Supporting Transactional Atomicity in Flash Storage Devices

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

Flash memory does not allow data to be updated in place, and the copy-on-write strategy is adopted by most flash storage devices. The copy-on-write strategy in modern FTLs provides an excellent opportunity for offloading the burden of guaranteeing the transactional atomicity from a host system to flash storage and for supporting atomic update propagation. This paper presents X-FTL *as a model case of exploiting the opportunity in flash storage to achieve the transactional atomicity in a simple and efficient way.* 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.*

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 [3, 11].

Atomic propagation of one or more pages updated by a transaction can be implemented by shadow paging or physical logging. However, the cost will be considerable during normal processing. Although some commercial database systems adopt a solution based on physiological logging for I/O efficiency and and flexible locking granularity, others still rely on costlier but less sophisticated mechanisms based on redundant writing to limit the scale of its code base. For example, InnoDB uses a double-write-buffer strategy, and SQLite runs with rollback or write-ahead journaling for transactional atomic updates.

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

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

by leaving the existing page intact and writing the new content into a clean page at another location [5, 9]. 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 [10]. This provides an excellent opportunity for supporting atomic update propagation almost for free.

This paper presents *X-FTL* [6] as a model case of exploiting the opportunity in flash storage to achieve the transactional atomicity in a simple and efficient way. We describe the X-FTL approach and its extended abstractions, and sketch how *X-FTL* can be implemented in an SSD using a development board called OpenSSD. As use cases of *X-FTL*, we explain how SQLite and `ext4` file system can be minimally modified so as to make them compatible with *X-FTL*, and demonstrate the effectiveness of *X-FTL* using real and synthetic SQLite workloads for smartphone applications.

2 Transactional Support in SQLite

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 provided by the SQL interface.

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 [7, 8]. According to a recent survey [8], 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.

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.

In order to support the atomicity of transaction execution without the atomicity of a sector write, SQLite operates usually in either *rollback* mode [3] or *write-ahead log* mode [4]. 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.

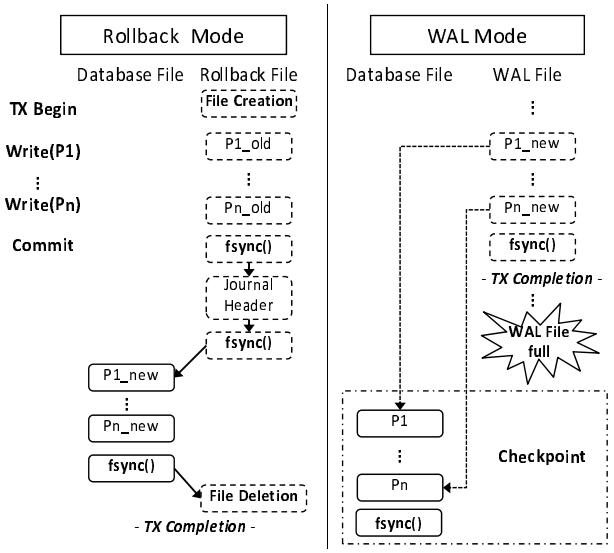


Figure 1: SQLite Journal Modes

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.

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 X-FTL for Transactional Atomicity

This section provides a brief overview of the design principles and the abstractions of *X-FTL*. *X-FTL* 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. Also, we present the performance evaluation carried out to analyze the impact of *X-FTL* on SQLite.

3.1 Design Principles

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 [5], 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 [12, 13, 14], *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.

3.2 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.

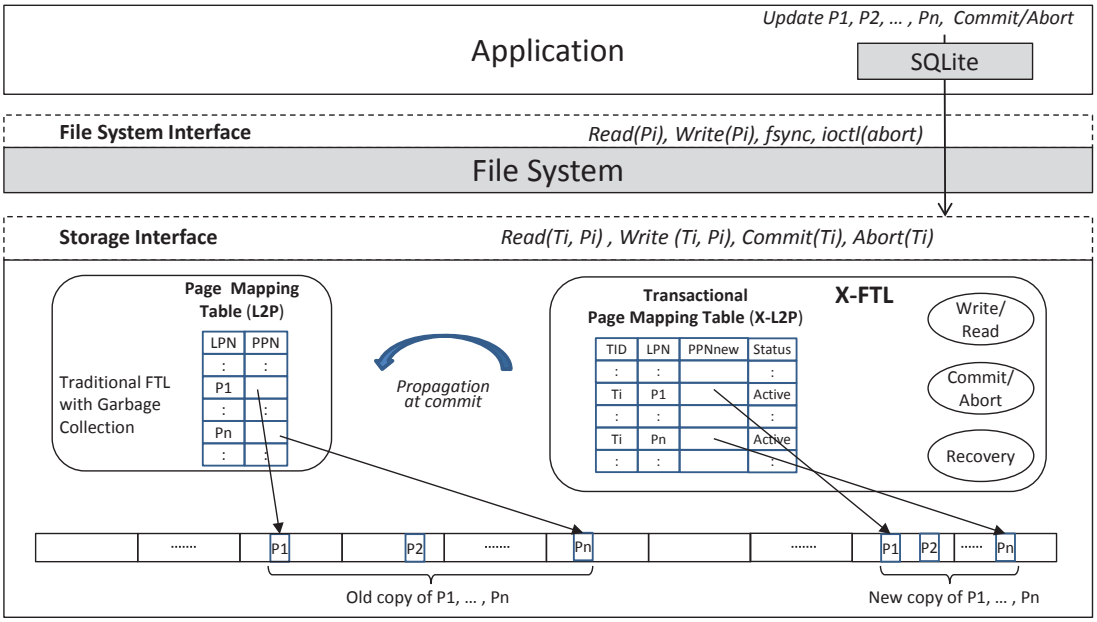


Figure 2: *X-FTL* Architecture: an FTL for Transactional Atomicity

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.

3.3 X-FTL Advantages for SQLite

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 [3, 4]. 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 [3]. 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.

3.4 Performance Evaluation

In order to understand the impact of *X-FTL* on SQLite, 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*.

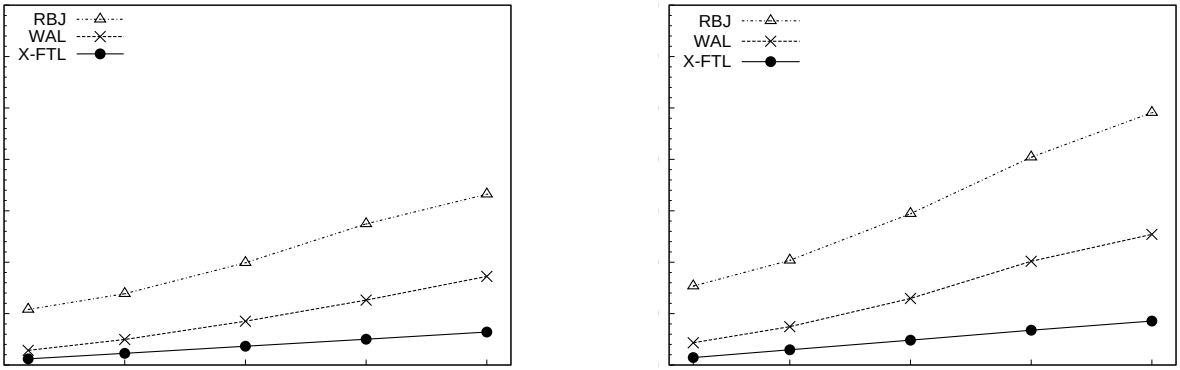
Using two different database workloads, we ran SQLite in *rollback* and *write-ahead log* modes on top of the (unchanged) `ext4` file system with the OpenSSD board running the original FTL. We also ran the modified SQLite on top of the `ext4` file system with the changed system calls with the OpenSSD board running *X-FTL*. The workloads are a synthetic workload, a set of traces from four popular Android benchmarks.

3.4.1 Experimental Setup

The OpenSSD development platform [2] 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 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 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.

3.4.2 Run-Time Performance

This section demonstrates the effectiveness of *X-FTL* by comparing the performance of SQLite with and without *X-FTL*. We use the `RBJ`, `WAL` and `X-FTL` symbols to denote the execution of SQLite 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.



(a) GC validity 30% (b) GC validity 50%

Figure 3: SQLite Performance (x1,000 Synthetic Transactions)

Mode	Host-side					FTL-side			
	SQLite		File System	Total Counts	fsync calls	Write	Read	GC	Erase
	DB	Journal							
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 4: I/O Count (# of updated pages per transaction = 5, GC validity = 50%)

3.4.2.1 Synthetic workload The synthetic 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. 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% or 50%. Garbage collection is always done for an individual flash memory block. When a flash memory block is picked up for garbage collection, the pages marked as valid in the block are copied into a new block, while invalid ones are simply discarded. For instance, if the ratio of valid pages is 50%, 64 out of 128 pages will be copied from the victim block to a new block, and the new block will contain just 64 free pages. This implies the garbage collection will leave just half of the block available for use after all the cost of erasing a block and copying 64 pages.

Figure 3 shows the elapsed times of SQLite when it ran in rollback or write-ahead log mode and when it ran in *off* mode with *X-FTL*. In Figure 3(a), under GC validity 30%, *X-FTL* helped SQLite process transactions much faster than *write-ahead log* and *rollback* modes by 2.7 and 9.7 times, respectively. In Figure 3(b), under GC validity 50%, the improvement ratios were even higher, 3.2 and 11.7 times, respectively. 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 4 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 4 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 (GC) and block erase operations. The write and block erase counts in Table 4 include the data pages and blocks garbage collected and the metadata blocks erased by FTL.

3.4.2.2 Android Smartphone Workload Android Smartphone workload consists of traces obtained by running four popular applications on an Android 4.1.2 Jelly Bean SDK, namely, RL Benchmark [1], 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.

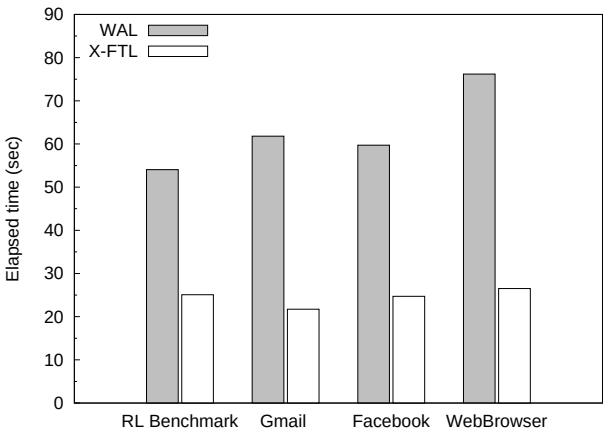


Figure 4: Smartphone Workload Performance

In Figure 4, 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 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 3(b)).

4 Concluding Remarks

X-FTL is a novel transactional FTL which can efficiently support atomic propagation of one or more pages updated by transactional applications (*e.g.*, SQLite databases and NoSQL key-value stores) to a flash memory storage device. The main contributions of the *X-FTL* scheme are: (1) it realizes low-cost atomic update for individual pages, (2) it provides the awareness of database semantics in page updates to support the atomicity of transactions, and (3) it exposes an extended abstraction (and APIs) to upper layer applications such as SQLite.

There are many non-database applications that require the semantics of transactional atomicity semantics (*i.e.*, the atomic propagation of a group of data pages) without a sophisticated recovery mechanism. *X-FTL* provides an effective means for transactional support as well as a simple storage abstraction with extended functionality. *X-FTL* also demonstrates that advanced storage devices could offload essential functions from the host system to the devices and help simplify the software stack of the host system.

References

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

[2] OpenSSD Project. http://www.openssd-project.org/, 2011.

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

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

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

[6] 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*, SIGMOD '13, pages 97–108, New York, NY, USA, 2013. ACM.

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

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.