WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly

Wongun Lee, Keonwoo Lee, and Hankeun Son, *Hanyang University;*
Wook-Hee Kim and Beomseok Nam, *Ulsan National Institute of Science and Technology;*
Youjip Won, *Hanyang University*

https://www.usenix.org/conference/atc15/technical-session/presentation/lee_wongun

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly

Wongun Lee,[*] Keonwoo Lee,[*] Hankeun Son,[*] Wook-Hee Kim,[†] Beomseok Nam[†] and Youjip Won[*]
[*]Dept. of Computer Software, Hanyang University, Seoul, Korea
[†]Ulsan National Institute of Science and Technology, Ulsan, Korea

Abstract

This work is dedicated to resolve the Journaling of Journal Anomaly in Android IO stack. We orchestrate SQLite and EXT4 filesystem so that SQLite's file-backed journaling activity can dispense with the expensive filesystem intervention, the *journaling*, without compromising the file integrity under unexpected filesystem failure. In storing the logs, we exploit the direct IO to suppress the filesystem interference. This work consists of three key ingredients: (i) *Preallocation with Explicit Journaling*, (ii) *Header Embedding*, and (iii) *Group Synchronization*. Preallocation with Explicit Journaling eliminates the filesystem journaling properly protecting the file metadata against the unexpected system crash. We redesign the SQLite B-tree structure with Header Embedding to make it direct IO compatible and block IO friendly. With Group Synch, we minimize the synchronization overhead of direct IO and make the SQLite operation NAND Flash friendly. Combining the three technical ingredients, we develop a new journal mode in SQLite, the `WALDIO`. We implement it on the commercially available smartphone. `WALDIO` mode achieves 5.1× performance (insert/sec) against `WAL` mode which is the fastest journaling mode in SQLite. It yields 2.7× performance (inserts/sec) against the `LS-MVBT`, the fastest SQLite journaling mode known to public. `WALDIO` mode achieves 7.4× performance (insert/sec) against `WAL` mode when it is relieved from the overhead of explicitly synchronizing individual log-commit operations. `WALDIO` mode reduces the IO volume to 1/6 compared against the `WAL` mode.

1 Introduction

Smart device, e.g. smartphone, smart TV, and smart pad, firmly position themselves as mainstream computing device. The mobile DRAM and mobile NAND Flash sales for smart device account for 30% [41] and 40% [8] of the world DRAM sales and NAND Flash sales, respectively. In the smartphone, the storage subsystem is arguably the main governing factor for performance [23].

Android IO stack suffers from the excessive IO behavior. Sending two character message, 'Hi', through the text messaging application yields at least 48 KByte of writes to the storage device. This anomalous amplification is due to the uncoordinated interaction between SQLite and EXT4 filesystem. The broken relationship between the EXT4 filesystem and SQLite is caused by the fact that SQLite synchronizes each change in the database file or in rollback journal file through `fsync()/fdatasync()` and that each call to `fsync()/fdatasync()` triggers the bulky EXT4 journal module to log the updated metadata. This phenomenon is called Journaling of Journal Anomaly [19].

There have been a number of efforts to mitigate the Journaling of Journal anomaly [19, 27, 35, 25, 38]. These works either modify SQLite to reduce the number of `fsync()` calls [19, 27] or modify the filesystem to mitigate the overhead of a single `fsync()` [19, 35, 25, 38]. While the overheads may vary, these works still need to journal the metadata of the SQLite journal file for each database transaction.

In this work, we dedicate our effort in resolving Journaling of Journal anomaly. We orchestrate EXT4 filesystem and SQLite so that SQLite can dispense with the expensive filesystem journaling in maintaining its journal file without compromising the file integrity under the unexpected system failure. We successfully eliminate the root cause for Journaling of Journal anomaly, the *filesystem journaling*. In our optimization effort, SQLite exploits "direct IO" in updating its journal file. Our work consists of three key technical ingredients: (i) Block Preallocation with Explicit Journaling, (ii) Header Embedding and (iii) Group Synch.

- **Preallocation with Explicit Journaling**: We preallocate the data blocks to the SQLite journal file and explicitly journal the file metadata. The subse-

quent direct IO based log-commit operation does not incur any metadata update and the filesystem journaling can be eliminated. Via explicit journaling, the SQLite journal file is protected by the underlying filesystem against the unexpected system failure.

- **Header Embedding**: We develop Header Embedding and re-design the journal file structure. With Header Embedding, the fragmented SQLite journal file structure becomes 4 KByte aligned. The Header Embedding makes the SQLite journaling operation direct IO compatible and block IO friendly.
- **Group Synchronization**: With Group Synchronization (Group Synch in short), we aggregate multiple logs and to synchronize them as a single unit. Group synch effectively reduces the overhead of synchronizing the direct IO based log-commit operation to the storage surface. It significantly improves the performance via aligning the IO with NAND Flash page size.

Combining all these techniques, we develop a new SQLite journal mode, the `WALDIO`. We implement the `WALDIO` mode in commercially available smartphone model (Samsung Galaxy S5). `WALDIO` mode with persistent direct IO exhibits 5.1× performance (insert/sec) and 2.7× performance (insert/sec) against `WAL` mode which is the fastest stock SQLite journal mode and `LS-MVBT` [27] mode which is the fastest SQLite journal mode known to public, respectively. Smartphone with non-removable battery can potentially make the direct IO a persistent operation for practical purpose, in which case `WALDIO` yields 7.4× performance (insert/sec) against `WAL` mode and 4.0× performance (insert/sec) against `LS-MVBT` mode, respectively. The improvement in `update` and `delete` follow the similar trend. `WALDIO` mode reduces the write volume of SQLite to 1/6 compared to `WAL` mode.

2 Background

2.1 SQLite

SQLite is a serverless embedded DBMS. SQLite is *the* way of maintaining the records in various smartphone platforms, e.g. Android, iOS, Tizen and etc. and is widely used as the embedded DBMS for desktop applications, e.g. Chrome web browser, Firefox, Adobe Acrobat reader, Skype [40].

SQLite adopts B-tree for its database. The size of the B-tree node is power of two ranging from 512 Byte to 64 KByte. Default node size is 1024 Byte. Fig. 1 illustrates the leaf node structure. The B-tree node consists of the page header, the index array and the cell array. The page header resides at the beginning of the node. Next to the page header, there exists an index array. Each index points the variable size record. The record, which is called *cell* in SQLite, is allocated from the end of the node. The index array and the records grow in the opposite direction. When a cell is deleted, the space occupied by the deleted record is marked as dead. The page header maintains the number of the deleted cells. The deleted cells are weaved together as a linked list. SQLite allocates a new node when there is no more free space in the page or the deleted area.

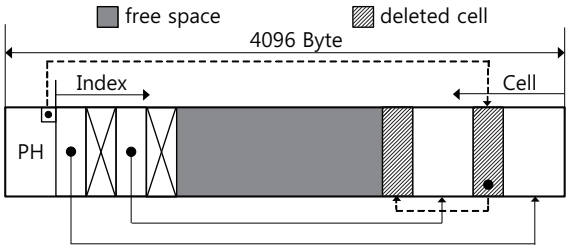


Figure 1: SQLite B-tree node structure, node size = 4 KByte, PH: Page header

Different from the large scale DBMS [39, 29, 10], SQLite does not have its own storage management module. SQLite heavily relies on the underlying filesystem to persistently manage its information and to protect it against unexpected system failure. SQLite uses *file* to maintain the log for crash recovery. For transactional guarantee, SQLite explicitly synchronizes, i.e. `fdatasync()`, the log file and the database file after committing the log or after updating the database, respectively.

SQLite provides six journal modes: `DELETE`, `TRUNCATE`, `PERSIST`, `WAL`, `MEMORY` and `OFF`. As the name suggests, `MEMORY` mode and `OFF` mode maintain the journal information in memory and does not maintain the journal information, respectively. The remaining four journal modes can be categorized into two: rollback journal and rollforward journal. `DELETE`, `TRUNCATE` and `PERSIST` modes are for rollback and `WAL` mode is for rollforward journaling, respectively.

In the rollback journal mode, the SQLite operation, e.g. `INSERT`, `DELETE` and `UPDATE`, consists of three phases: (i) logging, (ii) database update and (iii) log reset. The three SQLite journal modes in rollback journaling share the first and the second phase. In logging phase, SQLite updates the journal header and logs the old database pages (undo log) in the journal file. In database update phase, the updated database pages are written to the database file. The objective of the log-reset phase is to mark that a given transaction has successfully completed. In the third phase (log reset), there is

minor difference among the three SQLite journal modes. In DELETE mode, SQLite deletes the journal file. In TRUNCATE mode, SQLite truncates the journal file to 0. In PERSIST mode, SQLite puts special mark at the beginning of the journal file to denote that the transaction has completed. While the difference is subtle, it bears the profound implication on the filesystem journaling overhead. In DELETE mode, SQLite always needs to create the new journal file in the logging phase. Creating a file accompanies the large amount of metadata updates; the directory block, inode table, block bitmap and etc. All these metadata need to be journaled by the filesystem when fsync()/fdatasync() is called.

TRUNCATE mode retains the inode and deallocates the file blocks. Since TRUNCATE mode does not create the journal file, it yields smaller amount of metadata update compared to DELETE mode. As a result, when fsync()/fdatasync() is called in the logging phase, it yields smaller amount of EXT4 journal IO compared to DELETE mode.

PERSIST mode recycles not only the inode but also the file blocks. In PERSIST mode, the "logging" updates only the time related fields in the file metadata, e.g. mtime. Compared against TRUNCATE mode, the PERSIST mode further reduces the amount of metadata to be fsync()'ed in the logging phase. Via replacing the fsync() with fdatasync(), PERSIST mode achieves more reduction on the amount of metadata journaled in the "logging" phase [19].

In WAL mode, SQLite appends the header and a set of updated database pages to the log file (redo log). We call this file as WAL file for convenience's sake. When the database table is closed or the number of committed database pages in WAL file reaches the predefined maximum, the committed database pages in WAL file are checkpointed to the database file. SQLite provides two options to synchronize the committed logs: Full Sync and Normal Sync. In Full Sync, SQLite calls fsync()/fdatasync() after each log-commit to persistently store the logs. In Normal Sync, SQLite calls fsync()/fdatasync() after each checkpoint. In Normal Sync option, the committed logs reside in the buffer cache till they are either checkpointed by SQLite or flushed by OS. The logs in the buffer cache are subject to loss in case of unexpected system failure, e.g. power failure, or operating system crash [6] and the durability of a transaction can be compromised. The default option is Full Sync.

2.2 EXT4 Journaling

EXT4 filesystem provides three journal modes; Journal, Ordered and Writeback. The Ordered mode is the most widely used one. In Ordered mode, the filesystem logs only the updated metadata. When logging the metadata, the filesystem flushes all the data blocks related to the updated metadata and then it logs the updated metadata. EXT4 journaling module is bulky. An EXT4 journal transaction consists of a 4 KByte journal header, a set of 4 KByte journal records each of which corresponds to the updated filesystem block and a 4 KByte journal commit block. EXT4 journaling module is activated either on regular basis, e.g. in 5 sec interval, or via an explicit call to fsync() or fdatasync().

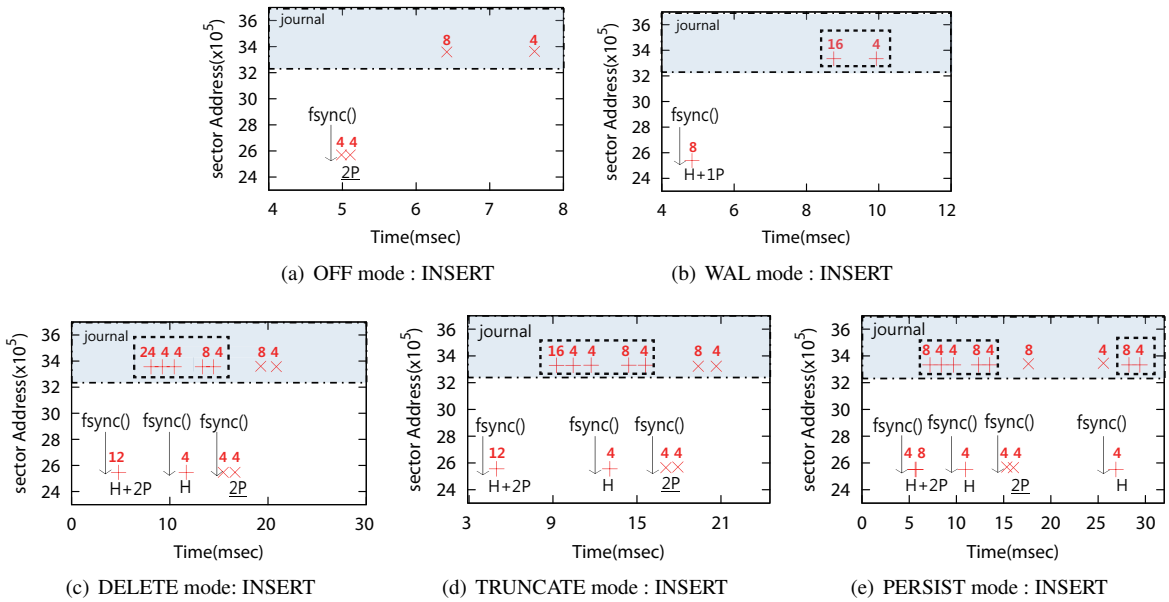
EXT4 journaling module functions efficiently when it is triggered in sufficiently large interval, e.g. in every 5 sec. With the large interval, the journal descriptor and the journal commit block pair carries sufficiently large amount of journal records in a single journal transaction. The overhead of journal descriptor and journal commit block is insignificant. SQLite drives the EXT4 filesystem in a way which, we carefully believe, has not been foreseen before and brings unacceptable inefficiency in Android IO stack. SQLite calls fdatasync() very frequently, typically after very few number of 4 KByte writes [19]. In fdatasync(), appending a 4 KByte block to a file accompanies at least 12 KByte of EXT4 journal writes.

3 Analysis of Journaling of Journal Anomaly

We overhaul the interaction between the SQLite and EXT4. SQLite inserts one 100 Byte record into an empty database table and we examine the block level IO behavior of the underlying filesystem. We use open-source benchmark, Mobibench [31] and MOST [18] to generate the workload and to analyze the IO trace, respectively. We examine the IO behavior under five SQLite journal modes: OFF, WAL, DELETE, TRUNCATE, and PERSIST.

Fig. 2 illustrates the block access patterns of SQL INSERT operations under five SQLite journaling modes. We mark SQLite journal related IO's and SQLite database related IO's with '+' and 'x', respectively. Each '+' and 'x' mark is annotated with the respective IO size in KByte unit. In the X-Y plane, the EXT4 journal region is marked with the light-grey background. The '+' marked IO's in the light-grey region correspond to the EXT4 journal writes for the SQLite journal file; *Journaling of Journal* overhead. We annotate each write in EXT4 data region with its type; the writes to journal file can be for journal header (H) or for journal record (P), respectively.

In OFF mode, SQLite synchronizes only the database file and does not accompany any SQLite journaling related IO (Fig. 2(a)). EXT4 filesystem writes two data blocks for database file and journals the respective metadata. The total 3 blocks are written in EXT4 journal re-



(a) OFF mode : INSERT (b) WAL mode : INSERT

(c) DELETE mode: INSERT (d) TRUNCATE mode : INSERT (e) PERSIST mode : INSERT

Figure 2: SQLite Block IO pattern (H: Journal Header or WAL Frame Header, P: DB page)

gion: one page of journal descriptor, one page for updated metadata and one page of journal commit mark.

In `WAL` mode (Fig. 2(b)), SQLite writes the redo log to WAL file and synchronizes the WAL file via `fdatasync()`. As a result of calling `fdatasync()`, EXT4 journals the updated metadata of the journal file, separately synchronizing the journal descriptor and journal commit mark. Since the committed database pages are checkpointed to the database file in batched manner, we do not observe any IO on the database file in Fig. 2(b).

Fig. 2(c), Fig. 2(d), and Fig. 2(e) illustrate block access patterns for rollback journal modes; `DELETE`, `TRUNCATE` and `PERSIST`, respectively. The rollback journal modes synchronize both the rollback journal file and the database file after they are updated. Compared to `WAL` mode, the filesystem journaling overhead doubles. In all these rollback journal modes (Fig. 2(c), Fig. 2(d), and Fig. 2(e)), the first and the second `fsync()` are for synchronizing the rollback journal file (phase 1: logging). The third `fsync()` is for synchronizing the updated database file (phase 2: update the database). `PERSIST` mode carries an additional `fsync()` to persistently store the reset mark in the log file (phase 3: log-reset).

The Journaling of Journal overhead for `DELETE`, `TRUNCATE` and `PERSIST` mode corresponds to 44 KByte, 36 KByte and 40 KByte, respectively. These differences are due to the way in which the SQLite journal mode resets the log file. The `WAL` mode yields the smallest JOJ overhead, 20 KByte.

Table 1 summarizes the traffic volume for five SQLite journal modes. In all SQLite journal modes, the filesystem intervention is overly excessive; the filesystem journaling activity accounts for more than 50% of the IO. While `WAL` mode yields the smallest amount of total IO, it is still subject to extreme IO inefficiency. In WAL, the filesystem journaling accounts for 70% of the entire IO traffic (20 KByte out of 28 KByte). `WAL` mode yields the smallest IO overhead and in the mean time, bears the largest room for improvement when the filesystem journaling overhead is eliminated.

	IO type (Write, KB)			
Mode	Data	Journal	JOJ	Total
`OFF`	8	12	0	20
`WAL`	8	20	20	28
`DELETE`	24	56	44	80
`TRUNCATE`	24	48	36	72
`PERSIST`	28	52	40	80

Table 1: IO Volume in inserting 100 Byte (DATA: EXT4 Data region, Journal: EXT4 Journal region, JOJ: EXT4 journal writes for SQLite journal file, and Total)

4 Direct IO and SQLite

4.1 Direct IO

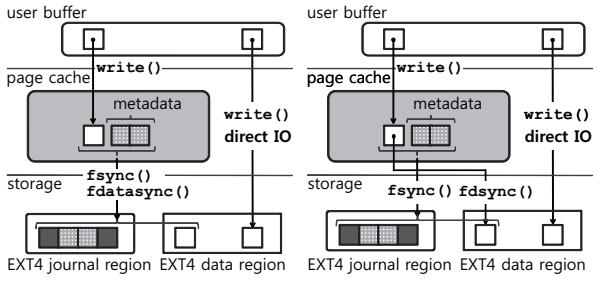
Direct IO is a filesystem feature which allows the user to read and to write the data directly from and to the storage device. In direct IO, the data block is immediately written to the storage device bypassing the page cache. Direct IO based write, DIO write for short, returns when the data blocks reach the writeback cache of the storage device.

DBMS [13, 3] and Virtual Machine Monitor [28, 33] use direct IO to manage the storage device with minimum file system intervention.

SQLite maintains the logs in a journal file. It uses buffered write in storing the log to the journal file and explicitly synchronizes the journal file via `fsync()/fdatasync()` for durability guarantee. Flushing the logs in the buffer cache can accompany the expensive filesystem journaling. If SQLite uses direct IO to write the log to the journal file, it can save the filesystem from expensive filesystem journaling activity.

4.2 Writing a Block to the Storage

We examine the three ways to write a block to the storage device: (i) `write()` followed by `fsync()`, (ii) `write()` followed by `fdatasync()` and (iii) DIO `write()`. These approaches differ in a way in which the filesystem handles the updated metadata. In `fsync()`, EXT4 filesystem journals the updated metadata for the respective file. In `fdatasync()`, EXT4 filesystem journals the updated metadata only when the file block is allocated (or deallocated). DIO write by itself does not entail any filesystem journaling. We can categorize the write operations into two types: *allocating write* and *non-allocating write*. Allocating write is a write system call which requires an allocation of a new filesystem block. Allocating write updates the various metadata, e.g. the block bitmap, inode table, intermediate node block and etc. Non-allocating write does not entail the allocation of a file block. It updates only access time related fields and possibly the `initialized` flag in the metadata.



(a) Allocating Write (b) Non-Allocating Write

Figure 3: Writing a block with `fsync()`, `fdatasync()` and direct IO

Fig. 3(a) and Fig. 3(b) schematically illustrate the IO paths of three different ways of writing 4 KByte to the storage device, for allocating and non-allocating write, respectively. For both allocating and non-allocating write, `fsync()` journals the updated metadata. In allocating write, `fdatasync()` exhibits the identical behavior as `fsync()`. In non-allocating write, `fdatasync()` does not journal any metadata. For both allocating and non-allocating write, direct IO does not accompany the filesystem journaling. In direct IO, the updated metadata, if there is any, can be subject to loss.

SQLite provides two options to synchronize the database (or journal) file: via `fsync()` and via `fdatasync()`. Android platform legitimately uses `fdatasync()` in SQLite to reduce the filesystem journaling overhead. Overhauling the IO behavior, we find an important caveat to resolve the Journaling of Journal anomaly, the *EXT4 journaling overhead*. In non-allocating write, "DIO write" yields the same behavior, though not precisely identical, with the "buffered write followed by `fdatasync()`" from the filesystem journaling's point of view; in delivering the data blocks to the storage, they both are free from the filesystem journaling.

5 Eliminating Filesystem Journaling in Android IO

We propose to use direct IO based write operation for committing the logs to the SQLite journal file so that the logs are directly written to the storage and the activity of committing the logs does not accompany any updates in the page cache entries; neither the data block nor the metadata. With this approach, the synchronization activity of SQLite, e.g. `fdatasync()`, does not trigger any filesystem journaling related IO. Our scheme consists of three key technical ingredients: (i) Preallocation with Explicit Journaling, (ii) Header Embedding, and (iii) Group Synch. Combining all these, we develop a new SQLite journal mode, `WALDIO`.

5.1 Preallocation with Explicit Journaling

The prime concern is to eliminate the interference of the EXT4 journaling in the log-commit operation and at the same time to protect the metadata of the journal file. We develop *Preallocation with Explicit Journaling*, where (i) we preallocate a certain amount of *initialized* blocks for a WAL file and (ii) journal the metadata for the created WAL file via explicitly calling `fdatasync()`. In this approach, we do rely on filesystem journaling to protect the metadata of the SQLite journal file, but suppress the every log-commit operation to accompany filesystem journaling. Fig. 4 schematically illustrates the detailed process; (i) WAL file is preallocated with the initialized blocks (labeled as 1), (ii) the metadata of the WAL file is synchronized to disk via `fdatasync()` (labeled as 2), and (iii) the logs are committed to WAL file via direct IO (labeled as 3, 4 and 5).

EXT4 filesystem maintains an `initialized` flag for each data block. A file block is said to be initialized when this flag is set. Any attempts to read the uninitialized

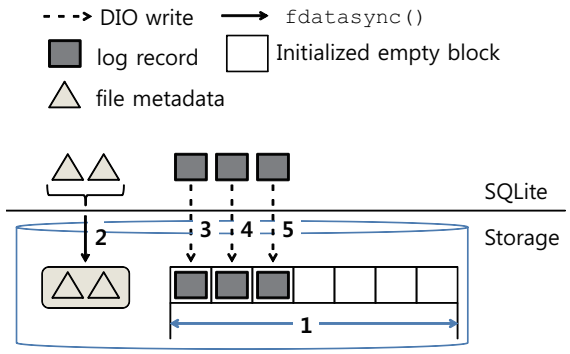


Figure 4: Preallocation with Explicit Journaling, 1: Preallocate the initialized blocks, 2: Journal the updated metadata, `fdatasync()`, 3, 4, and 5: Commit the logs with DIO

block are returned all 0's. The primary reason for this mechanism is to avoid exposing the stale data.

Being pre-allocated with the file blocks, the direct IO based log-commit operation becomes non-allocating write, where both the page cache entries and the metadata of the WAL file remain intact. The log-commit operation in `WALDIO` mode leaves no room for filesystem journaling module to interfere with. `WALDIO` mode saves the SQLite from the expensive filesystem journaling. When the WAL file is created or extended, `WALDIO` calls `fdatasync()` to synchronize the file metadata. With Explicit journaling, the WAL file becomes robust against the system failure.

In Preallocation, a special care needs to be taken to initialize the allocated blocks. Otherwise, the logs written in `WALDIO` mode may not be readable after unexpected system failure. Let us explain why. The `fallocate()` system call of EXT4 returns the uninitialized blocks. They are initialized when they are written for the first time. When a block is written with direct IO, the filesystem sets the respective `initialized` flag if it has not been initialized yet. However, since DIO write does not accompany the filesystem journaling, the updated flag is subject to loss under the unexpected system failure. While the dirty page cache entries and the updated metadata are synchronized to the storage in every few seconds, e.g. 5 sec, the contents in the writeback cache of the storage device are written to the storage surface in much shorter interval, e.g. in typically a few msec. Under the unexpected system failure, therefore, the logs written with direct IO may become unreadable even when they actually exist in the storage due to the unavailability of the `initialized` flag.

We propose three approaches to initialize the allocated blocks and subsequently to guard the stale contents in the allocated blocks against the exposure. The first and the easiest approach is to zero-fill the allocated blocks prior to use. In the second and the third approaches, we exploit the `discard` (or `trim`) command in the eMMC storage [1] to guard the stale content against the exposure. The `discard` command takes the list of the logical block addresses as an input and asks the eMMC storage to remove the mapping table entries for the respective logical blocks. In the second approach, we mount the filesystem with `discard` option and modify `fallocate()` to allocate the blocks with `initialized` flag set. When a filesystem uses `discard` mount option, it issues a `discard` command when the file blocks are deallocated. To force the `fallocate()` to return the blocks with `initialized` flags set, we port the existing `NO_HIDE_STALE` patch [34] to Linux source for Samsung Galaxy S5. In the third approach, we modify the `fallocate()` to allocate the blocks with `initialized` flag set and to discard the allocated blocks. We embed the `discard` command to the `NO_HIDE_STALE` patch [34] developed for the second approach and we develop a new flag `NO_HIDE_STALE_DISCARD` for `fallocate()`. The main difference between the second and the third approach is the time when the blocks are unmapped. In the second and third approaches, the file blocks are unmapped when they are deallocated and when they are allocated, respectively. In the second approach, the filesystem issues `discard` command for all deallocated blocks. Meanwhile, in the third approach, the filesystem discards only the file blocks allocated to WAL file. The third approach yields the smaller overhead than the second one.

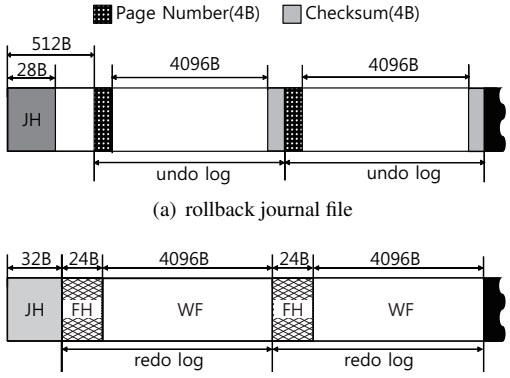
Each of these three approaches has pros and cons. The zero-fill operation accompanies IO overhead. Using `discard` mount option may slow down the filesystem [37]. Many recent smartphone devices including our test platform Galaxy S5 mount the filesystem with `discard` option. We implement all these schemes in our test platform. Via implementing all three schemes, one can choose the right scheme to initialize the allocated blocks subject to the available features of the underlying filesystem and storage device.

There exists an important implementation specific issue which deserves further attention. The `discard` command is designed to make the garbage collection more efficient [21]. It is not designed to hide the stale content. The eMMC standard [1] does not define what needs to be read when the discarded blocks are accessed. Some eMMC products, e.g. the one used by Samsung Galaxy S5 (Part No. MBC4GC), return all 0's when the discarded block is accessed. To use the `discard` command to hide the stale content, one needs to assure that the given eMMC product does not leak the stale content, i.e. is guaranteed to return all 0's or all 1's when the discarded block is accessed. Otherwise, one needs to take the resort to use `trim` command even though it is subject to larger overhead. The `trim` command is defined

to return all 0's or all 1's when the trimmed block is accessed [1]. On the same token, one also needs to assure that the given eMMC product does not ignore the `discard` command from the host in any circumstances, e.g. when the eMMC device is busy for performing background garbage collection.

5.2 Header Embedding

The direct IO operation fails when the IO size is not sector aligned. In SQLite, neither the redo log nor the undo log structures are sector aligned. Full fledged DBMS to align its database and cache organization for efficient block device interaction [10, 29, 39]. We reorganize the structure of SQLite WAL file and the database page to integrate direct IO into SQLite. Figures in Fig. 5 illustrate the redo and undo log structures of SQLite, respectively. B-tree node size is 4 KByte. The undo log of SQLite consists of 4 Byte prefix (page number), the 4 KByte journal record, and 4 Byte checksum (Fig. 5(a)). Each of these components is separately written with `write()`. In WAL file, a redo log consists of 24 Byte frame header and 4 KByte WAL frame (Fig. 5(b)). They are written separately as well. This fragmented data structure of SQLite bars the use of direct IO in managing its journal file.



(a) rollback journal file

(b) WAL file

Figure 5: Journal file structure of SQLite, node size: 4 KByte

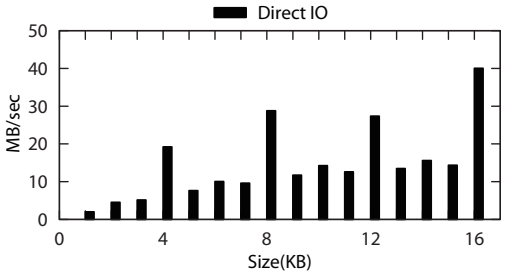


Figure 6: Effect of IO size: Sequential Write, Direct IO (GalaxyS5 eMMC Part No.: MBG4GC)

We examine the DIO write performance under varying IO size from 512 Byte to 16 KByte (Fig. 6). The sequential write performance of Samsung Galaxy S5 reaches over 70 MByte/sec. With 512 Byte IO size, the sequential write is subject to extreme inefficiency yielding mere 2 MByte/sec. This is because the IO size is not aligned with the block size. We observe larger degree of performance improvement when the IO size is aligned with the filesystem block size, 4 KByte, or NAND Flash page size, 8 KByte, respectively. This trend persists beyond 16 KByte IO size. This simple experiment provides an important direction for our optimization effort; Align the IO with the filesystem block size and with the NAND Flash page size.

We develop *Header Embedding* to align the SQLite IO with the filesystem block size. Instead of maintaining the header outside the log record, we embed the WAL header and the frame header into the header page and the WAL frame, respectively. In WALDIO, we set the B-tree node size to 4 KByte. WAL header is placed at the free space between database header and schema table in the root node of database B-tree. We harbor the 24 Byte frame header at the end of WAL-frame. Fig. 7 illustrates the log structure with header embedding. Embedding the frame header into the B-tree node, the available space in the B-tree node decreases. We physically examine the free spaces in B-tree nodes of SQLite. With few exceptions, there exists sufficient room to harbor 24 Byte field. Reserving 24 Byte for Header in the node, therefore, will not increase the number of nodes in the B-tree.

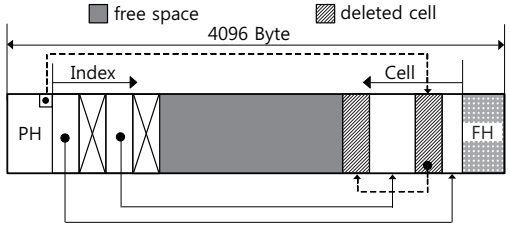


Figure 7: SQLite database page structure with Header Embedding (PH: Page Header, FH: Frame Header)

We examine the efficiency of the different aligning schemes for redo log structure. In page padding, SQLite pads the WAL header and WAL frame header to make them 4 KByte aligned. We examine four schemes: WAL (the original one), WAL with page padding, WALDIO with page padding, and WALDIO with header embedding. We perform 1,000 `INSERT` operations and examine the total IO volume. Fig. 8 illustrates the result. In `WAL` mode, total 29 MByte is written. Among 29.0 MByte, file data and EXT4 journal writes account for 12.3 MByte and 16.7 MByte, respectively. The size of WAL log record is 4120 Byte (24 Byte header and 4096 Byte frame). In committing 1,000 log entries of 4120

Byte each, total 12.3 MByte is written to filesystem data region. The fragmented log structure puts unnecessary stress on the storage device. Via properly aligning the log

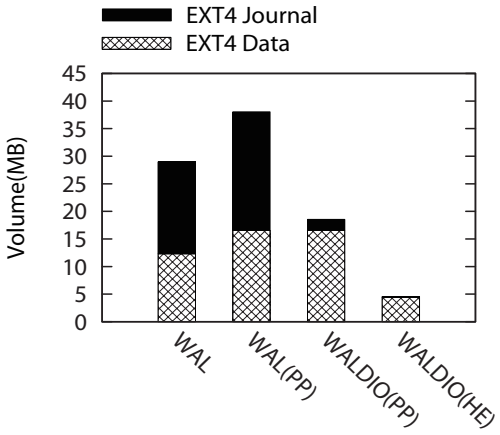


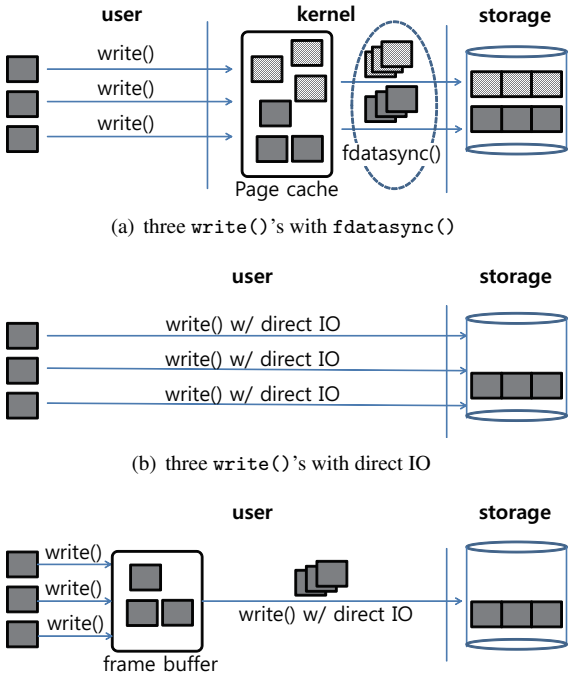
Figure 8: Total IO volume: 1000 INSERT in SQLite (PP: page pading, HE: header embedding)

structure, we reduce the IO volume to EXT4 Data region to 1/3 from 12.3 MByte to 4.4 MByte. With simple modification on the log structure accompanied by direct IO, the total IO volume decreases to 1/6 from 29.0 MByte to 4.6 MByte. The performance result will be dealt with in section 6.3.

5.3 Group Synchronization

Each layer in the IO stack, e.g. DBMS, filesystem, and block device layer, aggregates the IO's on its own way to remove the IO bottleneck [9, 11, 15, 5]. While `WALDIO` mode is successful in eliminating the filesystem journaling overhead, the individual log-commit operations are separately issued to the storage device. This nature of direct IO bars the underlying Operating System from aggregating and coalescing the IO's. If the logs are immediately synchronized to the storage surface after they are written with direct IO, the storage behavior is subject to further inefficiency since the storage device loses the opportunity to exploit its writeback cache. In this situation, the IO size plays a rather critical role in the storage performance. When the IO size is not properly aligned with the NAND Flash page size, it may cause read-modify-write problem [4, 24], proper handling of which requires complicated firmware technique such as subpage mapping [22].

We develop *Group Synchronization* (Group Synch) to mitigate the synchronization overhead of the direct IO based log-commit operation. In Group Synch, we employ *frame buffer* and *grouping interval*. All log records which have been written during a grouping interval are maintained at the frame buffer. When the grouping interval expires or when the frame buffer is full, SQLite flushes the frame buffer with DIO write. The Group Synch shares much of its idea with the prior arts; Group Commit from DBMS [9, 11] and Anticipatory disk scheduling from Operating System [17].



(a) three `write()`'s with `fdatasync()`

(b) three `write()`'s with direct IO

(c) three `write()`'s in Group Synch

Figure 9: Writing three blocks to storage: `fdatasync()`, direct IO vs. Group Synch

Figures in Fig. 9 schematically illustrates the IO behavior in writing three blocks to the storage. In Fig. 9(a), each of three blocks is written with separate `write()`'s (buffered IO) and then `fdatasync()` is called to synchronize them. We observe two sets of writes: dark gray ones and the light gray ones. The dark gray blocks correspond to data blocks. They are flushed to the disk with a single IO request. The set of light gray blocks correspond to EXT4 journal writes. In Fig. 9(b), each of three blocks is separately written via direct IO. Each of the IO requests is synchronously delivered to disk, yielding significant overhead. Fig. 9(c) illustrates the IO behavior in the Group Synch. The IO requests are first accumulated at the frame buffer and then flushed to the disk as a single DIO write. With Group Synch, the three blocks are written with single IO without accompanying the filesystem journaling. The benefit of Group Synch is twofold: reduce the the overhead for synchronizing the DIO write's and align the IO with NAND Flash page size.

Group synch provides weaker transactional guarantee than the other SQLite journaling modes since larger number of logs may get lost under system failure. However, we carefully conjecture that the difference may be

less than significant if the frame buffer size is properly set. In our limited empirical study, we find that a single SQLite transactions updates a number of database tables and indexes. For example, in contact manager application of Android, inserting an address book entry yields more than 8 logs to to the WAL file. We currently set the frame buffer size to four pages (16 KByte).

5.4 Durability

WALDIO should use Full Sync option to make the result of log-commit operation durable. When the WALDIO mode is used with Full Sync option, the log commit operation consists of two phases: (i) writing a log to the storage via DIO write and (ii) flush the writeback cache of the storage device via calling fdatasync() (Fig. 10). For better performance, we exploit Reliable Write command of eMMC standard [1] and implement *persistent direct IO*, PDIO. With PDIO write, WALDIO mode writes the logs directly to the storage surface bypassing the writeback cache of the Flash storage (Fig. 10). The blocks written with *persistent direct IO* are guaranteed to survive the power crash. With PDIO write, we can dispense with Full Sync option since each log-commit becomes immediately durable.

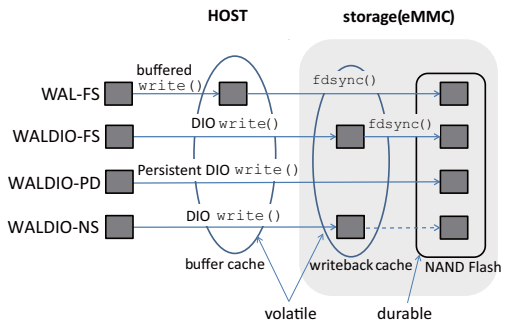


Figure 10: Making the log-commit durable: WAL with Full Sync, WALDIO with Full Sync, WALDIO with PDIO, WALDIO with Normal Sync

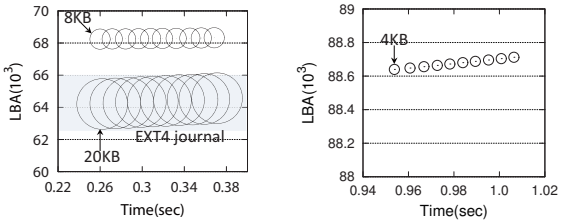
Non-removable battery in the smartphone can potentially make the DIO write a persistent one. Different from the logs in the buffer cache (Fig. 10), the logs in the writeback cache of the storage can survive the warm failure, e.g. Operating System crash [6] or kernel-panic [14]. It is very unlikely that the software bugs power off the device unexpectedly; Lue et.al. [35] reported that only 0.05% of AOSP software defect reports are related to the unexpected power failure. Also, as long as the power supply leaves some slack for eMMC to flush its writeback cache (typically a few msec), the content in the writeback cache will eventually be written to the storage surface. Given the rarity of the occasion, some application developers may prefer trading the perfect durability guarantee with the *almost* perfect durability guarantee with performance boost. For the device with non-removable battery, we carefully argue that WALDIO makes the Normal Sync option as one of the feasible choices for transactional guarantee for practical purpose.

6 Experiment

We examine the performance of WALDIO mode. We compare the behavior of the six SQLite journal modes: DELETE, TRUNCATE, PERSIST, WAL, LS-MVBT [27] and WALDIO. We implement these techniques in the recent smartphone model (Galaxy S5, Samsung, Android 4.4.2 (KitKat), Qualcomm MSM8974 Quadcore 2.5 GHz, 2 GByte DRAM, 32 GByte eMMC with 8 Kbyte page). We examine the performance of SQLite operations; INSERT, UPDATE and DELETE. We use Mobibench [31] and MOST [18] to generate the workload and to analyze the trace, respectively. We use NO_HIDE_STALE_DISCARD flag in Preallocation.

6.1 IO Access Pattern

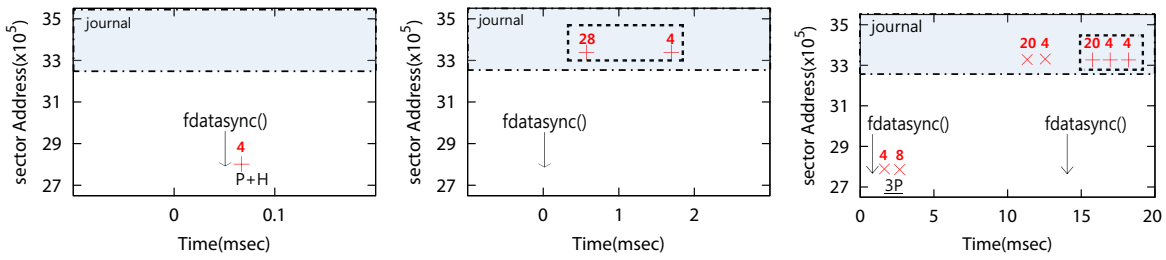
We first examine the IO access pattern of the newly proposed journal mode, WALDIO. With WALDIO mode, we insert a single 100 Byte record. Fig. 11 illustrates the result. In WALDIO, INSERT operation generates single page write (Fig. 11(a)). Be reminded that a single INSERT of 100 Byte record yields 80 KByte page writes and 28 KByte page writes in DELETE mode and WAL mode, respectively (Table 1). For illustrative purpose, we also show the IO accesses when the journal file is created (Fig. 11(b)) and when the journal file is extended (Fig. 11(c)), respectively.



(a) WAL mode (b) WALDIO mode

Figure 12: IO trace for ten INSERTs: WAL mode vs. WALDIO mode (Full Sync option)

To visualize the improvement on IO volume, we examine the IO trace for 10 INSERT operations in WAL mode (Fig. 12(a)) and in WALDIO mode (Fig. 12(b)), respectively. In this figure, the center and the radius of each circle denote the start address and the size of an IO, respectively. The circle radius is linearly proportional to the actual IO size. In WAL mode, each log-commit writes



(a) INSERT (b) Preallocation with Explicit Journaling (c) Checkpoint

Figure 11: IO accesses in WALDIO mode (P+H: Header embedding WAL Frame, 3P: three DB Pages)

8 KByte to the SQLite journal file and 20 KByte to EXT4 journal region. Due to the fragmented log structure, SQLite writes two 4 KByte blocks in committing the 4120 Byte log (24 Byte header and 4096 Byte page). In WALDIO, each log-commit yields 4 KByte IO since the frame header is embedded within the B-tree node. It does not entail any EXT4 journal IO since the log is committed with direct IO. An INSERT operation writes 28 KByte and 4 KByte to the storage in WAL and WALDIO, respectively. WALDIO successfully eliminates the filesystem journaling overhead and brings significant reduction on the total IO volume written to the storage.

6.2 Performance of Header Embedding

We examine the performance of four different page aligning schemes in WALDIO: sector padding, 4 KByte page padding, 8 KByte page padding and Header Embedding. We include the SQLite performance in WAL mode as the baseline. Fig. 13 illustrates the result. When the WAL file is sector padded (sector aligned), employing direct IO barely brings any performance gain against the WAL mode. When IO size is not aligned with the filesystem block size, the overhead of synchronously writing each data block offsets the benefit of eliminating the filesystem journaling overhead. When the WAL file structure is aligned with block size (4 KByte), the performance increases by 60% against the WAL. When the WAL file structure is aligned with NAND Flash page size (8 KByte), the SQLite performance increases by 100% against WAL. Via embedding the frame header information into the WAL frame, the SQLite yields 2.1× performance against WAL mode from 587 insert/sec to 1239 insert/sec.

6.3 WALDIO, the performance

We discuss the performance impact of WALDIO journal mode against existing SQLite journal modes: DELETE, TRUNCATE, PERSIST, WAL and LS-MVBT [27]. In WALDIO, we examine the performance under three synchronization options: (i) Full Sync, WALDIO-FS, (ii) Persistent Direct IO, WALDIO-PD and (iii) Normal Sync, WALDIO-NS. We examine the WALDIO performance with and without Group Synch. In Group Synch, the frame buffer size is set to 16 KByte with 2 msec grouping interval.

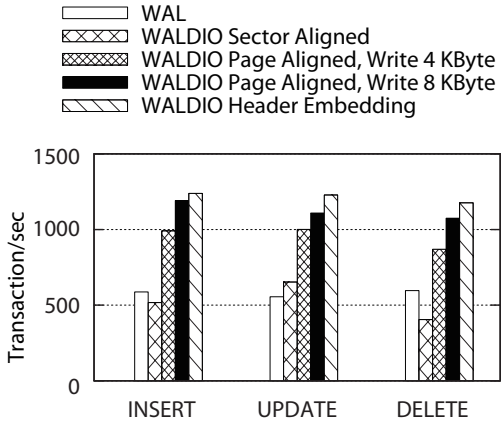


Figure 13: Page Aligning Schemes: Sector aligning, 4 KByte aligning, 8 KByte Aligning vs. Header Embedding (Full Sync)

We perform each of INSERT, UPDATE and DELETE operations 1,000 times and measure the performance. We put everything together in Fig. 14. The results in Fig. 14 are categorized into five groups: stock SQLite journal modes, LS-MVBT, WALDIO with Full Sync option, WALDIO with persistent direct IO and WALDIO with Normal Sync option. In stock SQLite journal modes, WAL mode yields the best performance (587 insert/sec). With LS-MVBT, SQLite yields 1083 insert/sec performance. With LS-MVBT, the SQLite performance increases by 80% from the WAL mode. In all these journals modes, SQLite issues fdatasync() after every log-commit.

In Full Sync option, WALDIO achieves 1219 insert/sec in the absence of Group Sync. With Group Synch with 16 KByte frame buffer, WALDIO performance leaps to 2729 insert/sec. It corresponds to 4.6× performance

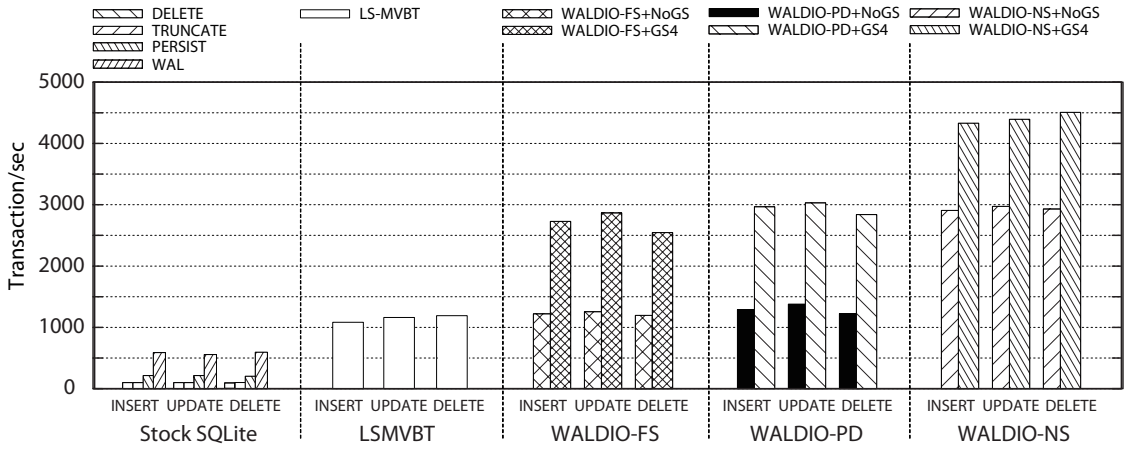


Figure 14: Performance Summary, NoGS: without Group Synch, GS4: Group Sync size = 4 pages

against stock WAL mode and 2.5× performance against LS-MVBT, respectively. Group Sync improves the performance by 120% from 1219 insert/sec to 2729 insert/sec. Group Synch is successful in eliminating the overhead of guaranteeing the durability.

With persistent DIO, WALDIO performance increases by 10% against WALDIO with Full Sync option. Bypassing the writeback cache at the storage device brings significant improvement. The performance under persistent DIO corresponds to 5.1 × performance against stock WAL mode and 2.8× performance against LS-MVBT, respectively.

In Normal Sync, the individual DIO based log-commit operations are relieved from the burden of calling the expensive fdatasync(). With Group Synch with 16 KByte frame buffer, WALDIO achieves 4332 insert/sec. The performance increases by more than 35% against the case where individual log-commits are persistently written to the storage surface; from 2967 insert/sec (WALDIO-PD) to 4332 insert/sec (WALDIO-NS). The WALDIO exhibits dramatic 7.4× and 4.0× performance compared against the WAL and the LS-MVBT, respectively. DELETE and UPDATE operations exhibit the similar performance gain with the INSERT operation. Table 2 illustrates the performance numbers for individual modes.

6.4 IO Volume

We examine the total IO volume for performing 1,000 SQLite operations, insert, update and delete. Fig. 15 illustrates the result. We limit our discussion to insert operation due to the space limit. In rollback journal modes (DELETE, TRUNCATE and PERSIST), as much as total 90 MByte is written to disk and 60% of which are for EXT4 journal writes. Via using WAL mode, the total page writes decreases to 29 MByte. LS-MVBT further de-

Journal Mode	INS	UPD	DEL
DELETE	98	97	96
TRUNCATE	99	98	97
PERSIST	213	212	203
WAL	587	556	596
LS-MVBT	1083	1161	1191
WALDIO-FS + NoGS	1219	1254	1197
WALDIO-FS + GS	**2729**	**2867**	**2546**
WALDIO-PD + NoGS	1290	1380	1224
WALDIO-PD + GS	**2967**	**3030**	**2839**
WALDIO-NS + NoGS	2907	2973	2930
WALDIO-NS + GS	**4332**	**4395**	**4507**

Table 2: SQLite performance (WALDIO-FS: WALDIO with Full Sync, WALDIO-PD: WALDIO with Persistent Direct IO, WALDIO-NS: WALDIO with Normal Sync, NoGS: Without Group Synch, GS: Group Synch with 4 pages)

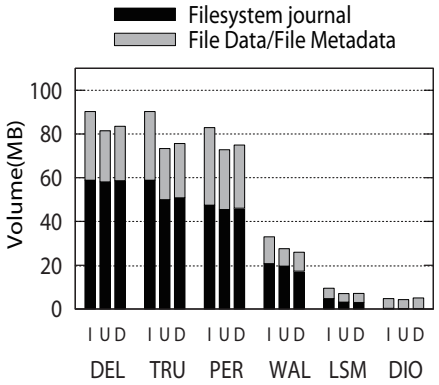


Figure 15: IO volume for 1000 INSERTs, DEL: DELETE, TRU: TRUNCATE, PER: PERSIST, WAL: WAL, LSM: LS-MVBT, DIO: WALDIO

creases the write volume to 7 MByte. In WALDIO, SQLite generates 4.6 MByte in executing an INSERT operation 1000 times. Compared to WAL mode, the total volume decreases to 1/6 from 29 MByte to 4.6 MByte.

Limited erase/write cycle of the NAND Flash storage is one of the main governing factors for the lifespan and the performance of the smartphone. The SQLite is responsible for dominant fraction of entire IO volume written to the storage. Reducing the IO volume to 1/6, the WALDIO technique can potentially allow the smartphone vendors to adopt the NAND Flash device with smaller Erase/Write cycle, e.g. TLC NAND Flash or NAND device with finer process technology, as the storage for their smartphone.

7 Related Work

While not everybody entirely agrees [35], the performance of the smartphone is governed by the performance of the storage device, not by the performance of the airlinks [23]. In Android, it is reported that more than 70% page writes generated by the smartphone application are for filesystem journal and dominant fraction of which are generated by SQLite DBMS [32]. The excessive filesystem journaling activity is due to the fact that the SQLite maintains a separate rollback journal file and synchronizes the every update in the rollback journal file via fsync() [19]. Tizen [16] also suffers from JOJ anomaly [26].

Jeong et. al. applied various IO optimization techniques, e.g. WAL mode, F2FS [30], external journaling and polling based IO and achieved 300% performance improvement against stock Android IO stack with DELETE journal mode [19]. Shen et. al. modified the EXT4 journal module and achieved 7% performance improvement against WAL mode [38]. Kim et. al. proposed to use LS-MVBT (Multiversion B-tree with Lazy Split) instead of B-tree in SQLite database [27]. LS-MVBT weaves the crash recovery information into the database file so that SQLite does not have to maintain separate file for crash recovery. LS-MVBT brings 80% performance gain against WAL mode in SQLite.

There are a number of benchmark programs for Android IO performance [12, 18, 2]. Kim et. al. [25] proposed to maintain the EXT4 journal region at NVRAM and to exploit its byte-granularity accessibility. Lue et. al. proposed to maintain the SQLite rollback journal file at DRAM [35] in the smartphone. Chidambaram et. al. proposed OPTFS to reduce the fsync() overhead involved in EXT4 journaling [7]. Kang et. al. proposed a transactional API for block device so that filesystem operations are free from the journaling overhead [20]. Piernas et. al. proposed to maintain the data and the metadata on the different blocks and maintains only single copy of metadata [36].

8 Conclusion

In this work, we successfully resolve the Journaling of Journal Anomaly in Android IO stack. We remove the root cause for excessive IO behavior in Android IO stack: the filesystem journaling. We develop a novel SQLite journal mode, WALDIO. In WALDIO mode, SQLite uses direct IO for log-commit operation so that it does not entail the expensive filesystem journaling. We develop *Preallocation with Explicit Journaling*, *Header Embedding* and *Group Synch* to enable the SQLite to exploit the direct IO semantics without compromising the filesystem integrity optimizing its performance for NAND Flash storage. The proposed features are implemented on the commercially available smartphone. WALDIO achieves as much as 7.4× increase against WAL mode and as much as 4.0× increase against LS-MVBT, respectively. With WALDIO mode, SQLite generates only 1/6 of the IO volume generated by SQLite in WAL mode.

Despite the dramatic improvement, WALDIO mode does not cost any major changes on the existing interface definitions of SQLite or of the filesystem, nor the introduction of the new ones. It is achieved by the minimal set of right modifications.

The contribution of this work should be viewed not only from the performance perspective but also from the NAND Flash endurance point of view. We carefully believe that via decreasing the IO volume generated by SQLite to 1/6, WALDIO can make the TLC NAND Flash not an infeasible choice for storage device in Android platform. Adoption of TLC NAND Flash in Android device can significantly reduce the cost of the smartphone and can make it available to wider community in the world.

9 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and feedback. Special thanks go to our shepherd Theodore Ts'o whose constructive comment and advice have made our work further mature and rigorous. We also would like to thank Yongseok Jo at EFTECH, Dongjun Shin, Seunghwan Hyun, Dongil Park and Heegyu Kim at Samsung Electronics for their advice on revising this paper. Finally, we like to thank our colleague Seongjin Lee for his help in preparing the manuscript. This work is sponsored by IT R&D program from MKE/KEIT (No. 10041608, Embedded system Software for New-memory based Smart Device) and by ICT R&D program of MSIP/IITP (No.1I2221-14-1005).

References

[1] emmc electrical standard 5.0. http://www.jedec.org/sites/default/files/docs/JESD84-B50.pdf/.

[2] http://www.antutu.com.

[3] Mysql homepage. http://www.mysql.com/.

[4] BOUGANIM, L., JONSSON, B., AND BONNET, P. uFLIP: Understanding Flash IO Patterns. In *Proc. of CIDR 2009* (Asilomar, CA, USA, Jan 2009).

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[6] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The rio file cache: Surviving operating system crashes. In *Proc. of ASPLOS 1966* (Cambridge, MA, USA, Sep 1996).

[7] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proc. of ACM SOSP 2013* (Farminton, PA, USA, 2013).

[8] CHINAFLASHMARKET.COM. 2013 NAND Flash Market annual report.

[9] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD 1984* (Boston, MA, USA, 1984).

[10] EFFELSBERG, W., AND HAERDER, T. Principles of Database buffer management. *ACM Trans. on Database Systems 9*, 4 (Dec. 1984), 560–595.

[11] GAWLICK, D., AND KINKADE, D. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull. 8*, 2 (1985), 3–10.

[12] GINGRICH, A. The Great Android Police Storage Benchmark: 11 Modern Devices Compared In 13 Tests. http://www.androidpolice.com/tags/rl-benchmark/.

[13] GRANCHER, E. Oracle and storage IOs, explanations and experience at CERN. In *Proc. of JPCS CHEP 2009* (Prague, Czech, Mar 2010).

[14] GU, W., KALBARCZYK, Z., IYER, R. K., AND YANG, Z. Characterization of linux kernel behavior under errors. In *Proc. of DSN 2003* (San Francisco, CA, USA, Jun 2003).

[15] HAGMANN, R. Reimplementing the Cedar File System Using Logging and Group Commit. *SIGOPS Oper. Syst. Rev. 21*, 5 (Nov. 1987), 155–162.

[16] HTTP://WWW.TIZEN.ORG.

[17] IYER, S., AND DRUSCHEL, P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of ACM SOSP 2001* (Banff, Alberta, Canada, Oct 2001).

[18] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Framework for Analyzing Android I/O Stack Behavior: From Generating the Workload to Analyzing the Trace. *Future Internet 5*, 4 (2013), 591–610.

[19] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC 2013* (San Jose, CA, USA, Jun 2013).

[20] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: Transactional FTL for SQLite Databases. In *Proc. of ACM SIGMOD 2013* (New York, NY, USA, Jun 2013).

[21] KIM, B., KANG, D. H., MIN, C., AND EOM, Y. I. Understanding implications of trim, discard, and background command for emmc storage device. In *Proc. of IEEE GCCE 2014* (Tokyo, Japan, Oct 2014).

[22] KIM, D., AND KANG, S. Partial page buffering for consumer devices with flash storage. In *Proc. of IEEE ICCE-Berlin* (Berlin, Germany, 2013).

[23] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proc. of USENIX FAST 2012* (San Jose, CA, USA, Feb 2012).

[24] KIM, H., AND AHN, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proc. of USENIX FAST 2008* (San Jose, CA, USA, Feb 2008).

[25] KIM, J., MIN, C., AND EOM, Y. I. Reducing Excessive Journaling Overhead with Small-Sized NVRAM for Mobile Devices. *IEEE Transactions on Consumer Electronics 6*, 2 (June 2014).

[26] KIM, M., LEE, S., AND WON, Y. IO Workload Characterization Of Tizen Based Consumer Electronics. In *Proc. of IEEE ISCE 2014* (Jeju, Korea, June 2014).

[27] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proc. of USENIX FAST 2014* (Santa Clara, CA, Feb 2014).

[28] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the Linux virtual machine monitor. In *Proc. of the Linux Symposium* (Ottawa, Ontario, Canada, Jun 2007).

[29] LANG, T., WOOD, C., AND FERNANDEZ, E. B. Database Buffer paging in virtual storage systems. *ACM Trans. on Database Systems 2*, 4 (Dec. 1977), 339–351.

[30] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proc. of USENIX FAST 2015* (San Jose, CA, US, Feb 2015).

[31] LEE, K. Mobile Benchmark Tool (MOBIBENCH). https://github.com/ESOS-Lab/Mobibench.

[32] LEE, K., AND WON, Y. Smart Layers and Dumb Result: IO Characterization of an Android-based Smartphone. In *Proc. of EMSOFT 2012* (Tampere, Finland, Oct 2012).

[33] LI, D., LIAO, X., JIN, H., ZHOU, B., AND ZHANG, Q. A New Disk I/O Model of Virtualized Cloud Environment. *IEEE Transactions on Parallel and Distributed Systems 24*, 6 (June 2013), 1129–1138.

[34] LIU", Z. vfs: add falloc_fl_no_hide_stale flag in fallocate. http://patchwork.ozlabs.org/patch/153251/.

[35] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi Non-Volatile RAM for Low Overhead Persistency Enforcement in Smartphones. In *Proc. of USENIX HotStorage 2014* (Philadelphia, PA, USA, Jun 2014).

[36] PIERNAS, J., CORTES, T., AND GARCÍA, J. M. DualFS: A New Journaling File System Without Meta-data Duplication. In *Proc. of ICS 2002* (New York, NY, USA, 2002).

[37] "REDHAT". "performance of trim command on ext4 filesystem".

[38] SHEN, K., PARK, S., AND ZHU, M. Journaling of Journal Is (Almost) Free. In *Proc. USENIX FAST 2014* (Santa Clara, CA, Feb 2014).

[39] SILBERSCHATZ, A. S., KORTH, H. F., AND SUDARSHAN, S. Database System Concepts. McGraw-Hill.

[40] "SQLITE". www.sqlite.org/famous.html.

[41] TRENDFORCE. Global mobile dram revenue rises 6%, Nov 2014. http://www.dramexchange.com/WeeklyResearch/Post/5/3904.html.