X-FTL: Transactional FTL for SQLite Databases

Woon-Hak Kang[†] Sang-Won Lee[†] Bongki Moon[‡]

Gi-Hwan Oh[†] Changwoo Min[†]

[†]College of Info. and Comm. Engr.
Sungkyunkwan University
Suwon, 440-746, Korea
{woonagi319,swlee,wurikiji,multics69}@skku.edu

[‡]School of Computer Science and Engineering
Seoul National University
Seoul, 151-744, Korea
bkmoon@snu.ac.kr

ABSTRACT

In the era of smartphones and mobile computing, many popular applications such as Facebook, twitter, Gmail, and even Angry birds game manage their data using SQLite. This is mainly due to the development productivity and solid transactional support. For transactional atomicity, however, SQLite relies on less sophisticated but costlier page-oriented journaling mechanisms. Hence, this is often cited as the main cause of tardy responses in mobile applications.

Flash memory does not allow data to be updated in place, and the copy-on-write strategy is adopted by most flash storage devices. In this paper, we propose *X-FTL*, a transactional flash translation layer(FTL) for SQLite databases. By offloading the burden of guaranteeing the transactional atomicity from a host system to flash storage and by taking advantage of the copy-on-write strategy used in modern FTLs, *X-FTL* drastically improves the transactional throughput almost for free without resorting to costly journaling schemes. We have implemented *X-FTL* on an SSD development board called OpenSSD, and modified SQLite and `ext4` file system minimally to make them compatible with the extended abstractions provided by *X-FTL*. We demonstrate the effectiveness of *X-FTL* using real and synthetic SQLite workloads for smartphone applications, TPC-C benchmark for OLTP databases, and FIO benchmark for file systems.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Transaction procesing*

General Terms

Design, Performance, Reliability

Keywords

SQLite, Transactional Atomicity, Flash Storage Devices, Flash Translation Layer, Copy-On-Write

1. INTRODUCTION

An update action in a database system may involve multiple pages, and each of the pages in turn usually involves multiple disk sectors. A sector write is done by a slow mechanical process and can be interrupted by a power failure. If a failure occurs in the middle of a sector write, the sector might be only partially updated. A sector write is thus considered non-atomic by most contemporary database systems [4, 16].

Atomic propagation of one or more pages updated by a transaction can be implemented by shadow pages or physical logging. However, the cost will be considerable during normal processing. Most commercial database systems adopt a solution based on physiological logging, which is more I/O efficient in handling updates and more flexible in the locking granularity. On the other hand, *SQLite*, developed as an embedded database system, faces additional challenge to limit the scale of its code base. Hence SQLite relies on costlier but less sophisticated mechanisms similar to shadow paging.

In order to support transactional atomicity, SQLite processes a page update by copying the original content to a separate *rollback* file or appending the new content to a separate *write-ahead log*. This is often cited as the main cause of tardy responses in smartphone applications [11, 12]. According to a recent survey [12], such popular applications as Facebook, Gmail, twitter, web browsers, and even Angry birds game manage their data using SQLite on smartphone platforms like Android, and approximately 70% of all write requests are for SQLite databases and related files. Considering the increasing popularity of smart mobile platforms, improving the I/O efficiency of SQLite is a practical and critical problem that should be addressed immediately.

Most contemporary mobile devices, if not all, use flash memory as storage media to store data persistently. Since flash memory does not allow any page to be overwritten in place, a page update is commonly carried out by leaving the existing page intact and writing the new content into a clean page at another location [9, 13]. This strategy is called *copy-on-write (CoW)*. Whether it is intended or not, the net effect of copy-on-write operations by a flash memory drive remarkably resembles what the shadow paging mechanism achieves [15]. This provides an excellent opportunity for supporting atomic update propagation almost for free.

In this paper, we present *X-FTL* that (1) realizes low-cost atomic update for individual pages, (2) provides the awareness of database semantics in page updates to support the atomicity of transactions, and (3) exposes an extended abstraction (and APIs) to not only SQLite but also other upper layer applications such as a journaling file system.

The key contributions of this work are summarized as follows.

- *X-FTL* provides an extended abstraction at the storage device level and guarantees the atomicity of transactions and the durability of committed changes almost at no cost. SQLite and journaling file systems can build on *X-FTL* to minimize the overhead of transactional support and metadata journaling.

- When running on top of *X-FTL*, a journaling file system may turn journaling off and can still achieve the same level of consistency provided by full journaling. This indicates that a lightweight transactional file system can be developed without taking extra burden for duplicate data and metadata writes and synchronous write ordering [10, 23, 24].

- We have implemented *X-FTL* on an open SSD development hardware platform called *OpenSSD* by enhancing its FTL code with the *X-FTL* features presented in this paper. We have demonstrated SQLite and `ext4` file system can take advantage of *X-FTL* with only minimal changes in their code.

The rest of the paper is organized as follows. Section 2 describes the transactional support and I/O efficiency of SQLite and presents the motivation of our work. Section 3 reviews the existing techniques for atomic updates and journaling file systems and flash translation layer (FTL) approaches. In Section 4, we present the design principles and the architecture of *X-FTL* and discuss its advantage for SQLite databases. Section 5 presents the implementation details of the *X-FTL* architecture. In Section 6, we evaluate the performance impact of *X-FTL* on SQLite databases as well as file systems. Lastly, Section 7 summarizes the contributions of this paper.

2. MOTIVATION

2.1 Transactional Support in SQLite

For its portability on a wide spectrum of platforms, SQLite makes a few assumptions on the underlying hardware and operating system. While some of them are optimistic, some are pessimistic. One of the critical and pessimistic assumptions is that a disk sector write is not atomic [4]. If a failure occurs in the middle of a sector write, it might be that part of the sector is modified, while the rest is left unmodified.

In order to support the atomicity of transaction execution without the atomicity of a sector write, SQLite operates usually in either *rollback* mode [4] or *write-ahead log* mode [6]. If a transaction updates a page in *rollback* mode, the original content of the page is copied to the rollback journal before updating it in the database, so that the change can always be undone if the transaction aborts. The opposite is done in *write-ahead log* mode. If a transaction updates a page in *write-ahead log* mode, the original content is preserved in the database and the modified page is appended to a separate log, so that any committed change can always be redone by copying it from the log. The change is then later propagated to the database by periodical checkpointing. The delayed propagation in *write-ahead log* mode allows a transaction to maintain its own *snapshot* of the database and enable readers to run fast without being blocked by a writer.

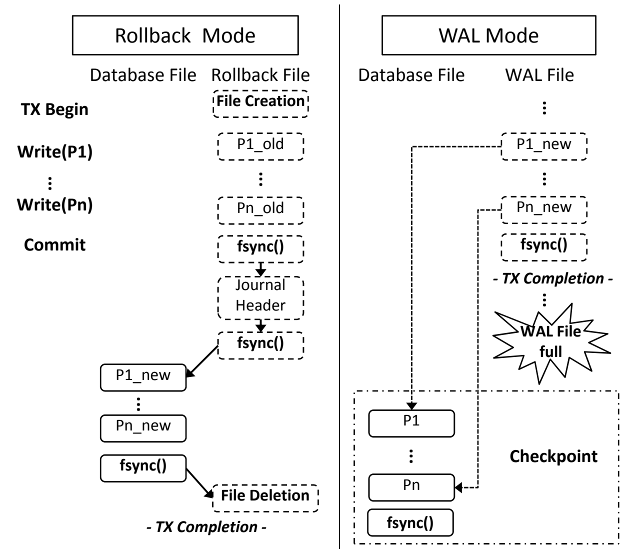


Figure 1: SQLite Journal Modes

Other notable features of SQLite are summarized below for the background information [4]. SQLite is a software library that implements a serverless transactional SQL database engine. It manages tables and indexes in a single database file on an underlying file system such as `ext4`. SQLite adopts *force* and *steal* policies for buffer management. When a transaction commits, all the pages updated by the transaction are force-written to a stable storage using the `fsync` command. When the buffer runs out of free pages, even uncommitted updates can be written to a stable storage.

2.2 I/O Efficiency of SQLite

The pessimistic side of SQLite further assumes that, for a file that grows in size, the file size is updated before the file content. This requires SQLite to do extra work to ensure that the metadata and content of a file are written to persistent media in particular order, so that a failure between the two events does not cause inconsistency in the database. One popular approach to imposing write ordering is to execute a sync operation as a barrier between a pair of ordered write operations [22].[1] This may be the reason why SQLite is criticized for excessive use of the sync file system call [11].

SQLite assumes that a file deletion is atomic. This assumption is essential for SQLite to maintain consistency in the database, because the presence of a *hot* rollback journal in *rollback* mode, for example, indicates that a transaction was committing when a failure occurred. This assumption cannot be made without support for atomic update of file metadata, which SQLite again assumes is guaranteed by the metadata journaling of an underlying file system.

[1]Most SCSI or SATA drives support the force unit access (FAU) command to bypass the internal cache for reading and writing.

The I/O behaviors of SQLite, as depicted in Figure 1, depend on which mode it runs in. If SQLite runs in *rollback* mode, a journal file is created and deleted whenever a new transaction begins and ends. This increases I/O activities significantly for updating metadata. If SQLite runs in *write-ahead log* mode, a log file is reused and shared by many transactions until the log file is cleared by a checkpointing operation. Thus, the overhead of updating metadata is much lower when SQLite run in *write-ahead log* mode. Another aspect of I/O behaviors is how frequently files are synced. SQLite invokes `fsync` system calls more often when it runs in *rollback* mode than in *write-ahead log* mode. Since the header page of a journal file requires being synced separately from data pages, SQLite needs to invoke at least one more `fsync` call for each committing transaction.

2.3 Problem Statement

SQLite relies heavily on the use of *rollback* journal files and *write-ahead log* as well as frequent file sync operations for transactional atomicity and durability. The I/O inefficiency of this strategy is the main cause of tardy responses of applications running on SQLite. Our goal is to achieve I/O efficient, database-aware transactional support at the flash based storage level so that SQLite and similar applications can simplify their logics for transaction support and hence run faster. We have developed a new flash translation layer (FTL) called *X-FTL*. With *X-FTL*, SQLite and other upper layer applications such as a file system can achieve transactional atomicity and durability as well as metadata journaling with minimum overhead and redundancy.

3. RELATED WORK

By taking advantage of the *copy-on-write* mechanism of flash storage, *X-FTL* can efficiently guarantee that all the pages updated by a transaction are successfully propagated or no pages are written at all upon a failure. In this regard, three types of existing work are closely related to *X-FTL*: shadow paging technique [15], journaling file systems [7, 19, 21, 25], and a few FTL techniques for atomic write of file system journal data [17, 18, 20]. Now, let us briefly explain each work and compare it with *X-FTL*.

3.1 Shadow Paging Technique

In the shadow paging approach [15], when a transaction updates pages, a new version of each page is created, instead of overwriting the existing one. The reason for preserving the old copy (called *shadow page*) is to make page updates atomic by replacing a long page write process with a short pointer manipulation. Hence, the database system needs to keep two page tables: one for new versions and the other for old copies. In fact, each mode of *rollback* and *write-ahead log* in SQLite can be regarded as a variant of this shadow paging technique.

In this respect, our *X-FTL* is conceptually no less than offloading the shadow paging overhead from the database engine to the flash storage layer. This simple offloading opens up new opportunities for implementing the transactional atomicity inside flash storage much more efficiently because most contemporary FTLs rely on *copy-on-write* for update operations. That is, by extending an existing FTL which does not overwrite an old copy, atomic update of pages can be achieved at the flash storage level at no additional cost.

Besides, with *X-FTL*, unlike the traditional shadow paging technique, SQLite-like applications are relieved from the burden of managing two separate page tables and reclaiming the space occupied by old copies.

3.2 Journaling File Systems

Modern file systems take a journaling approach to guarantee the consistency of both metadata and data [10, 19, 21]. The consistency provided by the journaling file systems is aimed at updating a fixed set of pages atomically, which is not semantically sufficient to support more general database transactions. Recently, a few flash-aware journaling file systems such as JFFS [25] and YAFFS [7] have been developed. These file systems, when writing data or metadata to storage upon explicit sync calls by users or upon periodic flush, do not overwrite the original copies. Instead, the new copies are saved in a separate journal area until they are propagated to their original location by checkpointing. This way they achieve data consistency at the expense of increased IOs required for journaling.

In contrast, *X-FTL* provides at least the same level of consistency as provided by modern journaling file systems and can boost transaction performance significantly without resorting to redundant journal files.

3.3 FTL Approaches

Recently, mainly from the storage and file system communities, a few interesting suggestions have been made to support atomic updates on flash-based storage devices [17, 18, 20]. Like journaling file systems described above, those FTL approaches focus more on updating a fixed set of pages atomically.

The *atomic write FTL* by Park *et al.* [18], among the first studies for supporting atomic write in flash storage, exploits 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(`$p_1, .., p_n$`)`. Each write call leaves a commit record after writing all the data pages so that they can be rolled back together if necessary. This approach was recently applied to FusionIO flash SSDs, which helped MySQL achieve atomicity of multiple page writes [17].

Prabhakaran *et al.* proposed a transactional FTL called *txFlash* to support atomic writes for file system journaling [20]. In addition to supporting atomicity of mult-page writes, TxFlash provides isolation among multiple atomic write calls by ensuring that no conflicting writes are issued. It relies on a new commit protocol called Simple Cyclic Commit (SCC) to eliminate the need of a separate commit record for each write, which would impose a write ordering and additional latency.

The common limitation of these approaches is that the atomicity of a multi-page write 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. This requirement clearly contradicts the steal policy of buffer management SQLite and most database systems adopt for performance reasons. Unlike these FTL approaches, *X-FTL* allows for any data page to be written at any time and still supports atomicity of any group of pages belonging to a transaction. Consequently, we can realize flexible and efficient support for general transaction semantics with *X-FTL*, which can also be used to provide atomic writes required in a journaling file system.

3.4 Different Media

It has been recently reported that SQLite database is one of the main performance bottlenecks in smartphone applications [11, 12]. However, no solution has been proposed yet to address the performance problem of SQLite, except for a few suggestions that DRAM or PRAM be utilized as storage for shorter latency and higher bandwidth instead of flash-based storage [11]. The *X-FTL* solution we propose assumes only flash storage which is more common, more economical and has strong market ground. In this respect, *X-FTL* is, as far as we know, the first flash-based approach to tackling SQLite performance concerns.

4. ARCHITECTURE OF X-FTL

In the previous sections, we have reviewed how SQLite and a journaling file system achieve atomic page updates on a storage device that does not support it natively, and identified heavy redundancy in page writes (required by SQLite in the *rollback* or *write-ahead log* mode and by a file system for data and metadata journaling) as the main cause of long latency in update operations. In this section, we present a new method called *X-FTL* that supports atomic page updates at the flash based storage level, so that upper layers such as SQLite and a file system can be freed from the burden of heavy redundancy of duplicate page writes. The design principles and the abstractions of *X-FTL* will be presented.

4.1 Design Principles

In order to have a flash based storage layer provide a transactional support, we have designed a new flash translation layer called *X-FTL*. The design objectives of *X-FTL* are threefold. First, *X-FTL* takes advantage of the copy-on-write mechanism adopted by most flash-based storage devices [9], so that it can achieve transactional atomicity and durability at low cost with no more redundant writes than required by the copy-on-write operations themselves. This is especially important for SQLite that adopts the *force* policy for buffer management at commit time of a transaction. Second, *X-FTL* aims at providing atomic propagation of page updates for individual pages separately or as a group without being limited to SQLite or any specific domain of applications. So the abstractions of *X-FTL* must introduce minimal changes to the standards such as SATA, and the changes must not disrupt existing applications. Third, SQLite and other upper layer applications should be able to use *X-FTL* services without considerable changes in their code. In particular, required changes, if any, must be limited to the use of extended abstractions provided by *X-FTL*.

This approach is novel in that it attempts to turn the weakness of flash memory (*i.e.*, being unable to update in place) into a strong point (*i.e.*, inherently atomic propagation of changes). Unlike the existing FTLs with support for atomic write [17, 18, 20], *X-FTL* supports atomicity of transactions without contradicting the steal policy of database buffer management at no redundant writes. This enables low-cost transactional support as well as minimal write amplification, which extends the life span of a flash storage device.

4.2 X-FTL Architecture and Abstractions

In the core of *X-FTL* is the *transactional logical-to-physical page mapping table* (or *X-L2P* in short) as shown in Figure 2. The *X-L2P* table is used in combination with a traditional page mapping table (or *L2P* in short) maintained by most FTLs. The *L2P* and *X-L2P* tables appear in the left and right sides of Figure 2, respectively. In order to provide transactional support at the storage level, we add to it more information such as the transaction id of an updater, the physical address of a page copied into a new location, and the status of the updater transaction. This will allow us to have full control over which pages can be reclaimed for garbage collection. Specifically, an old page invalidated by a transaction will not be garbage-collected as long as the updater transaction remain active, because the old page may have to be used to rollback the changes in case the transaction gets aborted. If the updater transaction commits successfully, the information on the old page can be released from the *X-L2P* table so that it can be reclaimed for garbage collection.

Obviously the additional information on transactions must be passed from transactions themselves to the flash-based storage, but it cannot be done through the standard storage interface such as SATA. We have extended the SATA interface so that the transaction id can be passed to the storage device by `read` and `write` commands. Besides, two new commands, `commit` and `abort`, have been added to the SATA interface so that the change in the status of a transaction can also be passed. The extensions we have made to the SATA interface are summarized below.

write(tid t, page p) The write command of SATA is augmented with an id of a transaction t that writes a logical page p. This command writes the content of p into a clean page in flash memory and add a new entry $(t, p, paddr, active)$ into the *X-L2P* table, where *paddr* is the physical address of the page the content of p is written into.

read(tid t, page p) The read command of SATA is also augmented with an id of a transaction t that reads a logical page p. This command reads the copy of p from the database snapshot of transaction t. Depending on whether t is the transaction that updated p most recently or not, a different version of p may be returned.

commit(tid t) This is a new command added to the SATA interface. When a commit command is given by a transaction t, the physical addresses of new content written by t become permanent and the old addresses are released so that the old page can be reclaimed for garbage collection.

abort(tid t) This is a new command added to the SATA interface. When an abort command is given by a transaction t, the physical addresses of new content written by t are abandoned and those pages can be reclaimed for garbage collection.

Note that the SATA command set is not always available for SQLite and other applications that access files through a file system. Instead of invoking the SATA commands directly from SQLite, we have extended the `ioctl` and `fsync` system calls so that the additional information about transactions can be passed to the storage device through the file system.

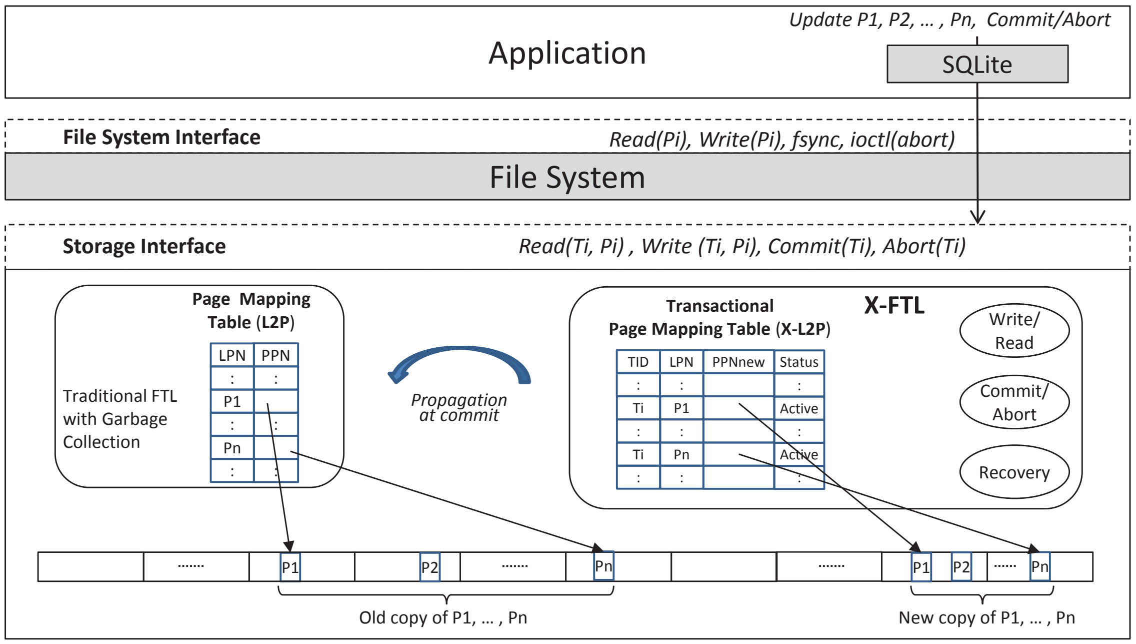


Figure 2: *X-FTL* Architecture: an FTL for Transactional Atomicity

4.3 X-FTL Advantages for SQLite

When SQLite runs in *rollback* mode, it creates a journal file for every transaction when it begins so its changes can be rolled back if necessary, and deletes the journal file when the transaction ends. The amount of overhead incurred by frequent file creations and deletions is non-trivial. Whenever a page is about to be updated by a transaction, the old content of the page is synchronously copied into a journal file. When a transaction commits, `fsync` system calls must be invoked for both the log and database files to guarantee the durability of the transaction. This makes SQLite execute the costly `fsync` system call excessively.

When SQLite runs in *write-ahead log* mode, for a page that is about to be updated by a transaction, the new content of the page is written to the log file. So, reading a certain page may require accessing both the log file and the database file to find the correct version of the page. This process can be facilitated by an in-memory index but reading the two files may still be unavoidable and the overhead is not trivial when it happens.

With *X-FTL* that supports atomic page updates at the storage device level, the runtime overhead of SQLite can be reduced dramatically. First, SQLite does not have to write a page (physically) more than once for each logical page write. Second, a single invocation of `fsync` call will be enough for each committing transaction because all the updates are made directly to the database file. Consequently, the I/O efficiency and the transaction throughput of SQLite can improve significantly for any workload with non-trivial update activities.

The current version (3.7.10) of SQLite supports the atomicity of a transaction that updates multiple database files but it is awkward or incomplete [4, 6]. When a transaction updates two or more database files in *rollback* mode, a master journal file, in addition to regular journal files, should be created to guarantee the atomic propagation of the entire set of updates made against the database files [4]. With *X-FTL*, in contrast, SQLite keeps trace of the multi-file updates in the *X-L2P* table under the same transaction id and supports the atomicity of the transaction without additional effort.

5. IMPLEMENTATIONS

This section presents the implementation details of the *X-FTL* architecture depicted in Section 4. The dominant portion of the *X-FTL* implementation is made into the firmware of the OpenSSD development board by cross-compiling the source code written in C. As an intermediary between *X-FTL* and SQLite, the file system accepts the information of active transactions through the `fsync` and `ioctl` system calls and passes it to *X-FTL* via the extended SATA command set. To carry this out, additional changes are also made in the system calls and the SATA interface. As will be described below, the changes made in SQLite is minimal.

We have chosen the `ext4` as the underlying file system, as it is the most common file system in Android platforms and any improvement in performance by *X-FTL* would have direct impact on many real-world mobile applications running on `ext4`.

5.1 Changes made in SQLite

The original SQLite runs in either *rollback* or *write-ahead log* mode to achieve atomic propagation of updated pages. However, if SQLite runs on *X-FTL*, it does not have to run in either mode. Instead, it can simply turn them off, because *X-FTL* supports transactional atomicity and durability at the storage level. When running in *off* mode, SQLite applies changes directly to database pages and the metadata of the database file. Recall that, even in *off* mode, the database buffer is still managed by the steal and force policies.

The force policy can be enforced by simply force-writing all the pages updated by a committing transaction to the database using `fsync` system call of an underlying file system. For an aborting transaction, however, the process of rolling back the changes made by the transaction is not defined [4] when SQLite runs in *off* mode. Consequently, an aborting transaction may leave a database in an inconsistent state. This is because the steal policy allows an uncommitted change to be propagated to the database and SQLite does not undo the uncommitted change in the *off* mode.

Unfortunately, there is no counterpart of the `fsync` system call that can undo changes already written to a file. The only way to address this problem is to pass the information of an aborting transaction to *X-FTL*, so that uncommitted changes can be rolled back inside a flash-based storage device. This is done by modifying SQLite such that it invokes a new system call for an aborting transaction, and this is the only change we have made to SQLite. The new system call is implemented in the file system by adding a new request type '*abort*' to the `ioctl` system call.

5.2 Changes made in File System

A file system plays a messenger role in passing the transactional context of database accesses from SQLite to *X-FTL*. For example, when a page is updated by a transaction, its id needs to be passed to *X-FTL* along with the content of the page so that *X-FTL* can keep track of the pages that are updated by the transaction. When the transaction requests a read or write operation, it is translated into `read(t,p)` or `write(t,p)`, where `t` and `p` are the id of the transaction and the logical address of a page, respectively. As will be described in Section 5.3, `read(t,p)` and `write(t,p)` are new commands added to the standard SATA command set. Note that transaction ids are managed by the file system instead of SQLite. This implementation decision is made because SQLite does not run as a stand-alone server but instead is linked to an application as a library, and thus it is difficult for applications to manage transaction ids globally.

Similarly, a commit or abort command is passed from a transaction to *X-FTL* via the file system. When a transaction t is about to commit, the file system is notified of that via an `fsync` system call invoked for the transaction so that all updated pages are force-written to the database. The `fsync` system call is translated to one or more `write(t,p)` commands for all updated pages. This is then followed by a `commit(t)` command passed to *X-FTL* via the extended SATA interface. We have added a new `commit(t)` command to the SATA command set by extending the parameter set of `trim` command.

When a transaction t is about to abort, the file system is notified of that via an `ioctl` system call given for the transaction. Since SQLite adopts the steal policy for buffer management, some of the pages updated by the transaction may have already been propagated to the database, while the rest may still be cached in the file system buffer. Undoing the cached changes is done simply by dropping them from the file system buffer. To rollback the changes written to the database, an `abort(t)` command is issued to *X-FTL*. Similarly to the `commit(t)` command, the `abort(t)` command is implemented by extending the parameter set of `trim` command.[2] The detailed implementation of `commit(t)` and `abort(t)` commands is described in the next section.

[2]An eMMC flash memory card, commonly used in smartphones, allows to add application-specific commands to the storage interface [5]. With an eMMC card, the commit and abort commands could be added without modifying the trim command.

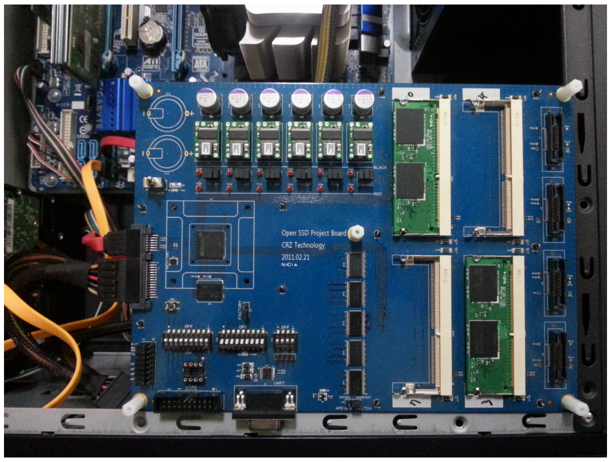


Figure 3: OpenSSD connected to host system

5.3 Changes made in FTL

In order to prototype *X-FTL*, we have implemented *X-FTL* on the OpenSSD platform. OpenSSD is an SSD development platform that is made publicly available by the OpenSSD Project to promote research and education on the recent flash-based solid state drive technology [3]. The OpenSSD platform is based on the *Barefoot* controller, which is an early commercial product by Indilinx. Thus it has the same performance characteristics as a commercial SSD equipped with the Barefoot controller. OpenSSD adopts a page mapping scheme for flash memory management like most contemporary SSD products and eMMC flash memory cards. The OpenSSD board is connected to a host system through the SATA interface as shown in Figure 3.

The *X-L2P* table is maintained persistently in the flash memory of the OpenSSD development board. The most recent changes are cached in DRAM until they are committed. As shown in Figure 2, the *X-L2P* table stores an entry (t, la_p, pa_a, s) for each page p updated by an active transaction t, where la_p is the logical address of p, pa_p is the (new) physical address of p, and s is the status of transaction t. The physical address pa_p is determined by the FTL when the new content of p is written to a free page in flash memory. The value of s can be *active*, *committed* or *aborted*. Note that a new entry for page p is added to the *X-L2P* table only when p is updated for the first time. If the same page p is updated again, the existing entry is just updated with a new physical address.

Each entry in the *X-L2P* table serves dual purposes. First, it keeps track of the latest version of a page while maintaining the older but committed copy intact so that a correct version of the page can be read by concurrent transactions. Specifically, when a `read(t,p)` command is given, if t matches the transaction id of an entry of p in the *X-L2P* table, then the copy of p whose physical address in the entry is returned. Otherwise, the older committed copy is found in the *L2P* table and returned to the reader. Second, the entry of p in the *X-L2P* table prevents the latest uncommitted copy of p from being garbage-collected. When a flash memory block is picked up for garbage collection, the pages in the block are considered valid and copied into a new block if their physical addresses are found in the *X-L2P* table. That is, a page is considered invalid only when it is not found in either the *L2P* table or the *X-L2P* table.

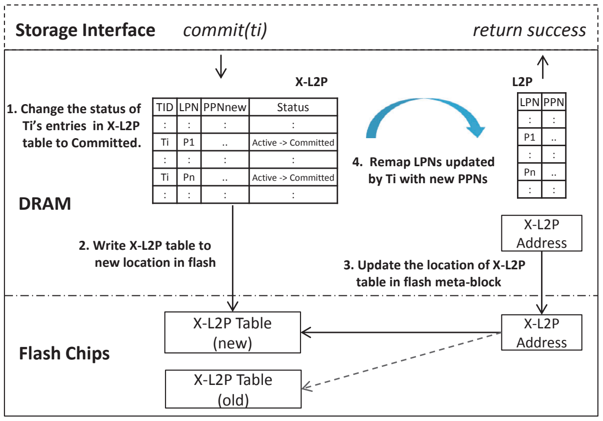

Storage Interface
commit(ti)
return success
X-L2P
L2P
1. Change the status of Ti's entries in X-L2P table to Committed.
TID LPN PPNnew Status
Ti P1 .. Active -> Committed
Ti Pn .. Active -> Committed
4. Remap LPNs updated by Ti with new PPNs
LPN PPN
P1 ..
Pn ..
DRAM
X-L2P Address
2. Write X-L2P table to new location in flash
3. Update the location of X-L2P table in flash meta-block
Flash Chips
X-L2P Table (new)
X-L2P Table (old)
X-L2P Address

Figure 4: Commit procedure in X-FTL

When a `commit(t)` command is received (via a `trim` command), all the pages updated by t must have been written to flash memory persistently and their physical addresses must have been saved in the *X-L2P* table. Thus, the only remaining steps are to change the status field of each entry from *active* to *committed*, and flush all the entries created by t persistently to flash memory. From this moment on, transaction t is considered committed, and the *X-L2P* table entries with a *committed* status become available for reuse by other transactions. Besides, the *L2P* table is updated so that its entries point to the committed physical copies of the pages updated by t. Note that the propagation of committed changes from the *X-L2P* table to the *L2P* table is idempotent and it is the original FTL that is responsible for backing the *L2P* table up persistently in flash memory and getting prepared for crash recovery.

The *X-L2P* table can be configured to have any number of entries, but it is very small in the current implementation – either 500 entries (8KB) or 1000 entries (16KB).[3] This is because the minimum required number of entries is a few tens due to the relatively small number of concurrent transactions running for SQLite databases and each *X-L2P* entry is only 16 bytes long. Thus, when a transaction commits, the entire *X-L2P* table is written to flash memory in copy-on-write fashion. The physical location of the *X-L2P* table is kept track of by a meta-data block in flash memory reserved by an FTL. We assume that updating the physical location in the meta-data block is done atomically. The commit procedure of *X-FTL* is summarized in Figure 4.

Processing an `abort(t)` command is much simpler and done in two steps. First, for all t's entries in the *X-L2P* table, the status fields are changed from *active* to *aborted*. Second, all the corresponding pages in flash memory are invalidated.

5.4 Recovery

For the recovery of a database, we consider two types of common failures. The first type of a failure occurs when SQLite is terminated abnormally but without failing the entire system. The other type occurs when the entire system fails, for example, by a power outage.

In the first case, since the file system and *X-FTL* are still functioning normally, the system kernel can detect the file opened by a killed process. Then, using the abort logic described in Sections 5.2 and 5.3, the file system drops all the dirty pages of the file from the file system cache, and *X-FTL* rolls back all the uncommitted updates that are already written to the database file.

In the second case, on a system reboot, both the *L2P* and *X-L2P* tables are loaded from the flash memory to determine the status of transactions killed by a system crash. All the *X-FTL* table entries with a *committed* status are reflected to the *L2P* table. This operation is idempotent and the only step required to ensure durability of all recently committed transactions. Propagation of committed changes (*i.e.*, updated pages) must have been done already to the database for all committed transactions by the commit logic described in Section 5.3. For an incomplete transaction at the time of a crash, all the uncommitted changes are undone by the same abort logic described above.

6. PERFORMANCE EVALUATION

In this section, we present the performance evaluation carried out to analyze the impact of *X-FTL* on SQLite as well as a file system. To understand the impact on SQLite, we picked three different database workloads. We then ran SQLite in *rollback* and *write-ahead log* modes on top of the (unchanged) `ext4` file system with the OpenSSD board running the original FTL. We also ran the modified SQLite on top of the `ext4` file system with the changed system calls with the OpenSSD board running *X-FTL*. We included a file system benchmark to evaluate the impact of *X-FTL* on the performance of the `ext4` file system.

6.1 Experimental Setup

The OpenSSD development platform is equipped with Samsung K9LCG08U1M flash memory chips. These flash memory chips are of MLC NAND type with 8KB pages and 128 pages per block. The OpenSSD has a Barefoot 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. The version of SATA interface is 2.0 that supports 3 Gbps.

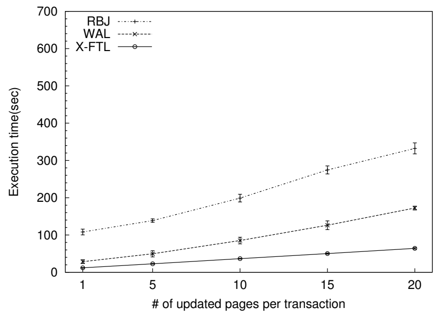
The host machine is a Linux system with 3.5.2 kernel running on Intel core i7-860 2.8GHz processor and 2GB DRAM. We used the `ext4` file system in *ordered* mode for metadata journaling when SQLite ran in *rollback* or *write-ahead log* mode. When SQLite ran on *X-FTL*, the file system journaling was turned off but the changes we added to the file system (described in Section 5.2) were enabled.

The version of SQLite used in this paper was 3.7.10, which supports both *rollback* and *write-ahead log* modes. The page size was set to 8KB to match the page size of the flash memory chips installed on the OpenSSD board.

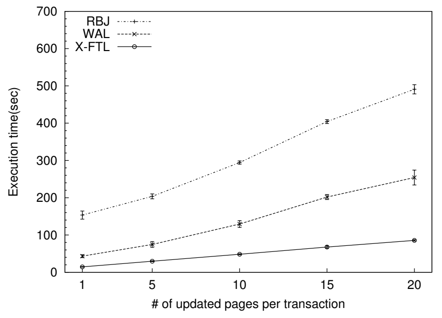
6.2 Workloads

We used three database workloads and a file system benchmark. The database workloads are a synthetic workload, a set of traces from and a popular benchmark for Android smartphones, and the TPC-C benchmark. The Flexible I/O (FIO) was used as a file system benchmark.

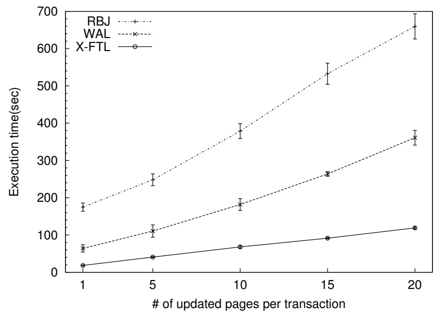
[3]Most eMMC flash memory cards are equipped with 512KB SRAM, 128KB of which is used for maintaining the L2P mapping table. We believe it is not so impractical to store an *X-L2P* table of 8KB $\sim$ 16KB in the SRAM of an eMMC card.



(a) GC validity 30%



(b) GC validity 50%

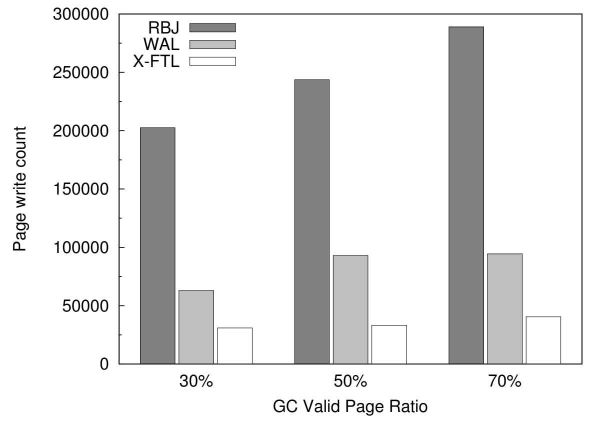


(c) GC validity 70%

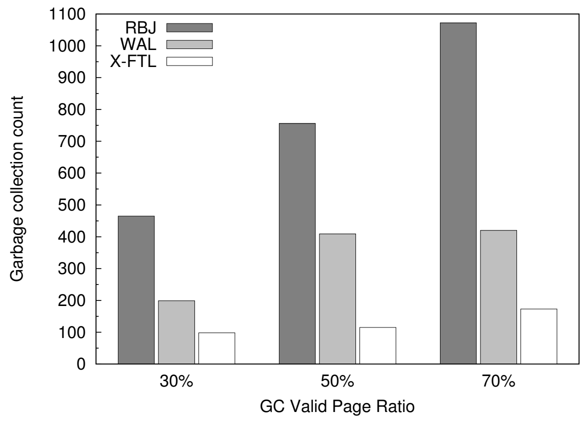
Figure 5: SQLite Performance (x1,000 Synthetic Transactions)

Mode	Host-side					FTL-side			
	SQLite DB	SQLite Journal	File System	Total Counts	fsync calls	Write	Read	GC	Erase
RBJ	6,230	7,222	15,987	29,439	2,999	243,639	9,792	756	2,044
WAL	3,523	5,754	3,646	12,923	1,013	92,979	3,472	409	897
X-FTL	5,211	0	994	6,205	994	33,239	2,011	115	243

Table 1: I/O Count (# of updated pages per transaction = 5, GC validity = 50%)



(a) Write count



(b) Garbage collection count

Figure 6: I/O Activities inside OpenSSD (# of updated pages per transaction = 5)

Synthetic This workload consists of a `partsupply` table created by the `dbgen` tool of the TPC-H benchmark. This table contains 60,000 tuples of 220 bytes each. Each transaction reads a fixed number of tuples using random `partkey` values, updates the `supplycost` field of each tuple, and commits.

Android Smartphone This workload consists of traces obtained by running four popular applications on an Android 4.1.2 Jelly Bean SDK, namely, RL Benchmark [2], Gmail, Facebook, and a web browser. RL Benchmark is a popular benchmark used for performance evaluation of SQLite on Android platforms. We modified the source code of SQLite to capture all the transactions and their SQL statements.

TPC-C The DBT2 tool was used for TPC-C benchmarking with 10 warehouses [1, 14]. Two separate workloads, mix and read-intensive, were created by adjusting the ratio of five transaction types. Since the locking granularity of SQLite is a database file, the number of database connections was set to one in order to avoid frequently aborting update transactions.

File System Benchmark The Flexible I/O (FIO) benchmark is commonly used to test the performance of file and storage systems [8]. It spawns a number of threads or processes doing a particular type of I/O operations as specified by the user parameters. This benchmark was added to evaluate the effects of *X-FTL* on the random write performance of a file system.

6.3 Run-Time Performance

This section demonstrates the effectiveness of *X-FTL* by comparing the performance of SQLite with and without *X-FTL*. Similarly, the performance of `ext4` was tested with and without *X-FTL* for file system benchmark. We use the `RBJ`, `WAL` and `X-FTL` symbols to denote the execution of SQLite

	RLBenchmark	Gmail	Facebook	WebBrowser
# of database files	1	2	11	6
# of tables	3	31	72	26
# of queries	82,234	15,533	4,924	7,929
# of select queries	5,200	3,540	1,687	1,954
# of join queries	0	1,381	28	1,351
# of insert queries	51,002	7,288	2,403	1,261
# of update queries	26,000	889	430	1,813
# of delete queries	2	2,357	117	1,373
Average updated pages per transaction	3.31	4.93	2.29	2.95
# of DDL/SQLite commands	30	78	259	177

Table 2: Analysis of Android Smartphone Traces

in *rollback* mode, *write-ahead log* mode and with *X-FTL* enabled, respectively. Each performance measurement presented in this section was an average of five runs or more. The standard deviation is also presented as an error bar whenever it was large enough to be visible in the graph.

6.3.1 Synthetic workload

In the synthetic workload, we varied the number of updates requested by a transaction from one to 20, and 1,000 transactions were executed for each fixed number of updates. To evaluate the effect of garbage collections by FTL, we controlled aging of the OpenSSD flash memory chips such that the ratio of valid pages carried over by garbage collection was approximately 30%, 50% or 70%.

Figure 5 shows the elapsed times of SQLite when it ran in *rollback* or *write-ahead log* mode on the original FTL of the OpenSSD and when it ran in *off* mode on *X-FTL* under various configurations. As is shown clearly in Figure 5(b), *X-FTL* helped SQLite process transactions much faster than *write-ahead log* and *rollback* modes by 3.5 and 11.7 times. The standard deviation of elapsed times was 6.04% on average.

The considerable gain in performance was direct reflection of reductions in the number of write operations and `fsync` system calls. Recall that, with the force policy, SQLite force-writes all the updated pages when a transaction commits. Table 1 compares *rollback* and *write-ahead log* modes with *X-FTL* with respect to the number of writes and `fsync` calls. In the case of *rollback*, in particular, both numbers were very high. This is because SQLite had to create and delete a journal file for each transaction and consequently had to use `fsync` call very frequently. In *write-ahead log* mode, SQLite wrote twice as many pages as running it with *X-FTL*, because it had to write pages to both log and database files.

Table 1 drills down the I/O activities further for the case when the number of pages updated per transaction was five. In the 'Host-side' columns, we counted the number of page writes requested by SQLite and the number of metadata page writes requested by the file system separately as well as the total number of `fsync` calls. In the 'FTL-side' columns, we counted the number of pages written and read (including those copied-back internally in the flash memory chips) as well as the frequencies of garbage collection and block erase operations. Garbage collection is always done for an individual flash memory block. The block erase count includes the data blocks garbage collected and the metadata blocks erased by FTL. Figure 6 visualizes two key measurements from the 'FTL-side' column in Table 1 in bar charts: (a) the number of pages written and (b) the frequencies of garbage collection. Figures 5 and 6 show that the trend in elapsed times among the three modes is consistent with the trend in the I/O activities.

In *rollback* mode, each transaction writes 5 data pages and one header page to the journal file. This incurs two `fsync` calls, one for data pages and the other for a head page. When it commits, the transaction writes 5 data pages and one header page to the database file, which is followed by a single `fsync` call.

Similarly, in *write-ahead log* mode, each transaction writes 5 data pages and one header page to the log file. Then the transaction invokes a single `fsync` call before it commits. The propagation of the changes to the database file, however, is not done until a checkpointing operation is triggered. Since SQLite carries out checkpointing every 1,000 page writes by default, a total of five checkpointing operations were done for 1,000 transactions. The number of page writes requested for the database file was less than 5,000, because pages updated more than once are flushed to the storage only once by a checkpointing operation.

When SQLite ran with *X-FTL*, the total number of page writes requested by SQLite and the file system was about a half of that for *write-ahead log* mode, because SQLite did not have to write anything to the log file.

6.3.2 Android Smartphone Workload

The four Android smartphone traces, whose characteristics are summarized in Table 2, were replayed with SQLite on the OpenSSD board. SQLite ran in *rollback* mode, *write-ahead log* mode or *off* mode with *X-FTL*. In Figure 7, we measured the elapsed time taken by SQLite to process each workload completely. Since the performance gap between the *rollback* and *write-ahead log* modes was similar to that

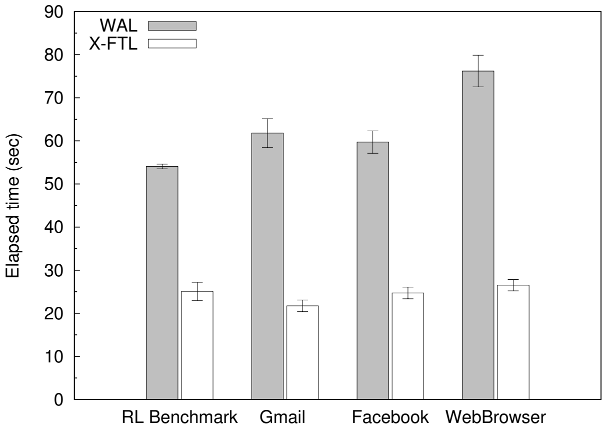


Figure 7: Smartphone Workload Performance

observed in the synthetic workload, we did not include the elapsed time of *rollback* mode for the clarity of presentation. Across all the four traces, SQLite performed 2.4 to 3.0 times faster when it ran with *X-FTL* than when it ran in *write-ahead log* mode. These results match the elapsed times and the trend of I/O activities observed in the synthetic workloads (shown in Figure 5(b) and Figure 6). The standard deviation of elapsed times was 5.02% on average.

To provide better insight into the observations, the characteristics of the traces are described below with respect to database query processing.

RL Benchmark is a write-intensive workload that consists of 13 different types of SQL statements performed on a single table with three attributes. The workload includes roughly 50,000 insertions, 5,000 selections, 25,000 updates, an index creation, and a table drop.

Gmail trace includes common operations such as saving new messages in the inbox, reading from and searching for keywords in the inbox. The Gmail application relies on SQLite to capture and store everything related to messages such as senders, receivers, label names and mail bodies in the *mailstore* database file. This trace includes a large number of insert statements. The read-write ratio was about 3 to 7 with more writes than reads.

Facebook trace was obtained from a Facebook application that reads news feed, sends messages and uploads photo files. A total of 11 files were created by the Facebook application, but `fb.db` was accessed most frequently by many SQL statements. Similarly to Gmail, this trace includes a large number of insert statements, because Facebook uses SQLite to store most of the information on the screen in a database. In addition, Facebook stores many small thumbnail images in the SQLite database as blobs. This makes the number of updates per transaction tend to be high. The read-write ratio was about 3 to 7 like the Gmail trace.

Browser trace was obtained while the Android web browser read online newspapers, surfed several portal sites and online shopping sites, and SNS sites. The web browser uses SQLite to manage the browsing history, bookmarks, the titles and thumbnails of fetched web pages. Since the URLs of all visited web pages are stored, the history table receives many update statements. In addition, cookie data are frequently inserted and deleted when web pages are accessed. Thus, the cookie table also received a large number of update statements. Among the six files the browser creates, `browser2.db` was the dominant target of most SQL statements as the main database file. Another interesting thing about this trace is that it includes quite a large number of join queries. The read-write ratio was about 4 to 6.

6.3.3 TPC-C Benchmark

By adjusting the relative frequencies of the five types of transactions, we created four separate workloads: write-intensive, read-intensive, selection-only, and join-only. Table 3 shows the configuration of each workload. The TPC-C benchmark results measured in transactions processed per minute (tpmC) are summarized in Table 4.

Transaction Types	Delivery	Order Status	Payment	Stock Level	New Order
Write-intensive	4%	4%	43%	4%	45%
Read-intensive	0%	50%	0%	45%	5%
Selection-only	0%	100%	0%	0%	0%
Join-only	0%	0%	0%	100%	0%

Table 3: TPC-C Workloads

(measured in tpmC)	Write intensive	Read intensive	Selection only	Join only
WAL	251	3,942	281,856	35,662
X-FTL	582	9,925	277,586	35,888

Table 4: TPC-C Performance

In the write-intensive workload, each transaction updates two pages on average. The performance gap between *X-FTL* and *write-ahead log* mode was not as wide as what was observed in the synthetic workload, which is purely update only, but the gain in the transaction throughput was still considerable. SQLite with *X-FTL* achieved about 130 percent higher transaction throughput than SQLite in *write-ahead log* mode.

In the read-intensive workload, the results were somewhat surprising. The performance gap was even higher in the read-intensive workload. With *X-FTL*, SQLite achieved about 152 percent improvement in transaction throughput than running in *write-ahead log* mode. This was due to the additional overhead incurred for SQLite to retrieve the most recent version of a page requested by each read. When it ran in *write-ahead log* mode, for each page to read, SQLite had to access the *write-ahead log* to find the most recent version. If it is not found in the *write-ahead log*, then it needs to be read from the database file. This is a well known problem common in the database systems that maintain multiple versions of a data page. This also clearly demonstrates the advantage of *X-FTL*, which allows SQLite to run in *off* mode without the burden of maintaining and accessing a log file.

In the selection-only or join-only workload, the performance of SQLite in *write-ahead log* mode and with *X-FTL* was comparable. This is obviously because, with no pages updated, SQLite did not have to read from or write into the log file. Since SQLite uses the nested-loop join algorithm, it does not create any temporary file for intermediate result.

6.3.4 File System Benchmark

As is described in Section 4, *X-FTL* supports atomic update propagation not only for SQLite but also for other applications such as a file system. We used the FIO benchmark (version 2.0.10) to evaluate the effects of *X-FTL* on the random write performance of a file system.

In this experiment, the average write IOPS was measured while a single thread updated data pages in a 4GB file randomly for 600 seconds. The size of a page was 8KB. An `fsync` system call was invoked every 1, 5, 10, 15, or 20 page writes to mimic the different sizes of transactions in the synthetic workload. In each test case, the `ext4` file system ran in *ordered* journaling mode (for metadata only), in *full* journaling mode (for both data and metadata), or in *off* mode with *X-FTL* enabled. Their throughput measured in IOPS is shown in Figure 8.

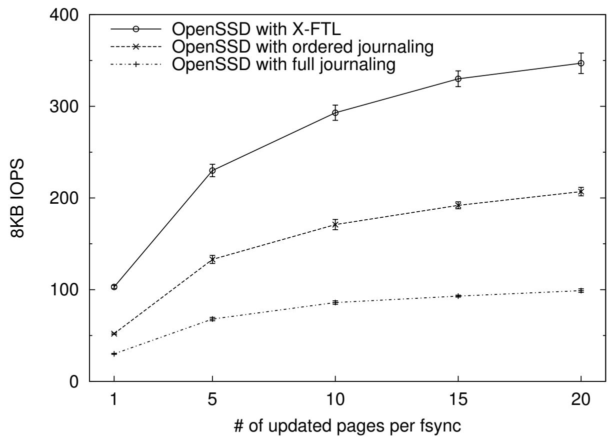


Figure 8: FIO Benchmark with a Single Thread

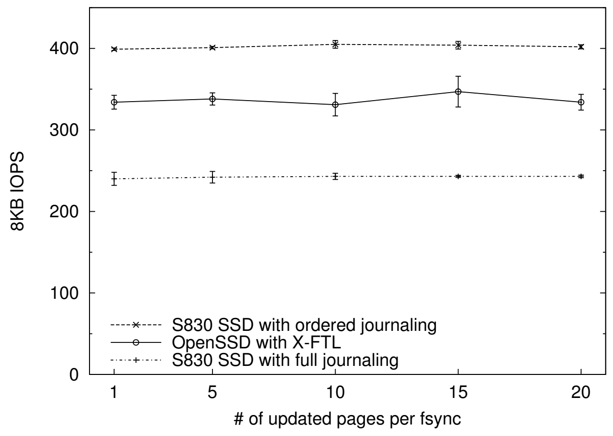


Figure 9: FIO Benchmark: *X-FTL* vs. S830 SSD (with 16 Concurrent Threads)

The IO throughput increased steadily in all the three cases as the `fsync` interval increased, obviously because less frequent `fsync` calls lowered the overhead of force-write operations. More importantly, the `ext4` file system with *X-FTL* achieved the highest IO throughput consistently across all `fsync` intervals by about 67 to 99% and 240 to 254% over the ordered and full journaling, respectively. The standard deviation was 2.34% on average. The full journaling guarantees the atomicity of page writes at the expense of writing each data page twice. The ordered journaling writes a data page only once but does not guarantee the atomicity of page writes. Besides, it requires two write barriers to guarantee the write order of data pages and metadata [22]. On a flash based storage device including OpenSSD, a write barrier command stores the mapping table as well as data pages persistently in flash memory chips. On the other hand, *X-FTL* invokes a commit command once as part of a `fsync` system call, which plays the same role as a write barrier. Therefore, the cost of an additional write of mapping table to flash memory contributed to the gap in IOPS between *X-FTL* and the *ordered* journaling.

To put the results from the OpenSSD development board in perspective, we repeated the same file system benchmark with an MLC based SSD, Samsung S830 (128GB), with 16 concurrent threads. Figure 9 shows the I/O throughput of S830 in ordered and full journaling modes as well as the I/O throughput of OpenSSD with *X-FTL*. In comparison with the results shown in Figure 8, the I/O throughput of OpenSSD was less than 25% and higher than 35% that of S830 in ordered and full journaling modes, respectively. This should not be surprising given that the controller of OpenSSD is at least one generation older than that of S830. The standard deviation was 2.00% on average in this experiment.

In summary, *X-FTL* achieves the level of consistency that can be achieved by a file system operating in full journaling mode at the cost even lower than that of ordered journaling mode.

6.4 Recovery Performance

In order to evaluate recovery performance, the OpenSSD board was turned off while SQLite was executing the synthetic workloads presented in Section 6.3.1. This test was repeated in *rollback*, *write-ahead log*, and *X-FTL* modes to examine the impact of the three execution modes on the recovery time of SQLite. We measured the time taken to restart the SQLite database in each mode.

mode	*Rollback*	*Write-ahead log*	*X-FTL*
(msec)	20.1	153.0	3.5

Table 5: SQLite Restart Time

Table 5 presents the average recovery times of three modes. For each mode, we took the average of restart times measured from five separate runs. The restart time given in Table 5 includes the time taken by SQLite to recover the database in each mode. Note that it does not include the common time taken by the OpenSSD to recover its FTL data structures (for example, the *L2P* table) and the time taken by the file system to remount the OpenSSD device. We observed the time taken by each step using a debugging tool attached to the OpenSSD.

In the case of *rollback* mode, the recovery time includes the time taken to carry out two steps. The first is to copy the old versions of pages updated by active transactions from the *rollback* journal file, if exists, to the original database file. The second is to delete the *rollback* file for undo purpose. The number of pages to be copied in this experiment was approximately ten.

In the case of *write-ahead log* mode, the recovery time includes the time taken to copy the latest copies of pages updated by committed transactions from a *write-ahead log* file to the original database file. Because the *write-ahead log* file is sized to one thousand pages by default, the recovery time in *write-ahead log* mode (153.0 milliseconds) was much longer than that in *rollback* mode (20.1 milliseconds).

Since only one SQLite database was updated in the experiment, the recovery times observed in *rollback* and *write-ahead log* modes represent the time to recover the database when accessed for the first time after the crash. If multiple databases were running at failure time, each database needs to be recovered individually.

In the case of *X-FTL* mode, the recovery steps include loading the *X-L2P* table and reflecting the *X-L2P* table entries with a *committed* status to the *L2P* table. All these steps took only about 3.5 milliseconds. The restart time of *X-FTL* was much shorter, because it did not involve copying any data page from the database or log file.

7. CONCLUSION

Considering the increasing popularity of SQLite in smartphone applications, improving the I/O efficiency of SQLite is a practical and critical problem that need be addressed immediately. This paper proposes a novel transactional FTL called *X-FTL*, which can efficiently support atomic propagation of multiple pages updated by transactional applications (*e.g.*, SQLite databases and file systems) to a flash memory storage device. The existing solutions such as running SQLite in *rollback* or *write-ahead log* mode and relying on the full journaling of `ext4` file system are essentially equivalent in that they are based on data journaling and managing the journal data is done by a host system.

X-FTL can relieve the host system of the burden of guaranteeing the transactional atomicity. *X-FTL* offloads the responsibility to the flash storage layer. By taking advantage of a copy-on-write mechanism adopted by existing FTLs, *X-FTL* implements all-or-nothing propagation of multiple pages without resorting to the costly journaling schemes. Therefore, *X-FTL* halves the amount of data to be written to the storage, and doubles the transactional performance and the life span of flash storage.

We have implemented *X-FTL* on an SSD development platform called OpenSSD by enhancing its FTL code with the *X-FTL* features. We have modified SQLite and *ext4* file system to take advantage of *X-FTL* with only minimal changes in their code. Using a comprehensive set of synthetic and real workloads for SQLite databases and file systems, we have demonstrated that *X-FTL* achieves a significant improvement in transaction performance.

8. ACKNOWLEDGMENTS

This research was supported in part by National Research Foundation of Korea(NRF) (2012R1A1A2A10044300, No.2010-0026511, No.2012-0006423), the IT R&D program of MKE/KEIT [10041244, SmartTV 2.0 Software Platform], and the IT/SW creative research program of MKE/NIPA (A-2012- H0505-12-1013). This work was also supported by Research Resettlement Fund for the new faculty of SNU.

9. REFERENCES

[1] OSDL Database Test 2(DBT-2). `http://osdldbt.sourceforge.net`.

[2] RL Benchmark:SQLite. `http://redlicense.com/`.

[3] OpenSSD Project. `http://goo.gl/JOTs5`, 2011.

[4] Atomic Commit In SQLite. `http://www.sqlite.org/atomicommit.html`, 2012.

[5] Embedded Multi-Media Card (eMMC), Electrical Standard (4.5.1). `http://www.jedec.org`, Jun 2012.

[6] Write-Ahead Logging. `http://www.sqlite.org/wal.html`, 2012.

[7] YAFFS: A NAND-Flash Filesystem. `http://www.yaffs.net/`, 2012.

[8] J. Axboe. FIO (Flexible IO Tester). `http://git.kernel.dk/?p=fio.git;a=summary`.

[9] A. Ban. Flash file system, Apr 1995. US Patent 5,404,485.

[10] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of USENIX conference on File and Storage Technologies (FAST '12)*, pages 101–116, 2012.

[11] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proceedings of USENIX conference on File and Storage Technologies (FAST'12)*, 2012.

[12] K. Lee and Y. Won. Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone. In *Proceedings of ACM EMSOFT*, pages 23–32, 2012.

[13] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation. *ACM Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[14] S. T. Leutenegger and D. Dias. A Modeling Study of the TPC-C Benchmark. In *Proceedings of ACM SIGMOD*, pages 22–31, 1993.

[15] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, Mar 1977.

[16] C. Mohan. Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging. In *Proceedings of ICDE*, pages 324–331, 1995.

[17] X. Ouyang, D. W. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of International Conference on High-Performance Computer Architecture (HPCA '11)*, pages 301–311, 2011.

[18] 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 (ISCE 2005)*, pages 155 – 160, Jun 2005.

[19] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of USENIX Annual Technical Conference*, pages 105–120, 2005.

[20] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, 2008.

[21] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb 1992.

[22] M. Steigerwald. Working with Write Barriers and Journaling File Systems. Imposing Order. *Linux Journal*, May 2007.

[23] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of USENIX Annual Technical Conference*, pages 1–14, 1996.

[24] S. C. Tweedie. Journaling the Linux ext2fs File System. In *Proceedings of Annual Linux Expo (LinuxExpo '98)*, 1998.

[25] D. Woodhouse. JFFS: The Journaling Flash File System. In *Proceedings of the Ottawa Linux Symposium*, 2001.