NVM-assisted Non-Redundant Logging for Android Systems

Yuanchao Xu*† and Zeyi Hou*
*College of Information Engineering, Capital Normal University, Beijing, China
†State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
{xuyuanchao, houzeyi}@cnu.edu.cn

***Abstract*—Logging is the most widely used technique to ensure the consistency of persistent data structures across system crashes or power failures in Android systems. Recent proposals are primarily limited to resolving the journaling of journal anomaly in order to mitigate or eliminate explicit redundant logging, and therefore fail to reduce non-trivial overhead induced by implicit redundant logging. In this paper, we analyze the root cause of implicit redundant logging, and then present *i-FTL*, a byte-addressable *non-volatile memory* (NVM) assisted non-redundant logging scheme for Android systems. i-FTL consists of two primary technical ingredients: i) filesystem metadata-aware application-level transaction, ii) NVM-assisted FTL mapping table. The evaluation demonstrates that i-FTL yields approximate 3× performance speedup over the ext4 file system with ordered journal mode and 10% on average performance speedup over X-FTL.**

***Index Terms*—Redundant Logging; Crash Consistency; Transaction; Android System; Non-volatile Memory**

I. Introduction

In recent years, Android-based smartphones have become mainstream mobile devices in place of personal computers. Smartphones can not only be used to give a call or send a message, but also to surf the Internet or pay online. Therefore, data reliability has also caused the extensive concern, aside from performance.

As for data reliability, consistency is one of the key challenges faced by system architects. Data consistency means that all the data within a user operation (e.g., withdraw) must transfer from one consistent state to another consistent state, even if in the case of power failure, system crash, or network interrupt. To ensure data consistency, Android systems take some measures, including using SQLite database with journal support, ext4 journaling file system, and atomic writes at the storage device level.

Unfortunately, these combined measures also induce non-trivial performance overhead. For example [1], a single SQLite operation triggers at least 11 writes to the storage device when used with the ext4 file system. Eighty percent (80%) of the write operations are for purely managerial purposes. The ext4 Journal IO accounts for more than 30% of all writes. In contrast, Metadata write constitutes only 10% of all writes. Kim et al. [2] also demonstrated that storage has greater impact on the whole system performance than network bandwidth in Android systems.

There have been a variety of research efforts to mitigate the overhead induced by crash consistency guarantees. One of the most important efforts is to resolve **explicit redundant logging** problem called *journaling of journal anomaly* (JOJ) [3]. These efforts are based on the fact that the excessive I/O behavior of Android systems originates from the uncoordinated interaction between SQLite and ext4. Thus, system designers should address explicit redundant logging in Android systems in an integrated manner. However, these efforts fail to eliminate the **implicit redundant logging**, which is closely related to the characteristic of NAND flash storage devices.

In this paper, we focus on eliminating implicit redundant logging in Android systems. We also leverage byte-addressable non-volatile memory as a supplement to mitigate write amplification of NAND flash. Our work is closely related to X-FTL [4], both of us use the native out-of-place update feature of NAND flash to simulate copy-on-write technique. However, X-FTL has two shortcomings. First, it only implements individual database operation (i.e., insert, update) atomicity. In practice, each user operation often consists of more than one individual database operations. Second, it does not take metadata feature, such as small I/O and random access, into consideration in its design scheme. These metadata can be categorized into three types: SQLite metadata, file system metadata, and *flash translation layer* (FTL) metadata.

We then consider the characteristic of all layers, including SQLite database (application layer), ext4 file system (operating system layer), and Flash storage device (FTL layer) in state-of-the-art Android systems, and use small-sized NVM to optimize write performance of flash device. By exploiting the feature of out-of-place update and using non-volatile memory technology, we propose *i-FTL*, a NVM-assisted non-redundant logging for Android systems. Our contribution consists of two technical ingredients: i) filesystem metadata-aware application-level transaction, ii) NVM-assisted FTL mapping table. Our proposed scheme exhibits approximate 3× performance speedup over the ext4 file system with ordered journal mode and 10% on average performance speedup over X-FTL.

II. Background and Motivation

A. I/O Stack of Android Systems

Android is a widely used open-source Linux-based software system for smartphones. To date, the typical I/O stack of an Android system consists of SQLite database, ext4 journaling

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.222

file system with ordered mode, and NAND flash storage device (e.g., eMMC). Although emerging byte-addressable non-volatile memory, such as *phase change memory* (PCM), *spin-transfer torque random memory* (STT-RAM), and recently announced 3D XPoint technology [5], offer an intriguing blend of storage and memory, NAND flash is still the best choice in terms of storage capacity per dollar as well as the maturity of software system.

SQLite is a server-less lightweight embedded database management system, the most popular tool for maintaining persistent data in Android systems. Many popular applications including Facebook and Twitter use SQLite to store persistent data. To enable the consistency of data updates across power failures or system crashes, databases and file systems usually adopt *logging*, which is one of the two typical consistency guarantee techniques. This technique aims to recover those interrupted data to be in a consistent state upon a failure by keeping track of changes not yet committed and recording such changes in a data structure known as a "log". SQLite provides six journal modes which can be categorized into *roll-back* mode and *write-ahead log* mode, to ensure application-level data consistency. Although some novel flash-friendly file systems, e.g., F2FS [6], are proposed recently, ext4 is still the most widely used file system in current Android system. For the same reason, ext4 also provides several journal modes, including *data*, *ordered*, and *writeback*. Ordered is the default journal mode in ext4 file system. Similar to writeback mode, ordered mode only logs the updated metadata. The difference between them is that ordered mode requires all the related user data ahead of metadata being persisted to their home locations. Current Android-based smartphones usually use eMMC, a managed NAND flash chip specifically designed for mobile devices. Due to out-of-place update feature of flash, eMMC cannot update data in place. Only after the new-version data are written in free pages successfully, it invalidates the old data. The feature of out-of-place update behaves very similar to copy-on-write, which is another consistency guarantee technique.

B. Overhead of Crash Consistency Guarantee

Crash consistency means the atomicity, consistency, and durability of data updates in the event of a power failure, system crash or interrupted network. Atomicity refers to "all or nothing", i.e. either succeeds completely or fails completely for each single persistent data update. Consistency means that all the data need to transfer from one consistent state to another consistent state. Durability can be achieved by issuing *fsync()* system calls to ensure data durable in flash storage device. The granularity of atomic data update is related to semantics. For example, from a user's perspective, it is expected to complete an application-level operation atomically, which is defined as a transaction. There are two typical techniques to ensure consistency, write-ahead logging and copy-on-write. As mentioned above, in Android-based smartphones, both SQLite and ext4 use write-ahead logging technique to ensure data consistency of database and file system, respectively.

Unfortunately, uncoordinated interaction between SQLite and ext4 leads to journaling of journal anomaly issue, which means that ext4 journals the metadata of SQLite journal file for each database transaction. SQLite is not sure of whether file system enables crash consistency. Likewise, ext4 is also not sure of whether database adopts the transaction mechanism. As a consequent, both SQLite and ext4 are likely to employ consistency guarantee techniques simultaneously for identical data including user data and metadata, thereby inducing unnecessary overheads. The root cause of JOJ is that each layer distrusts with each other, thereby leading to serious redundant logging.

In recent years, there have been a number of efforts to mitigate or eliminate the problem of JOJ. The main measures are to reduce the number of *fsync()* system calls in SQLite, to mitigate the overhead of a single *fsync()* system call by modifying ext4 file system, or to eliminate the root cause for JOJ by exploiting the direct I/O to bypass file system when storing the logs [7]. However, these measures only remove explicit redundant logging but fail to eliminate implicit redundant logging in Android systems, because we neglect the native feature of out-of-place update for flash storage.

III. Design Overview

In this section, we have an overview of the design how to eliminate implicit redundant logging. Fig. 1 illustrates the architecture of Android systems. We modify ext4 file system and FTL controller to implement multi-page atomic writes as well as application-level transaction. To mitigate write amplification, we add STT-RAM, a small-sized byte-addressable non-volatile memory to store ext4 filesystem metadata and FTL mapping table.

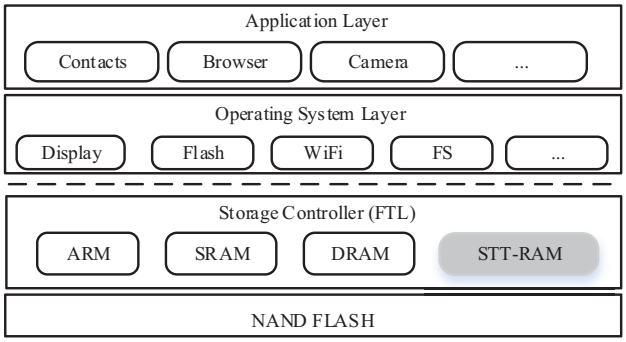

Fig. 1. Architecture overview of Android systems.

A. Multi-page Atomic Writes

To implement non-redundant logging, we design the I/O stack of Android systems from the systematic perspective. Theoretically, only one layer rather than all layers need to implement application-level transaction. This layer may be the application itself, file system, or FTL. However, the upper the layer, the more overhead, since the upper layer journaling incurs more metadata than lower layer journaling. For example,

if SQLite journal mode is turned off, those metadata related to SQLite journal are eliminated. We can enable *data* journal mode in ext4 filesystem to ensure the consistency of SQLite database. If we further disable ext4 journal mode, the journal (copy) of SQLite database are also eliminated completely. As a result, the consistency of all the data is ensured via multi-page atomic writes at the flash storage device level. Due to the non-overwrite property of NAND flash, each page maintains two versions before the old-version page is erased. Thus, the non-overwrite (also called out-of-place update) feature is very similar to copy-on-write. To achieve the application-level transaction, storage controller (also called FTL) has to be modified. First, after the new page is programmed, the old page should not be invalidated until the transaction related to the old page is committed successfully. In order to track these old pages, additional mapping table needs to be added. In addition, garbage collection should be deferred in order to avoid erasing those old pages belonging to a committing transaction. The recently proposed eMMC standard specification defines the feature of *reliable write*, which is similar to atomic write from the perspective of semantics. However, it does not discuss any implementation. We measure the performance difference between reliable write and normal write. Surprisingly, the reliable write is slowed down by 50% than normal write. Therefore, it demonstrates that the performance of atomic write is closely related to its implementation.

B. Application-Level Transaction

Although journaling file systems provide transaction mechanism and storage controllers provide multi-page atomic writes, additional work is needed if we want to ensure the application-level transaction, since users always expect to accomplish a user-level operation atomically, e.g., withdraw or deposit. Generally, a user-level operation involves in a variety of data updates, including SQLite metadata (e.g., database dictionary), filesystem metadata (e.g., inode table), and FTL metadata (e.g., mapping table). Thus, we need to identify the application-level transaction that all the data updates belong to, because after a system crash, power failure, or transaction aborts, all of the user data and corresponding metadata within this transaction should rollback to a previous consistent state. In essence, an application-level transaction may be implemented in any layer via logging or shadowing. The most efficient way is multi-page atomic write implemented in FTL layer, since it produces the minimal metadata.

IV. Implementation

We modified the ext4 file system and eMMC device driver in Linux Kernel 4.1.15. We also modified FTL controller in Jasmine OpenSSD development platform [8] to support atomic writes.

A. Application-Level Transaction

In a transaction system, programmers must define the range of an atomic writes region regardless of whatever layer implement the functionality of a transaction. In current system, both SQLite and ext4 provide transaction support. If multi-page atomic writes is enabled in FTL layer, the journal of all the upper layer can be turned off. It simplifies the transaction-based programming. More importantly, it makes the implementation of transactions more lightweight and efficient. A snippet of an application-level transaction is illustrated in Fig. 2. Programmers only need to add two primitives, **TX_BEGIN** and **TX_END**, to identify the range of a failure-atomic write.

```
TX_BEGIN(Tx-ID) {
  db_exec( "update account set income = 300 where acc_id=102" );
  db_exec( "insert table account(acc_id,balance) values(102,1500)" );
} TX_END(Tx-ID)
```

Fig. 2. An example of the snippet of an application-level transaction.

An application-level transaction is generally divided into several block I/O writes in file system layer. Likewise, each block write operation may be involved in a few pages write operation in FTL layer. To implement multi-pages atomic writes within an application-level transaction, it requires to differentiate which transaction each page belongs to. When a new application-level transaction is created, a unique transaction identification **Tx-ID** assigned to this transaction. All pages within this transaction own the same **Tx-ID**. **Tx-ID** will be passed from application layer to FTL layer. When FTL receives information about the commit or abort of a transaction, storage controller can decide whether to accept or abandon all the updates within this transaction. As Fig. 3 shows, there are five blocks in the filesystem layer from two application-level transactions, three of which belong to Tx-0, and the other two belong to Tx-1. From the perspective of FTL layer, five pages belong to Tx-0 and four pages belong to Tx-1.

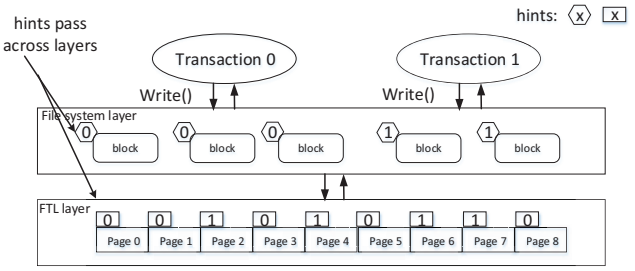

Fig. 3. Hint information passing across layer for application-level transactions

Kim et al. [2] found that most I/Os in Android systems are related to SQLite database operations. For example, SQLite and its journal file are responsible for approximately 90% of the write I/O requests in both Facebook and Twitter apps. The number of I/O requests from SQLite database and its journal file is about equal. If the journal is turned off, the number of I/O requests will be reduced by half. In ext4 file system, writes to the ext4 journal block constitute 30% of all writes. 10% and 60% of all writes are for the metadata and

data, respectively. Thus, filesystem journaling has non-trivial impact on the whole system performance. Although metadata is only one-sixth of data in the number of requests, metadata is smaller than data and only tens of bytes per I/O request, hence, it induces severe write amplification since flash device is programmed at the page-level granularity (e.g., 4KB). To mitigate this issue, we add a small-sized STT-RAM to store file system metadata including inode table and superblock. STT-RAM is non-volatile and byte-addressable, hence, metadata may be updated in place.

To tell FTL layer whether an I/O is data or metadata, we adopt a mechanism called *data tag*, to distinguish the data type. The changes are made in the file system and device driver. With data tag, FTL can identify the type of I/Os and then make correct decisions of data placement.

B. Log Mapping Table

Due to the out-of-place update characteristic of flash, the new data have to be written to a new free page first, and then the mapping table of FTL is updated. The old data will stay in flash till corresponding block in which the old data resided is erased. This process is called *garbage collection* (GC). Because the mapping information between *logical block address* (LBA) and *physical block address* (PBA) is lost, the old data become invalid.

To leverage these invalid pages for transaction recovery, we construct a new mapping table called log mapping table (referred to as **logFTL**), which is very similar to existing mapping table (referred to as **dataFTL**) in FTL. **logFTL** is used to record all the mapping relationship between logical pages and invalid pages, thereby simulating copy-on-write technique to enable page-level crash consistency. As shown in Fig. 4, compared to **dataFTL**, **logFTL** adds two additional fields, TX-ID and TX-state. TX-ID represents the number of application-level transaction that a page belongs to. TX-state indicates the state of a transaction. 0 indicates that transaction is committing, while 1 indicates that transaction is committed. Besides, unlike **dataFTL**, **logFTL** is append-only.

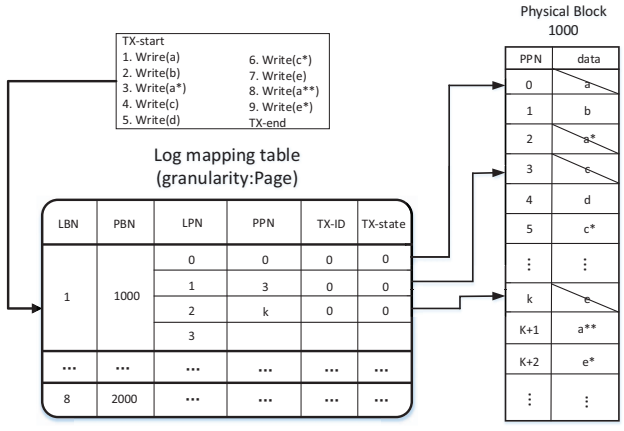

Fig. 4. Log mapping table. Nine write operations belong to transaction Tx0.

Fig. 4 illustrates the working process of **logFTL** when a series of write operations within a transaction is executing. Suppose that all the data related to this transaction belong to a block, though each of them may belong to different pages. The logical block number of this block is 1, and the physical block number of this block is 1000. The process for FTL controller is as follow: First, write *a* to a page, whose *physical page number* (PPN) is 0; Then, write *b* to adjacent page #1; Next, update data *a*. Because in-place update is not allowed for flash device, the new data *(*a*) have to be written to the next free page #2. After the new data are written successfully, FTL will invalidate the data of page #0. Because current transaction has not yet been committed, we must record the information about the invalid page (i.e., PPN 0) in the **logFTL**, as shown in the first entry in Fig. 4. The rest of write operations within this transaction will go on. After a transaction is committed, all the entries related to this transaction in **logFTL** may be discarded.

As we know, each I/O write request will accomplish the updates of two mapping tables including **dataFTL** and **logFTL**. Although the granularity is only one entry with tens of bytes, FTL also has to find a new page frequently to store these FTL metadata. Suppose that each page size is 4KB, write amplification coefficient is about 100. Obviously, frequent small-sized metadata updates induce non-trivial performance overhead. If the overhead of data movement and block erasement is considered, the negative impact of write amplification is more serious. Therefore, we also store all the mapping table in a small-sized STT-RAM. This design can not only continue to leverage the advantage of flash in capacity per dollar, but also overcome the disadvantage of out-of-place at page unit.

C. Lazy Garbage Collection

The mapping information lifetime of each page in **logFTL** depends on the state of the transaction this page belongs to. When the free pages of flash are below a predefined threshold, FTL controller will do garbage collection by erasing some blocks filled with all or partial invalid pages. Traditional garbage collection mechanism only guarantees those valid page copied to a new block, ignoring those invalid pages. However, in our design, even if some pages are marked as invalid, they should not be skipped during garbage collection if the transactions these pages belong to have not been committed, since these pages would be used for data recovery after a system failure or transaction abort. Therefore, we propose a new garbage collection mechanism called **lazy garbage collection**.

As shown in Fig. 5, under the traditional garbage collection, FTL controller scans all the data blocks in flash to find out those blocks filled with all or nearly all invalid pages, then erases them. For example, FTL decides to erase data block, whose PBN is 1000. There are several valid pages including page#1, page#4, page#5, page#(k+1), page#(k+2) in this block. These pages are copied to the physical block, whose PBN is 3500 first, then all the data in this physical block #1000 will be erased. If so, we find that those *invalid* pages which may be needed for crash recovery have also been

erased. To address this issue, we fetch a physical block (e.g., Log-Block #6500) from the reserved storage space to store these temporary invalid pages. Before FTL controller erases physical block #1000, it will first scan all pages including invalid pages. If an invalid page is found, e.g., page #0, FTL controller will go for checking the state of this transaction the invalid page belongs to. If the value of TX-state is zero, it indicates that this transaction has not been committed yet, thus page #0 still needs to be kept. Then, this page will be copied to Log-Block 6500 and the corresponding PBN and PPN of this page in **logFTL** will also be updated.

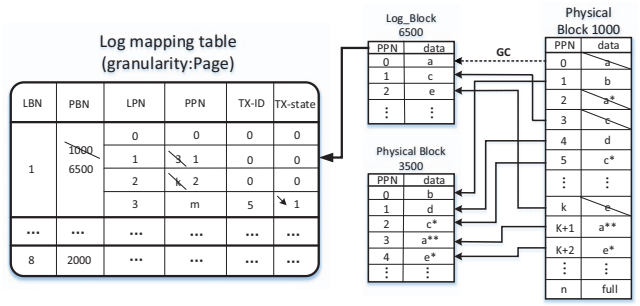

Fig. 5. Lazy garbage collection. All the invalid pages are not permitted to be erased until those corresponding application-level transactions are committed successfully.

D. Transaction Recovery

Fig. 6 shows several possible crash points. In essence, there is only one situation in our design. Transaction recovery includes the recovery of real data block and **dataFTL**. After a transaction aborts, **logFTL** will be checked first to determine which pages of this transaction have been modified. Then by scanning the **logFTL**, the corresponding physical page numbers of these dirty pages just before the point this transaction begins will be obtained. Next, some entries of the **dataFTL** will be recovered from the old-version physical page numbers. If a power failure or system crash occurs, the whole **logFTL** will be scanned since there may be more than one transactions uncommitted. The recovery process is similar to the one when a transaction aborts. The basic process is described in Algorithm 1.

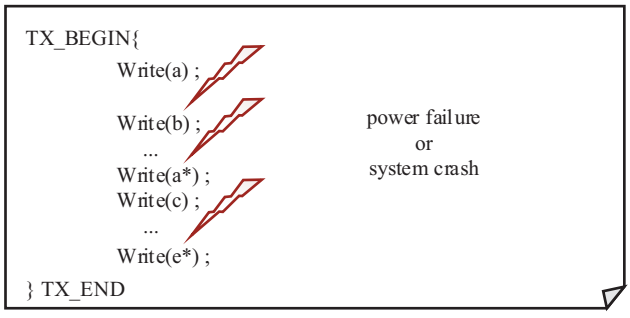

Fig. 6. Possible crash points during an application-level transaction running.

Algorithm 1 Data Recovery after a failure or abort

```
/*check logFTL with logical page number */
ADDR = logical_addr;
find = 0;
repeat
   find = check(log_mapping_table, ADDR);
   if find == 1 then
      physical addr = search(ADDR);
      return physical_addr;
   else
      ADDR++;
   end if
until find == 0
/*update dataFTL with physical page number*/
repeat
   find = check(existing_mapping_table, logical_addr);
   if find == 1 then
      /*update relevant mapping relationship*/
      update(logical_addr, physical_addr);
      else
      ADDR++;
   end if
until find == 0
```

V. Evaluation

A. Experimental Setup

Platform. We carried out the experiments on Jasmine OpenSSD platform [8], which has Indilinx controller with 87.5MHz ARM processor, and the controller contains 96KB SRAM to store the firmware and 64MB Mobile SDRAM to store metadata such as mapping tables. Because memory-bus attached STT-RAM devices are not yet available, we use a region of SDRAM as a proxy. The capacity of STT-RAM is 8MB. The capacity of NAND flash chip is 64GB and the page size is 4KB. The host machine is a Linux system with 4.1.15 kernel running on Intel core i7-860 2.8GHz processor and 4GB DRAM. The version of SQLite used in this paper is 3.7.10. Both ext4 file system journal and SQLite journal are turned off.

Benchmark. Mobibench (mobile benchmark) [9] is an I/O workload generator which is specifically designed for Android workload generation. Users can configure SQLite journaling option, and various file system I/O options, e.g., direct I/O, synchronous I/O, and etc. We perform each of the insert, update and delete operations 10,000 times and measure the throughput.

B. Performance Evaluation

Mobibench provides flexible configure for SQLite database. In this paper, we only evaluates three Android systems. The first is the baseline system, in which SQLite journal is turned off and ext4 file system journal is set to ordered mode. The second is the Android system based on X-FTL. The third is our proposed design based on i-FTL.

As illustrated in Fig. 7, given only one thread running, with i-FTL, insert, update, and delete operations exhibits dramatic 2.77×, 2.79×, and 3.30× performance (transactions/sec), respectively compared to the baseline system. It demonstrates that even if SQLite journal is turned off, ext4 ordered journaling still induces non-trivial overhead. We also examine the performance of multi-threaded applications. The results are illustrated in Fig. 8. It indicates that, with the increase of the number of threads, i-FTL can yield better performance than X-FTL. It can achieve performance improvement by 8.2% to 15.6% over X-FTL.

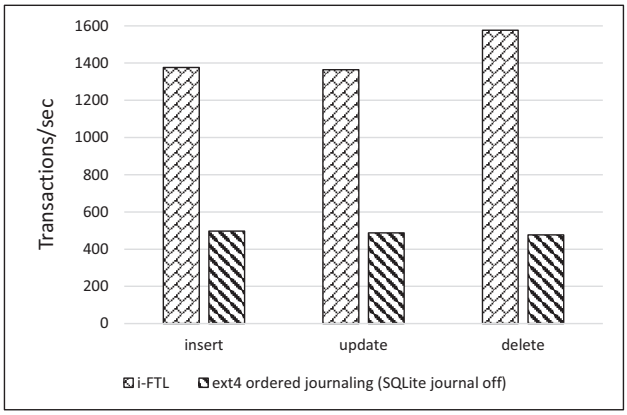

Fig. 7. Performance comparison of insert, update, and delete operations between X-FTL and i-FTL under only one thread.

C. Overhead Analysis

Our proposed design, i-FTL, introduces two extra overheads. One is from newly introduced log mapping table, while the other is from transaction recovery.

Log mapping table needs to occupy additional storage space, whose size is related to the number of concurrent transactions and the number of updates pages per transaction. For personal mobile devices, the number of transactions running in parallel is relatively limited compared to servers in datacenters, thus, small-sized storage space is enough. Aside from storage space, more serious overhead is non-trivial write amplification due to frequent updating for log mapping table (i.e., **logFTL**) and existing mapping table (i.e., **dataFTL**). However, this issue is eliminated efficiently because we use small-sized byte-addressable non-volatile memory to store mapping table.

With respect to transaction recovery, we only care about those updated pages of all the uncommitted transactions. For example, if a page #10 within a transaction Tx-50 is updated, the correspondingly entry of this page in **dataFTL** needs to be recovered. Besides, we need to mark page #10 as valid again. For the same reason as **logFTL**, there is not write amplification due to in-place update of STT-RAM.

In this work, we leverage the native feature of out-of-place update to simulate copy-on-write. It does not incur additional overhead. Therefore, the whole overhead is negligible. In other words, we achieve application-level crash consistency guarantee without introducing too much overhead and without requiring any logging.

VI. Related Work

Journaling of Journal. Many recent efforts have been directed towards the reduction of the overheads of journaling of journal anomaly [3], [7], [10]–[12]. Jeong et al. [3] proposed to eliminate unnecessary *metadata* journaling for the file system. Kim et al. [10] developed a multi-version B-tree with lazy split to minimize the write traffic caused by *fsync()* call via weaving the crash recovery information within the database itself instead of maintaining a separate file. Shen et al. [11] proposed two techniques to ensure a single I/O operation on the synchronous commit path and adaptively allow each file to have a custom journaling mode. Yang et al. [12] outlined the architectural issues that can arise when one or more logs are stacked atop an FTL, which offer similar functionalities. All of these proposals aim to mitigate journaling of journal anomaly, which is considered as *explicit* redundant logging. Lee et al. [7] exploited the direct I/O to suppress the file system interference in storing the logs, thereby successfully eliminating the root cause of journaling of journal anomaly. However, there is no consideration of how to eliminate the *implicit* redundant logging.

Atomic Writes. The *atomic write* FTL by Park et al. [13] is the first study to exploit the out-of-place update characteristics of flash storage. In particular, this work deals with supporting atomic write of multiple pages specified in a single write call like *write(p1;::;pn)*. Prabhakaran et al. proposed a transactional FTL called *txFlash* to support atomic writes for file system journaling [14]. In addition to supporting atomicity of multi-page writes, TxFlash provides isolation among multiple atomic write calls. Ouyang et al. proposed another transactional SSD design which could support both read and write operations in transactions [15]. Their design is based on log-based FTL and it stores an extra one-bit commit flag with the value 0 or 1 to determine whether the last transaction was committed before crashes or not. But this design also restricts that there is at most one running transaction. LightTx [16] [17] utilized a zone-based scheme to scan and recover transaction states and provided flexible isolation levels for transaction processing. Möbius was a recently proposed transactional SSD design [18], which provided different types of transactional primitives to support static and dynamic transactions separately. The recently proposed eMMC standard [19] also presented reliable write.

The common limitation of these approaches is that the atomicity of a multi-page writes can be supported only on a per-call basis or only when the whole set of pages are flushed from the buffer pool at once. They do not provide application-level crash consistency. In other words, they do not discuss how to implement application-level transaction by leveraging multi-pages atomic writes feature. CFS [20] provided application-level crash consistency on transactional flash storages, however, it only emphasized on the file system. Our work is closely related to X-FTL [4]. However, X-FTL only

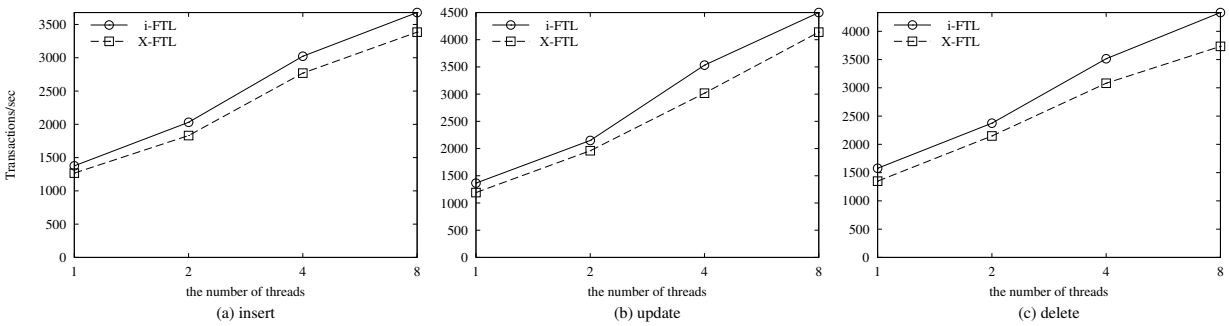

Fig. 8. Performance comparison of three database operations between X-FTL and i-FTL as the number of threads increases.

implements the atomicity of individual database operation (i.e., insert, update) and also does not take metadata feature into consideration in their design. Our work can ensure application-level crash consistency without logging at the expense of lightweight write amplification.

VII. Conclusions

In Android systems, SQLite and ext4 are the default database and file system, respectively. Most I/Os are related to SQLite, meanwhile, ext4 journal block constitute 30% of all writes (data, metadata, and journal). In this paper, we leverage the native out-of-place feature of NAND flash to simulate copy-on-write technique to enable multi-page atomic writes. We present *i-FTL*, a NVM-assisted non-redundant logging scheme for Android systems. i-FTL has two primary technical contributions: i) filesystem metadata-aware application-level transaction, ii) NVM-assisted FTL mapping table. Compared to the baseline system with ext4 ordered mode and SQLite journal disabled, i-FTL yields approximate 3× performance speedup. Because we store file system metadata and FTL mapping table in byte-addressable memory instead of page-addressable memory, i-FTL can achieve 10% on average performance speedup over X-FTL.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful comments. This work is supported by Beijing Natural Science Foundation (No.4143060) and State Key Laboratory of Computer Architecture (No.CARCH201503).

References

[1] K. Lee and Y. Won, "Smart layers and dumb result: IO characterization of an Android-based smartphone," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 23–32.

[2] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, p. 14, 2012.

[3] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones." in *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX, 2013, pp. 309–320.

[4] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: transactional FTL for SQLite databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 97–108.

[5] Intel and Micron, "Intel and Micron produce breakthrough memory technology," https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, July 2015.

[6] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 273–286.

[7] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, "WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly," in *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX, 2015, pp. 235–247.

[8] "Jasmine openSSD platform," http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform, July 2015.

[9] C. Kim, J. Jung, T.-K. Ko, S. W. Lim, S. Kim, K. Lee, and W. Lee, "Mobilebench: A thorough performance evaluation framework for mobile systems," in *Proceedings of The First International Workshop on Parallelism in Mobile Platforms, in conjunction with HPCA-19*, 2013.

[10] W.-H. Kim, B. Nam, D. Park, and Y. Won, "Resolving journaling of journal anomaly in android I/O: multi-version B-Tree with lazy split." in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*. USENIX, 2014, pp. 273–285.

[11] K. Shen, S. Park, and M. Zhu, "Journaling of journal is (almost) free." in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*. USENIX, 2014, pp. 287–293.

[12] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, 2014, pp. 43–52.

[13] S. Park, J. H. Yu, and S. Y. Ohm, "Atomic write FTL for robust flash file system," in *Proceedings of the Ninth International Symposium on Consumer Electronics*. IEEE, 2005, pp. 155–160.

[14] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash." in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX, 2008, pp. 147–160.

[15] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 301–311.

[16] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *Proceedings of the IEEE 31st International Conference on Computer Design*. IEEE, 2013, pp. 115–122.

[17] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "High-performance and lightweight transaction support in flash-based SSDs," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2819–2832, 2015.

[18] W. Shi, D. Wang, Z. Wang, and D. Ju, "Möbius: A high performance transactional SSD with rich primitives," in *Proceedings of the 30th Symposium on Mass Storage Systems and Technologies*. IEEE, 2014, pp. 1–11.

[19] JEDEC Standard, "Embedded multi-media card (eMMC) electrical standard (5.0)," *JESD84-B50*, 2013.

[20] C. Min, W.-H. Kang, T. Kim, and S.-W. Lee, "Lightweight application-level crash consistency on transactional flash storage," in *Proceedings of the USENIX Annual Technical Conference*. USENIX, 2015, pp. 221–234.