modes of SQLite: DELETE, TRUNCATE, PERSIST, and WAL. Figure 4 shows the results. We denote the time of I/O, the respective starting LBA, and the size. Additionally, we specified the file where the I/O is designated.

In SQLite, the insert operation primarily consists of two phases: (i) it logs the insert operation at the SQLite journal, and (ii) it inserts the record to the actual database table. SQLite calls `fsync()` at the end of each phase to make the results persistent. Each `fsync()` call makes EXT4 filesystem update the file (database journal or database table) and write the updated metadata to the EXT4 journal.

Let us begin with DELETE mode (Figure 4(a)). SQLite creates the journal file (`.db-journal`), enters the journal entry for the insert operation and then calls `fsync()`. Upon `fsync()`, EXT4 writes `.db-journal` to storage and commits the updated metadata for `.db-journal` to the EXT4 journal. Then, SQLite inserts the record into the database table (`.db`) and calls `fsync()` to force the record into storage. When `fsync()` is called again, the same steps are repeated as in the first phase. Finally, SQLite calls `unlink()` to delete the `.db-journal` file. *A single insert operation results in nine I/Os to the storage device*.

The differences among three different journaling modes of SQLite, DELETE, TRUNCATE and PERSIST, lie in how SQLite treats the database journal file (`.db-journal`). These differences affect the amount of metadata committed to the EXT4 journal. When SQLite reuses the journal file (TRUNCATE mode), EXT4 is relieved from the burden of committing the metadata updates caused by the creation and deletion of SQLite journal. In PERSIST mode, SQLite not only reuses the existing journal but also reuses the data blocks of the journal file. Consequently, when SQLite operates in PERSIST mode, EXT4 is further relieved from the burden of committing the updated metadata involved in allocating a new data block to SQLite journal. Let us look at our experiment results (Figure 3). The first write operation designated to filesystem journal in each of Figure 4(a), Figure 4(b) and Figure 4(c) is for committing the updated metadata for the SQLite journal (`.db-journal`) to the EXT4 journal. The sizes of these operations are 24 KB, 16 KB, and 8 KB in DELETE, TRUNCATE, and PERSIST modes, respectively.

In PERSIST mode, however, SQLite generates addi-

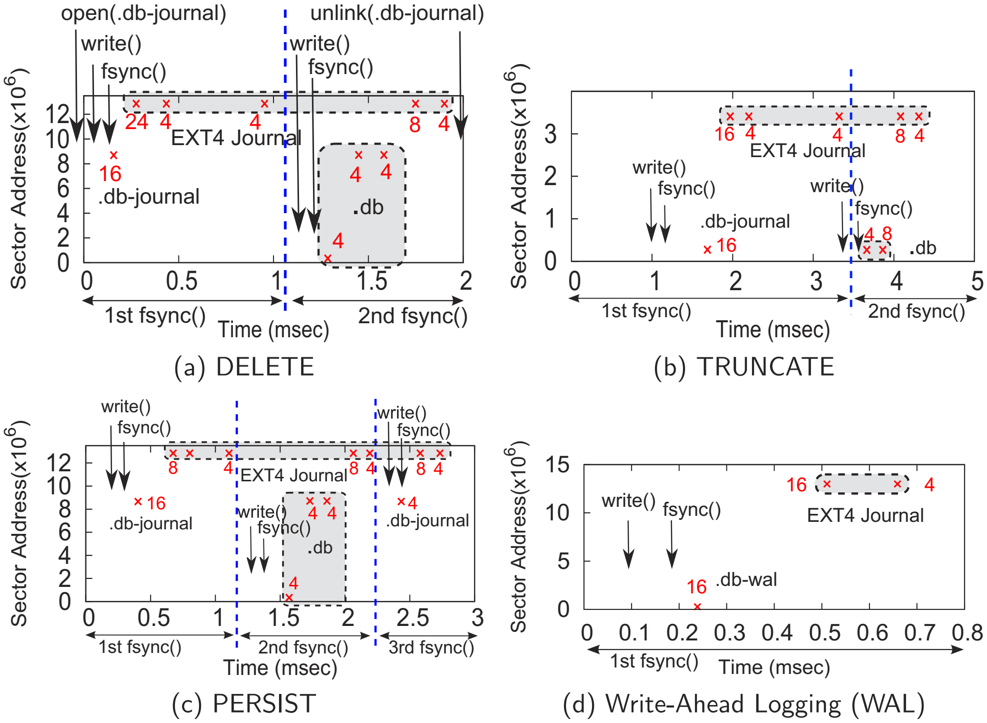


Figure 4: Block I/O accesses of the SQLite insert operation on EXT4 in Galaxy S3 (For four journal modes in SQLite 3.7.5: DELETE, TRUNCATE, PERSIST, and WAL mode). Number at each block I/O denotes I/O size in KB.

tional `fsync()` call at the end of a transaction (Figure 4(c)). This is to synchronize the zero fill operation to the SQLite journal in the storage. PERSIST mode generates the largest number of I/O (twelve I/O operations) among the four SQLite journaling modes.

In write-ahead logging (WAL), SQLite logs insert or update operations in the log file (`.db-wal`) and calls `fsync()`. Then, EXT4 filesystem updates the file (`.db-wal` file) and commits the updated metadata to the EXT4 journal. Because there is only one `fsync()` call, the overhead of filesystem journaling is the least severe and the database operation becomes the most efficient in WAL mode among five journaling modes of SQLite. Figure 4(d) shows the I/O trace in WAL mode. Because SQLite must maintain a sequence of logs in the log file, WAL mode may consume more storage space.

With an extensive analysis of the Android platform, we observed that the EXT4 filesystem *journals the database journaling activity* via `fsync()` calls from SQLite. The bulky journaling mechanism (4 KB log record) of EXT4 very frequently commits the metadata of SQLite database (`.db`) and SQLite journal (`.db-journal`). As a result, EXT4 filesystem, when used by SQLite generates excessive amount of small writes and stresses the storage in a way that has rarely been observed before. The overhead of the filesystem journaling and database journaling compounds when the operations are used together. We call this phenomenon JOJ (journaling of journal). We also found that none of the SQLite journaling modes are free from JOJ phenomenon, but WAL mode puts the least stress on the filesystem.

The ideal and classic remedy for the JOJ phenomenon is to have SQLite directly manage the storage without filesystem's assistance or to have Android apps use the filesystem primitive to maintain their data instead of using SQLite. These approaches mandate overhauling the SQLite stack or asking numerous Android application developers to use the inconvenient filesystem primitive when writing software for Android.

5 Alternative Filesystems on Android

We analyzed the behavior of four most popular filesystems to observe behavior on the Android platform: BTRFS [24], NILFS2 [13], XFS [29], and the recently introduced[5] F2FS [1]. We ported these filesystems to the Galaxy S3 (running Android 4.0.4). We examined the block-level I/O behavior and the overall performance of these filesystems.

5.1 Details of Filesystem Behavior

BTRFS [24] uses B+ tree to maintain both the data and metadata and adopts copy-on-write to update its content.

[5] Oct 5, 2012

Despite the filesystem's promising features (e.g., file and subvolume snapshots, online defragmentation, and TRIM support for SSD [28]), these two properties, copy-on-write and B+ tree, make BTRFS the worst filesystem on the Android platform. BTRFS suffers from the wandering tree problem, where an update in a tree node triggers cascading updates to the root of the tree [5]. Figure 3(b) shows the I/O behavior when `fsync()` is called after a 4 KB write. With `fsync()`, BTRFS writes four B+ tree logs and synchronizes the superblock to the storage at the end. For a 4 KB write, BTRFS generates five additional write operations when `fsync()` is called.

NILFS2 [13] is a log-structured file system. It merges a set of data writes and all updated metadata into a segment and synchronizes the segment to the storage. The size of a segment is 128 KB in NILFS2. The `fsync()` operation in NILFS2 is implemented to flush the entire logical segment. Figure 3(c) illustrates the result of a 4 KB write followed by `fsync()`. Each `fsync()` generates a 128 KB write. Despite its log-structured nature, NILFS2 does not exploit its structural advantages because of its large segment size and inefficient segment flush mechanism.

XFS [29] is a journaling filesystem that was originally designed for massive-scale enterprise storage. It is expected to handle as many files in a directory as the storage can hold, with a maximum file size of 8 EByte (8×2^{60}). XFS uses the B+ tree-based directory structure and supports sparse file for scalability. Despite its original design objective of massive-scale systems, XFS exhibits very good (the second best) performance in `write` followed by `fsync()`. Figure 3(d) shows the block access pattern of XFS. The performance advantage of XFS arises from two sources: the number of journal writes and the size of each journal write. The `fsync()` operation triggers only one journal write, which is half the number of journal writes in EXT4. Furthermore, the size of a journal write in XFS is 1 KB, whereas it is at least 4 KB in EXT4.

Flash-Friendly Filesystem (F2FS) is the youngest filesystem among the five filesystems that we studied [1]. It is a log-structured filesystem specifically designed for flash storage. F2FS categorizes incoming write requests with similar characteristics together to mitigate the overhead of garbage collection in flash-based storage. Unlike the existing log-structured filesystems that collect a sequence of writes in a single large write for filesystem updates, F2FS can also update the storage in small units, e.g., 4 KB. This feature makes F2FS behave very efficiently in the corner-case workload, such as `write()` followed by `fsync()`. Figure 3(e) illustrates the I/O trace in F2FS. It has two writes: one for data and one for inodes. The size of a write is 4 KB, whereas another log-structured filesystem, NILFS2 generates a 128 KB I/O in the same situation (Figure 3(c)).

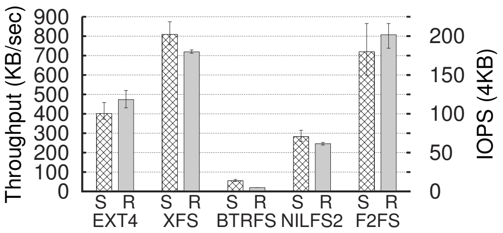


Figure 5: Sequential and random write using `fsync()` on 16GB Transcend SD card. S: Sequential (KB/sec), R: Random (IOPS), File size: 10MB, I/O size: 4 KB.

5.2 Summary: write() Followed by fsync()

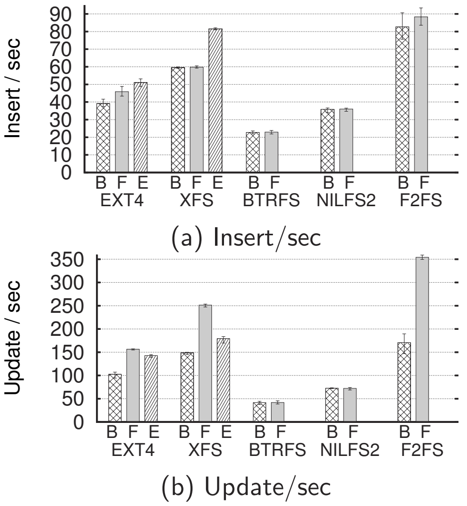
We now compare the performance of five filesystems in a typical workload in Android platform: 4 KB write followed by `fsync()`. Figure 5 shows the results. XFS and F2FS yield the best performance among the five filesystems. F2FS yields the best random write performance while the edge goes to XFS in a sequential write. The key factor governing the performance of `fsync()` is the efficiency of the filesystem journaling, which we carefully studied in Section 5.1. In XFS, the size of a log record is 1 KB, and it generates one write per one journal commit. In EXT4, the size of a log record is 4 KB, and it generates at least two writes for each journal commit operation. For random writes, XFS and F2FS surpass EXT4 by approximately 50% and 70%, respectively. BTRFS exhibits the worst performance in both sequential and random writes. We will see in Section 6 that the performance of SQLite operations in each filesystem is precisely proportional to the performance of `write()` followed by `fsync()` demonstrated in Figure 5.

6 Optimization of the I/O Stack

In this section, we introduce optimization techniques to improve inefficiency caused by JOJ phenomenon, and examine the performance effect of individual techniques.

6.1 Eliminating Unnecessary Metadata Flushes

Our first effort of the optimization is to reduce the amount of metadata committed to a filesystem journal, which is caused by `fsync()` call in SQLite. The `fsync()` operation flushes both the metadata and data to storage. We found that `fdatasync()` operation is a good alternative to `fsync()` [2] because it does not flush metadata unless it is required to allow a subsequent data retrieval to be correctly treated. In Android platform, the filesystem is mounted with `noatime` option, and SQLite states that it only cares the files size, not the other attributes. Guaranteed that the underlying OS and filesystem support `fdatasync()` correctly, the use of `fdatasync()` does not affect the filesystem integrity.



(a) Insert/sec

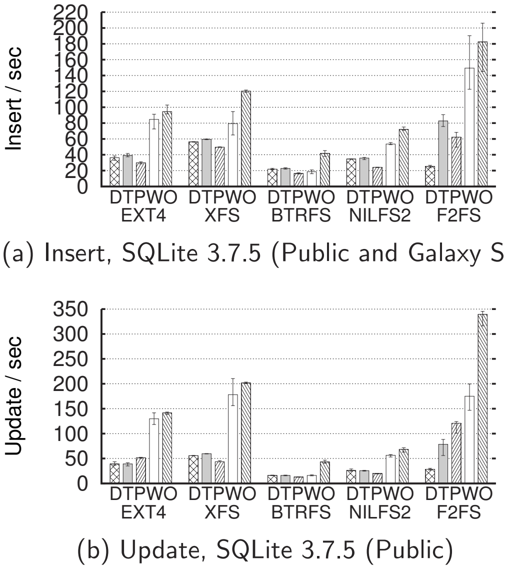
(b) Update/sec

Figure 6: SQLite Insert and update/sec for 1,000 database items on 16GB Transcend SD card. B: baseline, F: `fdatasync()`, E: External Journal.

We examined the performance of SQLite operations (insert and update) using five filesystems after replacing `fsync()` with `fdatasync()`. Figure 6 displays the results. The B, F, and E labels on the X-axis denote the baseline (`fsync()` only), `fdatasync()` enhanced version, and filesystem with external journaling, respectively. Details regarding external journaling will be presented in Section 6.3.

By using `fdatasync()`, we achieved 17% performance improvement with EXT4 for an insert operation. For an insert operation, SQLite performs the best with F2FS. SQLite exhibits a 111% faster insert rate (inserts/sec) with F2FS than with EXT4. In BTRFS and NILFS2, the advantage of using `fdatasync()` is marginal. This is because in BTRFS and NILFS2, an insert operation causes an allocation of new blocks, in which the metadata are flushed even with `fdatasync()`. Figure 6(a) illustrates the result.

The advantage of using `fdatasync()` is more significant for an update operation than for an insert operation. Figure 6(b) illustrates the result. An update is an overwrite on the existing database record from filesystem's point of view. In EXT4 and XFS, update operation does not bring any changes on the metadata such as file size, indirect blocks, free block bitmaps and etc. Therefore, using `fdatasync()` in place of `fsync()` saves significant amount of metadata flushes. In EXT4 and XFS, update/sec increases by 50% and 66% when `fsync()` is replaced with `fdatasync()`, respectively. In contrast, for copy-on-write based filesystems, e.g., BTRFS and NILFS, `fdatasync()` yields little improvement because an update operation triggers the allocation of new blocks and subsequent metadata updates, which are flushed even when using `fdatasync()`. For an update operation, F2FS yields the best performance among the five filesystems. When we used F2FS with `fdatasync()`, the SQLite performance improved by 250% compared to the baseline platform (SQLite on EXT4 with `fsync()`).



(a) Insert, SQLite 3.7.5 (Public and Galaxy S3)

(b) Update, SQLite 3.7.5 (Public)

Figure 7: SQLite performance (with `fsync()`) under varying journal modes, 1,000 database items on 16GB Transcend SD card. D: DELETE, T: TRUNCATE, P: PERSIST, W: WAL, O: OFF

6.2 Using the Optimal Journaling Mode in SQL

The I/O performance is very sensitive to the journaling mode of the underlying DBMS. We tested five journaling modes (DELETE, TRUNCATE, PERSIST, WAL, and OFF) of SQLite on each of the five filesystems (EXT4, NILFS2, XFS, BTRFS, and F2FS) and measured the performance of SQLite (insert and update). Figure 7 shows the results. The performance of an insert operation decreased by more than 50% when we used one of DELETE, TRUNCATE or PERSIST compared to when we turned off the journal. In all filesystems, WAL mode yields the best insert/sec performance among four journaling modes (Figure 7(a)).

In an update operation, WAL mode yields three times better performance compared to the other journaling modes in all filesystems (Figure 7(b)). It should be noted that different from the publicly available SQLite, Galaxy S3 version of SQLite does not create any journal file in update operation. This yields significantly better performance, but the update operation can be unrecoverable[6].

For insert and update operations, F2FS is the best-performing filesystem for most of the journaling modes.

[6] We do not include the performance result due to space limit

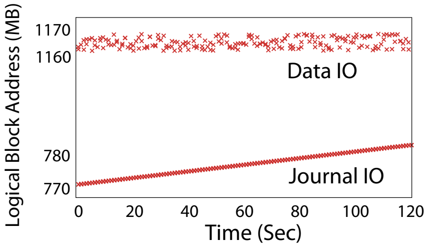


Figure 8: 4 KB random write followed by fsync()(EXT4)

Table 1: Throughput of 4 KB random write followed by fsync() on Internal eMMC with EXT4

	# of thread	Scenario	Idle		HD Record	
			base	poll	base	poll
eMMC	1	KIOPS	1002	981	667	756
		CPU (%)	7.5	10.9	26.4	30.2
	10	KIOPS	2609	2705	2136	2351
		CPU (%)	11.1	12.9	30.1	33.1

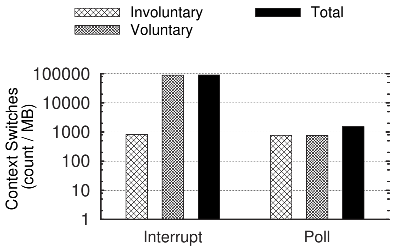


Figure 9: Number of context switches performed in interrupt-driven I/O (baseline) and polling-driven I/O

When we replace EXT4 with F2FS, SQLite performance increases at least by 67%. This is because F2FS only generates two 4KB I/Os, one for data and the other for metadata, whereas EXT4 generates 3 to 12 random I/Os depending on the journal modes. For all filesystems, WAL mode yields the best performance because all the log data created from insert and update operations are appended to .db-wal file and there occurs only one fsync() call. BTRFS exhibits the worst performance for both insert and update operations because BTRFS induces more write() calls than any other filesystems due to wandering tree behavior.

In summary, the WAL mode is the optimal journaling mode for the Android platform from performance point of view. Despite its performance benefits, Write-Ahead-Logging has some issues, space requirement and recovery time. These need to be dealt with in the separate context.

6.3 External Journaling

EXT4 and XFS have an option of storing journal blocks on a separate block device. This option is called external journaling. We now show that external journaling can be a viable option to remove randomness in the aggregate traffic and to cluster correlated writes together to the same storage region such that the underlying NAND storage can easily exploit the locality in the traffic [16, 20].

In Figure 8, we plot the I/O traces from a 4 KB random write followed by fsync() in the EXT4 filesystem. The data file is in the 1160 to 1170 MB range, whereas the EXT4 journal blocks are in the 770 to 780 MB range. We can clearly see that the aggregate traffic consists of an interleaved mixture of two different I/O streams; the locality in the data region is random, whereas that in the journal region is sequential. Separating the data and journal I/Os appears to be an obvious next step, which allows the FTL of the underlying NAND-based storage to easily identify and to exploit the locality in the incoming I/O stream. The recent eMMC interface standard [3] allows physical partitioning of the internal storage. Thus, external journaling can indeed be a practical option for future smartphone storage.

We examined the effectiveness of external journaling in EXT4 and XFS. We used a 16 GB Transcend SD card and internal eMMC for the data storage and the external journal, respectively. The results are shown in Figure 6, where E stands for external journal. External journaling yields a significant performance improvement in EXT4; the insert rate is improved by 30%, and the update rate is improved by 39%. The improvement in XFS was not as great as that in EXT4 because the journaling overhead in XFS is not as significant as in EXT4.

6.4 Polling-based I/O

Increasing number of CPU cores and decreasing I/O latency of a block device have led to a rediscovery of the value of polling-based I/O [32, 27]. State-of-the-art smartphones contain quad-core CPUs and NAND-based storage latency is an order of magnitude smaller than that of legacy hard disk drive. In this environment, interrupt-driven I/O may hinder the performance of a system due to context switches. When many small I/Os are generated from the block I/O layer, the I/O daemon for the eMMC, mmcqd, is subject to significant context switch overhead. Our results below show that this can indeed be the case and also show that polling-based I/O can provide a superior I/O performance to interrupt driven one without sacrificing the overall system performance.

We modify the I/O subsystem for the Android platform so that the mmcqd uses polling to access the storage device. There are two issues in polling-based I/O: CPU monopolization and power consumption. We perform an experiment if the polling based I/O interferes with the ongoing application, particularly CPU intensive one. We ran a HD-quality (1920x1080 at 30 fps) video recording application concurrently with our benchmark process. We found the soft real-time requirement of video recording is well preserved even when the I/O subsystem is driven by polling. We perform another set of experiment to examine the power consumption behavior of polling driven I/O subsystem. Polling-based I/O may consume more CPU cycles and may reduce the opportunity for the CPU to stay in low-power mode. According to our experiment, CPU utilization increases by 4% when we use polling based I/O. In smartphone, dominant source of energy consumption in LCD and Wi-Fi [6, 33]. We carefully argue that energy overhead of polling based I/O is marginal and therefore polling based I/O is not an infeasible option.

Figure 9 shows the number of context switches made by the `mmcqd` daemon for the baseline and poll-driven I/O. We observed that the number of voluntary context switches is reduced to 1/100 and the total number of context switches is reduced to 1/50.

We examined the I/O performance under the polling-based I/O subsystem. We ran two experiments, one for single thread and the other for ten threads, where each thread in the experiment generates 4 KB random write followed by `fsync()`. We created ten threads to examine how polling-based I/O behaves when there are frequent TLB misses. Table 1 shows the results. In the single-thread case, the performance gain shows marginal gain of 1-2% when CPU is idle; the performance gain in the polling-based I/O is 13% when the smartphone is recording HD video in the background. When there were ten threads, the performance gain is slightly smaller, but it still shows 10.1% performance gain while recording HD video. As discussed in Yang et al. [32], the performance gain will be more significant with a faster storage medium.

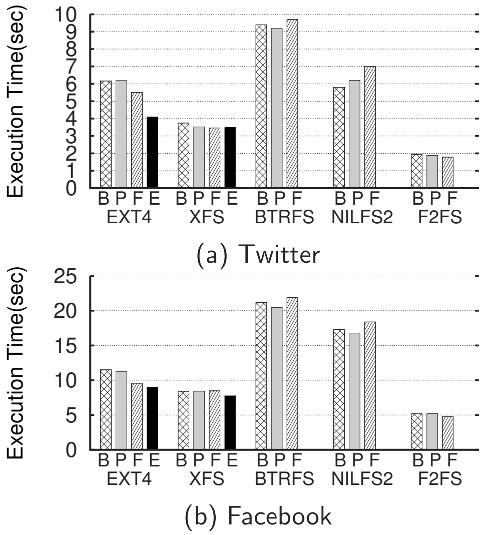
6.5 Replay of Real Workload

As the final step to verify the effectiveness of the optimization, we examined the performance of each optimization technique under a real workload. We collected the system call trace and then replayed it with `Mobigen`. We captured traces from two widely used Android applications: Twitter and Facebook.

By replaying captured I/O traces of Twitter and Facebook, we extracted the duration of I/Os processed in the two applications (Figure 10). The results of this study exhibit similar characteristics to the results that we obtained from the SQLite performance and `write()` followed by `fsync()` performance. In both Twitter and Facebook execution, F2FS performed the best.

7 Combining All the Improvements

We examined the SQLite performance on three filesystems (EXT4 as the baseline, XFS, and F2FS) when applying the aforementioned three techniques[7] in combination. SQLite journaling mode was set to TRUNCATE (default).



(a) Twitter

(b) Facebook

Figure 10: Compare execution-time of replaying script using Mobigen/Mobibench. B: Baseline, P: Polling, F: `fdatasync()`, E: External Journal

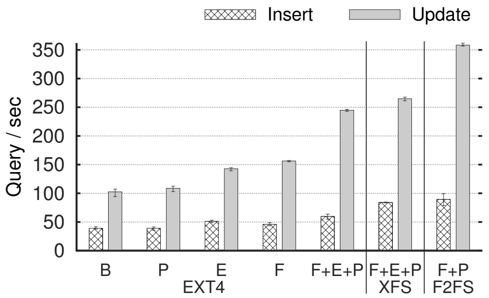


Figure 11: SQLite Performance for 1,000 database items. 16GB Transcend SD card. B: Baseline, P: Polling, F: `fdatasync()`, E: External Journal. TRUNCATE mode

Figure 11 illustrates the results. The baseline performance represents the current I/O performance: 39 inserts/sec and 102 updates/sec. Applying `fdatasync()`, external journaling, and polling-based I/O all together, SQLite on EXT4 showed 53% and 130% performance gains for insert and update operations, respectively. XFS and F2FS bring greater performance enhancements. F2FS with `fdatasync()` and polling-based I/O yields the best SQLite performance: the performance of the insert and update operations improved by 130% and 250%, respectively, compared to the baseline.

Finally, we combined all of the proposed techniques. We used WAL (write-ahead logging) SQLite journaling mode and examined the SQLite performance on three filesystems. Applying everything (`fdatasync()`, external journaling, polling-based I/O, and WAL SQLite

[7] External Journaling is not applicable to F2FS since it is a log-structured filesystem

Table 2: Performance measurements of vertical Android I/O Stack. Measurement shows performance of SQLite insert/sec and update/sec on 16GB Transcend SD card.

Optimizations	EXT4		XFS		F2FS	
	Insert/sec	Update/sec	Insert/sec	Update/sec	Insert/sec	Update/sec
Baseline	39	102	60	149	83	171
fdatasync() (F)	46	156	60	251	88	354
External Journal (E)	51	143	82	179	-	-
Polling (P)	39	109	61	153	85	226
WAL mode (W)	76	100	75	153	149	155
F + E + P	60	**245**	84	**265**	89	**358**
F + E + P + W	**92**	113	**86**	188	**157**	175

journaling mode), we achieved a 150% performance improvement (from 39 inserts/sec to 92 inserts/sec) for SQLite on EXT4. When we used F2FS instead of EXT4 in the Android I/O stack, applying everything, we achieved a spectacular 300% performance improvement for SQLite (from 39 inserts/sec to 157 inserts/sec). Table 2 summarizes the results.

8 Related Work

Storage I/O characterization has been extensively studied in various computing environments. Ruemmler et al. [26] analyzed the disk I/O in three different HP-UX systems and demonstrated that a majority of the I/O operations are writes and that the majority of writes (67-78%) are for metadata, with user-data I/O representing only 3-41% of all accesses. Roselli et al. [25] reported that file accesses follow a bimodal distribution: some files are written repeatedly without being read, whereas other files are almost exclusively for reading. Zhou et al. [34] found that the read/write ratio in the filesystem is 80%/20% and that the majority of write I/Os are random. Harter et al. [8] studied the I/O behavior of the Mac OS filesystem and demonstrated that sequential I/O on a file rarely results in sequential I/O on a block device because of the complex XML-based document format. Prabhakaran et al. [22] provided a thorough analysis of journaling filesystems, such as EXT4, ReiserFS, JFS, and NTFS, and explained the events that cause data and metadata to be written to the journal. Piernas et al. [21] suggested separating the metadata from the data and demonstrated that this separation may improve the filesystem's performance.

There are a variety of interesting studies regarding smartphones, ranging from analyzing user behavior [7] to measuring power consumption [6], security [31, 30], and storage performance [11]. Kim et al. [11] demonstrated that the conventional wisdom that storage bandwidth is higher than network bandwidth must be reconsidered for smartphones. They demonstrated that storage performance does indeed affect the performance of application and operating system because the network bandwidth has increased significantly. Kim et al. [12] proposed a new buffer cache replacement scheme that provides a better sequential access in NAND storage devices. Lee et al. [14] analyzed the I/O behavior of eleven smartphone applications and found that the journaling efforts of SQLite and EXT4 compound with each other and result in excessive random write operations. To mitigate the overhead of random writes in NAND-based storage, Min et al. [19] proposed merging multiple random writes into a single write in the log-structured filesystem. This approach does not work on the Android platform where individual random writes are synchronized to the storage.

Yang et al. [32] demonstrated that in ultra-low latency devices using next-generation non-volatile memory, polling can deliver a higher performance than the traditional interrupt-driven I/O.

9 Conclusions

Modern OSes adopt a layered architecture that guarantees the independent operation of each layer; however, neglecting the underlying mechanisms produces a considerable amount of overhead related to the storage device. The well-designed SQLite and EXT4 components have unexpected effects on NAND-based storage devices when combined together because they produce many small, random, and synchronous write I/Os due to their misaligned interaction. We thoroughly analyzed the I/O stack (DBMS, filesystem, and block device driver) of Android. We examine the block level I/O behavior of SQLite operation under its five journal modes with five different filesystems in combinatorial manner. By removing frequent updates of the metadata, dislocating the EXT4 journal to separate storage, and using polling-based I/O, we have achieved a significant performance improvement in the insert and update rates. With the F2FS filesystem, WAL journaling mode (SQLite), and the combination of our improvements, we have observed an overall performance increase of 300% in SQLite performance.

10 Acknowledgements

We would like to thank our shepherd Steve Ko, and anonymous reviewers for insightful comments and suggestions. 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] F2FS patch on LKML. https://lkml.org/lkml/2012/10/5/205.

[2] Linux programmer's manual for fdatasync. http://www.kernel.org/doc/man-pages/online/pages/man2/fsync.2.html.

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

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

[5] Bityutskiy, A. Jffs3 design issues, Nov. 2005.

[6] Carroll, A., and Heiser, G. An analysis of power consumption in a smartphone. In *Proc. of the USENIX Annual Technical Conference* (Boston, MA, US, June 2010).

[7] Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., and Estrin, D. Diversity in smartphone usage. In *Proc. of the 8th international conference on Mobile systems, applications, and services* (2010), ACM, pp. 179–194.

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

[9] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. Androstep: Android storage performance analysis tool. In *ME13: In Proc. of the First European Workshop on Mobile Engineering, Aachen, Germany* (Feb. 26 2013), vol. 215 of *Lecture Notes in Informatics*, pp. 327–340.

[10] Kant, K., and Won, Y. Server capacity planning for web traffic workload. *IEEE Trans. on Knowledge and Data Engineering 11*, 5 (Sep 1999), 731–747.

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

[12] Kim, H., Ryu, M., and Ramachandran, U. What is a good buffer cache replacement scheme for mobile flash storage? In *Proc. of the 12th ACM SIGMETRICS/PERFORMANCE, London, UK* (2012), ACM, pp. 235–246.

[13] Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., and Moriai, S. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev. 40*, 3 (July 2006), 102–107.

[14] Lee, K., and Won, Y. Smart layers and dumb result: Io characterization of an android-based smartphone. In *EMSOFT 2012: In Proc. of International Conference on Embedded Software, Tampere, Finland* (Oct. 7-12 2012).

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

[16] Lee, S., Shin, D., Kim, Y.-J., and Kim, J. Last: Locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev. 42*, 6 (2008), 36–42.

[17] Leimbach, C. Dram share in tablets growing to the detriment of pcs. DRAM Dynamics, issue 23, Sep 2012.

[18] Meeker, M. Kpcb internet trends year-end update. Kleiner Perkins Caufield & Byers, Dec 2012.

[19] Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y.-I. SFS: Random write considered harmful in solid state drives. In *Proc. of the 10th USENIX conference on File and storage technologie* (San Jose, CA, USA, Feb. 2012).

[20] Park, D., and Du, D. Hot data identification for flash-based storage systems using multiple bloom filters. In *Proc. of Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on* (may 2011), pp. 1 –11.

[21] Piernas, J., Cortes, T., and Garcia, J. M. The design of new journaling file systems: The dualfs case. *IEEE Trans. on Computers 56*, 2 (2007), 267–281.

[22] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *Proc. of the USENIX Annual Technical Conference, General Track, Anaheim, CA, USA* (2005), pp. 105–120.

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

[24] Rodeh, O., Bacik, J., and Mason, C. Btrfs: The linux b-tree filesystem. *IBM Research Report* (July 2012).

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

[26] Ruemmler, C., and Wilkes, J. UNIX Disk Access Patterns. In *Proc. of Winter USENIX* (1993), pp. 405–20.

[27] Salah, K., and Qahtan, A. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Computer Communications 32*, 1 (2009), 179–188.

[28] Shin, D. About SSD. In *Proc. of the USENIX Linux Storage and Filesystem Workshop (LSF08), San Jose, CA* (2008).

[29] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability In The Xfs File System. In *Proc. of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), USENIX Association, pp. 1–1.

[30] Vennon, T. A study of known and potential malware threats. Tech. rep., SMobile Global Threat Center, Feb 2010.

[31] Vidas, T., Votipka, D., and Christin, N. All your droid are belong to us: A survey of current android attacks. In *Proc. of the 5th USENIX conference on Offensive technologies, San Francisco, CA* (2011), USENIX Association, pp. 10–10.

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

[33] Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *Proc. of the USENIX Annual Technical Conference* (Boston, MA, US, June 2012).

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