MV-FTL: An FTL That Provides Page-Level Multi-Version Management

Doogie Lee, Mincheol Shin, Wongi Choi, Hongchan Roh, and Sanghyun Park, *Member, IEEE*

Abstract—In this paper, we propose MV-FTL, a multi-version flash transition layer (FTL) that provides page-level multi-version management. By extending a unique characteristic of solid-state drives (SSDs), the out-of-place (OoP) update to multi-version management, MV-FTL can both guarantee atomic page updates from each transaction and provide concurrency without requiring redundant log data writes as well. For evaluation, we first modified SQLite, a lightweight database management system (DBMS), to cooperate with MV-FTL. Owing to the architectural simplicity of SQLite, we clearly show that MV-FTL improves both the performance and the concurrency aspects of the system. In addition, to prove the effectiveness in a full-fledged enterprise-level DBMS, we modified MyRocks, a MySQL variant by Facebook, to use our new Patch Compaction algorithm, which deeply relies on MV-FTL. The TPC-C and LinkBench benchmark tests demonstrated that MV-FTL reduces the overall amount of writes, implying that MV-FTL can be effective in such DBMSs.

Index Terms—MVCC, FTL, flash translation layer, SSD, concurrency control

1 Introduction

Solid-state drives (SSDs) are being increasingly adopted in the DBMS research literature [1], [2], [3], [4], [5], [6]. In particular, there exists a unique and interesting characteristic of SSDs called the out-of-place (OoP) update. Because of the inability to overwrite in NAND-flash-memory, SSDs' main storage media, SSDs process updates in an append-only manner [7]; instead of overwriting existing data, SSDs redirect the updated data to an empty page and move the logical-to-physical (L2P) mappings onto them. When there is no free space to append new pages, an SSD triggers a garbage-collection mechanism that reclaims the space occupied by old and unnecessary page versions.

The way an SSD handles an update is quite familiar in the DBMS literature—in practice, the approach can easily be found on many DBMSs that implement multi-version concurrency control (MVCC)[8], [9], [10]. In MVCC, a DBMS handles updates in an append-only manner; that is, instead of replacing the old version with the new one, DBMS simply appends the new one, keeping old ones intact. Consequently, a DBMS stacks multiple versions of each record and serves concurrent accesses to a record in more flexible ways by using those multiple versions. Moreover, a DBMS also has garbage-collection mechanisms to reclaim the space occupied by expired versions.

However, being unaware of the SSD, a DBMS implements MVCC in its own way, without utilizing the OoP update characteristic of SSDs. In other words, a DBMS implements the append-only updates by itself, without knowing an SSD already handles updates in an append-only manner. Moreover, a DBMS executes garbage-collection operations to secure free space, which may overlap with SSD garbage collection.

From this observation, we propose MV-FTL, a multi-version flash translation layer (FTL), which is the core software layer of an SSD. The proposed solution explicitly manages multiple page versions and serves the multiple versions to a DBMS. More specifically, MV-FTL gathers the updated pages from each transaction into a new data structure called *diffL2P*, a logical data structure that represents a *version*. MV-FTL serves these diffL2Ps to transactions in an atomic fashion; that is, a transaction can see either all the pages in a diffL2P, or none of them. In addition to this atomic update, MV-FTL manages the commit orders between diffL2Ps, enabling the snapshot isolation [11] between updates from transactions.

Consequently, MV-FTL relieves the burdens of a DBMS to implement MVCC.[1] By cleverly exploiting the OoP update characteristic, MV-FTL provides the host the ability to manage multiple page versions without explicitly writing any additional log data. In addition, MV-FTL garbage-collects unnecessary page versions gracefully by extending the SSD's garbage-collection scheme, which will significantly reduce the MVCC garbage-collection overhead.

- *D. Lee is with Department of Computer Science, Yonsei University, Seoul 03722, Korea. E-mail: edoogie@gmail.com.*
- *M. Shin, W. Choi, and S. Park are with the Department of Computer Science, Yonsei University, Seoul 03722, Korea. E-mail: {smanioso, cwk1412, sanghyun}@yonsei.ac.kr.*
- *H. Roh is with SK Telecom, Seoul 04539, Korea. E-mail: fallsmal@gmail.com.*

Manuscript received 5 Sept. 2016; revised 4 Aug. 2017; accepted 14 Sept. 2017. Date of publication 27 Sept. 2017; date of current version 5 Dec. 2017. (Corresponding author: Hongchan Roh and Sanghyun Park.) Recommended for acceptance by C. Ordonez. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2017.2757016

1. MV-FTL will not fully replace MVCC, because MV-FTL can manage versions only in NAND page units, which is too coarse-grained to be used in enterprise-level DBMSs, where versions are managed in tuple units. Nevertheless, MV-FTL can still replace the MVCC of MyRocks or MongoRocks, which will be discussed later.

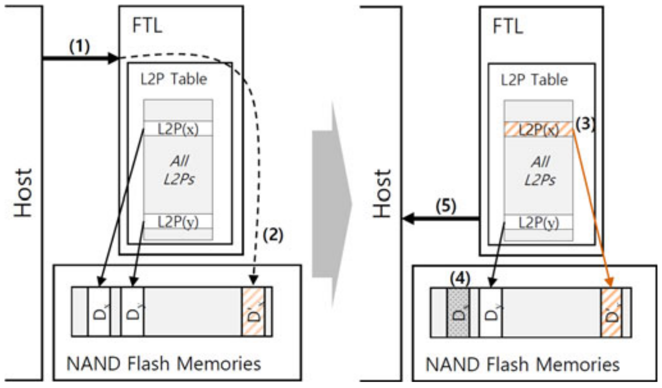


Fig. 1. Process of a write operation in FTL. L2P and flash page with diagonal stripes indicates a newly updated page, and shaded flash pages indicate the invalidated ones. When a write operation with LBA x and data D'_x is performed (1), the FTL writes new data into a free page in the NAND flash memories (2). The FTL then updates an L2P mapping for x (3) and invalidates the old flash page (4). Finally, the FTL issues an I/O completion message (5).

To show the performance and concurrency implications of MV-FTL, we first modified SQLite [12], one of the most popular DBMSs for mobile applications, and conducted several experiments using well-known benchmarks. In addition, to demonstrate the usefulness of MV-FTL in more sophisticated DBMSs, we modified MyRocks [13], a MySQL variant that is widely adopted by Facebook. Then, by running several benchmark tests on an MV-FTL emulator, we confirmed the potential benefits of MV-FTL on MyRocks.

In summary, the contributions of this paper are as follows:

- We propose a novel MV-FTL that provides page-level multi-version management to the host. MV-FTL guarantees the atomicity of transaction updates and provides snapshot isolation on SSDs in NAND page units. Because these come from the intrinsic OoP update characteristic, MV-FTL requires no additional log or data writes from the host.
- To minimize resource usage for multi-version management, we devised an efficient purge algorithm in MV-FTL. The algorithm purges unnecessary diffL2Ps without affecting other transactions.
- We implemented MV-FTL on *OpenSSD*, modified SQLite to cooperate with MV-FTL, and ran RL Benchmark and TPC-C on top of them. Because SQLite has a *lightweight* storage manager, the evaluations clearly showed the *real* effectiveness of MV-FTL in both the performance and concurrency aspects. MV-FTL improves SQLite's performance by up to 2.8x and significantly reduces the number of write operations. Furthermore, MV-FTL can handle multiple concurrent tasks gracefully, maintaining the transaction-processing performance even with conflicting read accesses.
- To prove the usefulness of MV-FTL in more sophisticated DBMSs, we modified MyRocks, implementing our novel *Patch Compaction* algorithm, which is based on MV-FTL. By running TPC-C and LinkBench, we proved that using MV-FTL and Patch Compaction could reduce the total number of bytes written to an SSD by up to 40%.

Meanwhile, many researchers have already proposed using the OoP characteristic for processing DBMS transactions [14], [15], [16]. In practice, those have successfully improved transaction-processing performance by exploiting the OoP update to guarantee atomic updates. However, from a concurrency point of view, because they cannot distinguish orders between committed updates, they need additional support from the DBMS to distinguish committed updates from each transaction, or they may suffer from concurrency anomalies such as *non-repeatable reads* or *phantom reads* [11].

The rest of the paper is organized as follows: First, in Section 2, we briefly review the necessary background knowledge, including FTL and MVCC. In Section 3, we present the basic idea behind MV-FTL, including the purging of unnecessary versions, and its durability. In Section 4, we describe the implementation of system modules, i.e., MV-FTL, SQLite, and other intermediate layers, and the evaluation using RL Benchmark [17] and TPC-C [18]. In Section 5, we present the TPC-C and LinkBench [19] evaluation results on RocksDB in an emulated MV-FTL environment. In Section 6, we introduce some previous studies related to our work. Finally, in Section 7, we conclude our paper and present ideas for future research.

2 Backgrounds

2.1 FTL

NAND flash memory is practically a very different media compared with the traditional block devices such as hard disk drives (HDDs), because it has unique characteristics—i.e., asymmetry in the units of read/program and erase operations, limited endurance, and so on. In particular, once written (programmed), a page can only be overwritten after being erased, whereas read and program operations are performed in NAND page units of 4 or 8 Kbytes each. The erase operations are carried out in NAND block units, with each NAND block comprising 64 or 128 pages. This mismatch makes overwrite operations very costly in NAND flash memories, i.e., to overwrite a single page, all the pages in a block must be erased and reprogrammed. In addition, the number of erase operations that a page can endure is limited to usually a few thousand, and therefore, an erase operation should be postponed as much as possible.

To overcome the problems above, most flash storages have adopted a sophisticated software layer called flash translation layer, that manages NAND flash memories [7], [20].

The FTL handles page write operations in an append-only manner. Fig. 1 illustrates the way in which FTL processes a page write operation. When a host overwrites data D'_x in a logical page, whose logical block address is x (i.e., step 1), D'_x is written to an empty page (step 2). The FTL then updates the logical-to-physical mapping, $L2P(x) \Rightarrow D'_x$ in the L2P table (step 3) and invalidates an old flash page (step 4). Finally, the FTL sends a complete message to the host (step 5).

As the FTL continues appending, the number of empty pages decreases to the point where there might be no more empty pages to which to write. In such a case, as illustrated in Fig. 2, the FTL triggers a garbage collection, which operates as follows:

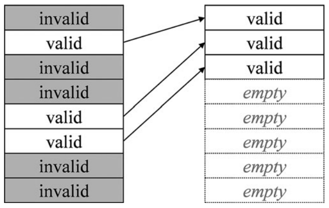


Fig. 2. Process of garbage collection in an FTL in which valid pages are moved to a separate block, freeing up a contiguous block of empty pages.

1) FTL first picks a block to be garbage-collected, with the selection being either arbitrary or via some predefined criteria;
2) FTL finds every valid page, i.e., pages with a valid L2P mapping in the L2P table, and moves the page to a new block, modifying the L2P mapping accordingly.

After garbage-collecting a block, the FTL increments the number of empty pages by the number of invalid pages in the block, where an invalid page means a page that has no L2P mapping in the L2P table.

2.2 FTL-Assisted Transaction Support

Many researchers have sought to improve the performance of data processing systems by implementing atomicity with OoP update characteristic of SSDs.

Prabhakaran et al. [14] proposed *TxFlash*. In *TxFlash*, the host sends the list of pages to update by a transaction to *TxFlash* before it starts updating. Then, *TxFlash* adds next-link to each page from the transaction, so that next-links of the pages form a cycle; that is, from the first to the penultimate pages, the next-link points to the next page (i.e., N + 1 for the N-th page) while the link on the last page points the first page. TxFlash keeps the updated L2P mappings of ongoing transactions separately from the main L2P table until the last page of the transaction is written to the NAND. In addition, by following these next-links, *TxFlash* can distinguish succeeded updates from failed ones after being powered off and on.

Ouyang et al. [15] proposed a new write command interface, *Write-Atomic()*. The *Write-Atomic()* command is used to deliver multiple discontinuous pages with a single command transfer. Based on this command, they proposed to transfer the page to be updated by a transaction *all-at-once*. Then, upon receiving the *Write-Atomic()* command, the SSD unpacks the pages and writes them onto a series of free pages. While writing, the SSD marks the flag bit as “0” for all the pages except the last page and “1” for the last page, in order to tell if all the pages from the transaction are successfully written. Similarly in *TxFlash*, the SSD keeps the updated L2P mappings separately until the last page write operation ends, applying the mapping changes just after the last page is firmly written. In addition, after power off, the SSD can find out if a series of updated pages are all done, simply by checking the flag bits; by scanning backward, the SSD simply discards the pages with flag “0” before the occurrence of a page with flag bit “1” is found.

Kang et al. [16] proposed *X-FTL*. In *X-FTL*, the host tags each page update from a transaction with the ID of the transaction. Accordingly, *X-FTL* keeps the L2P mappings from the transactions separately in the X-L2P table. After a while, when a transaction is committed, the host notifies the ID of the committed transaction to the X-FTL, and finally, the X-FTL finds all the L2P mappings that belong to the transaction and applies them to the main L2P table. In this way, X-FTL assures the atomic propagation of transaction updates.

All these approaches showed impressive improvements in transaction-processing performances, more or less depending on their target systems, workloads, and configurations. However, in the perspective of isolation, the three approaches described above have a common deficit. More specifically, although they are free from *dirty reads* as they can distinguish uncommitted pages from committed ones, they may suffer from *non-repeatable reads* or *phantom reads*. Berenson et al. [11] For this reason, they need additional help from DBMSs such as timestamp, multi-version management, and so on.

2.3 MVCC

To prevent conflicts between concurrent accesses, which could result in concurrency anomalies [11], DBMSs have traditionally used lock-based protocols [21]. By properly using shared and exclusive locks, a DBMS can avoid the conflicts in advance. However, at the same time, the locking protocol has too many disadvantages to be used in DBMSs; i.e., it requires very heavy overheads for detecting and resolving deadlocks; it is not scalable for the multi-core environment; it needs extra HW supports, and so on.

Meanwhile, MVCC takes an entirely different approach; instead of reading and writing the same records, each transaction on MVCC uses a virtual copy of each record on its snapshot of the database, which is usually taken when the transaction starts [11]. Accordingly, when other transactions concurrently attempt to read the given record from the database, the DBMS can service those requests by using the old committed versions of the same record, so that a read-only transaction does not have to wait for any other transaction. For this reason, most DBMSs implement MVCC to support more concurrency.

However, MVCC also adds non-trivial overhead to the storage systems. As MVCC requires more space for storing multiple versions, the DBMS can suffer from space overhead. To ensure that the database does not grow too much, an MVCC DBMS invokes garbage collection operations, purging *dead*—i.e., old and unnecessary—versions periodically, on demand, or both.

These garbage collection operations cause additional updates on underlying storage devices, which, in the worst case, could double the total number of pages written: once at update time and again at purging time. In addition, the DBMS must manage additional information to distinguish each version, worsening the space overhead of MVCC.

3 MV-FTL

3.1 Basic Idea

It is notable that the update schemes in MVCC and FTL have many things in common, that is, both handle updates in an append-only manner and have garbage-collection mechanisms. The main difference is that only one version is valid at a time in generic FTLs while multiple valid versions can simultaneously exist in MVCC DBMSs. In other words,

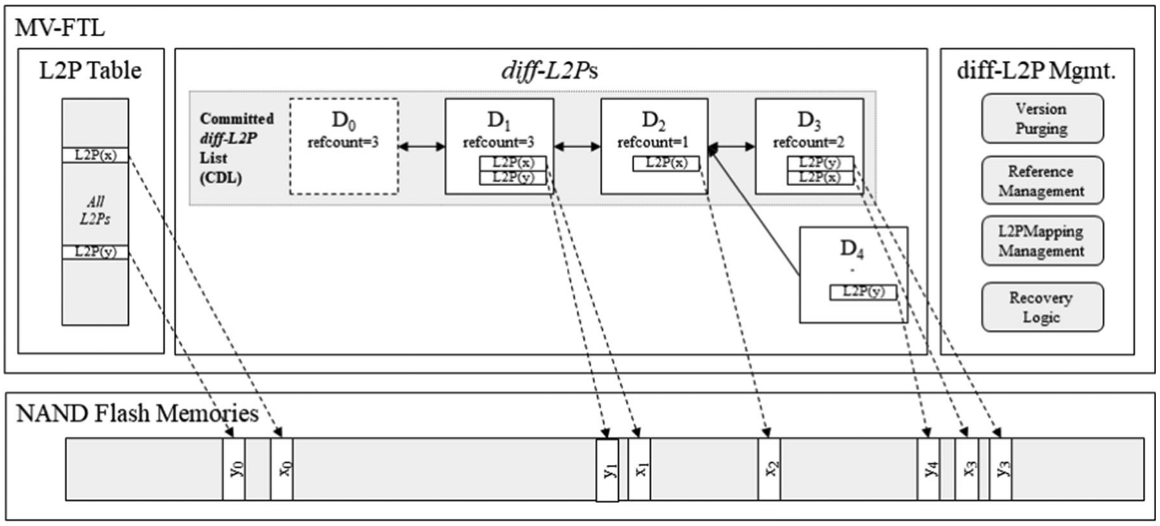


Fig. 3. Overall structure of our MV-FTL architecture.

FTL does invalidate the old version just after the host writes a new page version, whereas MVCC does not; MVCC allows multiple old but valid versions as long as transactions are accessing the specific ones.

Based on these observations, we extended an FTL to manage multiple versions, namely MV-FTL. MV-FTL allows each logical page to have multiple physical instances. In this way, MV-FTL can provide MVCC-like multi-version management. This version management would be more efficient, because it does not require any additional log writes.

MV-FTL is not intended to replace the MVCC implementation in enterprise-level DBMSs like [8], [9], [10]; rather, MV-FTL helps improve the MVCC of such DBMSs by providing efficient page-level multi-version management. This is mainly because MV-FTL manages multiple versions in too coarse-grained units. While MVCC DBMSs usually manage versions in variable-sized record units, MV-FTL can only manage versions in fixed-sized NAND page units, 4 or 8 Kbytes each. Of course, for some embedded DBMSs such as SQLite [12] and RocksDB [22], which manage versions in more coarse-grained units—pages in SQLite and files in RocksDB, MV-FTL can directly improve their version management performance.

3.2 MV-FTL Architecture and Design

Fig. 3 shows the overall architecture of MV-FTL. The core of an MV-FTL consists of a main L2P table, diffL2Ps, and Committed diffL2Ps List (CDL).

Each diffL2P corresponds to a version. When a transaction starts updating, it requests the MV-FTL to create a new diffL2P. Upon request, the MV-FTL creates a diffL2P and notifies the ID of the diffL2P back to the transaction. From then on, the transaction writes every page with the ID. MV-FTL then writes the page into a free NAND page. However, at this stage, the MV-FTL does not directly update the main L2P table as generic FTLs do; instead, the MV-FTL registers the L2P mapping into the diffL2P, keeping the L2P table and other diffL2Ps untouched. In this way, the MV-FTL has multiple L2P mapping versions on diffL2Ps. Algorithm 1 shows how the MV-FTL handles writes from the host.

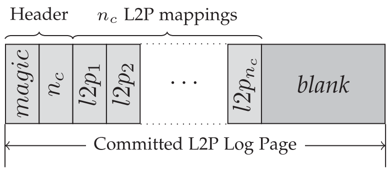
MV-FTL basically keeps every diffL2P and all its L2P mappings only in the volatile main memory inside SSD, thereby preventing uncommitted changes from being propagated to the storage even if any system failure occurs. MV-FTL only stores the L2P mappings on a diffL2P if the corresponding transaction finishes updating and is committed, such that the committed changes from the transaction become durable. (See the following Section 3.3)

In addition, we designed CDL into MV-FTL to manage the chronological order between committed diffL2Ps. After storing the L2P mappings of a committed diffL2P, MV-FTL appends the diffL2P to the tail of the CDL.

One important property of CDL is that CDL can never be empty; logically, there must be at least more than one version. To keep this property, MV-FTL initially puts an imaginary diffL2P node D_0, which virtually represents the original L2P table and has no additional L2P mapping in itself. D_0 shall be referenced by transactions as long as there has been no committed diffL2P yet. Like other diffL2Ps, MV-FTL will purge D_0 as soon as a new diffL2P has been committed and D_0's refcount becomes 0. Purging D_0 is almost free, because D_0 contains no L2P mappings in itself.

Algorithm 1. WritePage(did, logicalAddr, data)

```
Input: logicalAddr : logical Address of data to write
did : ID of diffL2P referred by the current transaction
data : data to write
Output: void
physicalAddr ← getFreePage();
nandWrite(physicalAddr, PAGE_SIZE, data);
foreach L2PEntry e in diffL2P[did].entries do
    if e.logicalAddr == logicalAddr then
        invalidate(e.physicalAddr);
        e.physicalAddr ← physicalAddr;
        return;
    end
end
entry ← AllocateL2PEntry();
entry.logicalAddr ← logicalAddr;
entry.physicalAddr ← physicalAddr;
diffL2P[did].entries.append(entry);
```



(a)

Name	Offset(B)	Size(B)	Description
$magic$	0	4	header magic number
n_c	4	4	# of L2P mappings, or 0 if the page is *overflowed*
$l2p_{n_c}$	8	8 * n_c	n_c L2P mappings(8B each)

(b)

Fig. 4. (a) Structure of a committed L2P log page and (b) its format.

3.3 Atomicity Supports

In DBMS literature, a transaction can either see all updates from another specific transaction or cannot see any. To guarantee this property, we established principles that regulate the visible diffL2P range of a transaction. The principles are as follows:

- Before a read-write transaction commits, only the transaction can see the corresponding diffL2P.
- When a read-write transaction commits, the MV-FTL stores all the L2P mappings belonging to the diffL2P to the NAND.
- After a transaction commits, the other transactions that started later than the commit operation can see the corresponding diffL2P. (See the next section for more details.)
- When a transaction is aborted, all the L2P mappings belonging to the *diffL2P* are simply discarded, marking all the written pages as invalid.

By obeying these principles, a transaction can see the updates from other transactions in an atomic manner.

When a transaction commits, the MV-FTL stores all the L2P mappings on the diffL2P. The MV-FTL first gathers the L2P mappings into the committed L2P log pages (shortly commitLog pages, hereafter) shown in Fig. 4. MV-FTL writes the commitLog pages into dedicated meta blocks called *committed L2P log block*s (shortly commitLog blocks, hereafter) in ascending order, from the first to the last page. After completely writing all the commitLog pages, MV-FTL notifies the host that the commit operation for the transaction has finished.

A commitLog page can contain only a limited number of L2P mappings,[2] whereas diffL2P can have more L2P mappings than the that. In this case, MV-FTL stores multiple commitLog pages with *overflow* markers. That is, supposing a commitLog page can have up to l mappings while a committed diffL2P has N L2P mappings, $(k-1)*l < N \leq k*l$, MV-FTL should write k commitLog pages *atomically*. To do this, from the first to the $(k-1)$-th pages, MV-FTL marks the page as *overflowed*; MV-FTL sets the n_c value at the each commitLog page header as 0, putting l entries to the page. Finally, for the k-th page, MV-FTL puts all the remaining L2P mappings to the page and sets n_c as non-zero value.

Using the overflow markers, MV-FTL can filter out *partially written* diffL2Ps, thereby guaranteeing the atomicity and durability of transactions, On next boot time, MV-FTL reads each of the commitLog pages in the commitLog blocks in reverse order, from the last page back to the first page. If MV-FTL meets a page with $n_c = 0$, it simply discards the page and go on to the previous page until it first meets a page with non-zero n_c value. After finding such a page, MV-FTL reads all the commitLog pages — from the oldest to the page with non-zero n_c value—and reflects the L2P mappings in these pages to the main L2P table.

Obviously, MV-FTL has only a limited number of commitLog blocks. In addition, the fact that MV-FTL writes at least one commitLog page on each commit operation[3] aggravates the shortage of commitLog blocks, making the remaining empty spaces in the page written in vain. When there's insufficient MV-FTL first reads the commitLog pages with empty spaces from the NAND flash memory, compacts them into a smaller number of pages, and write back to commitLog blocks. In this way, MV-FTL gets more empty pages. Still, if this procedure cannot obtain a sufficient number of empty pages, MV-FTL stores the whole L2P table into NAND, making the old commitLog pages which have already been applied to the main L2P table to become obsolete. However, these saving the whole L2P table may worsen the expected lifetime of MV-FTL—it generats more NAND page write operations. Therefore, for future research, we need more elaborated mechanisms to not store the whole L2P table at a time.

In our SQLite evaluations, for reference, we allocated a only 4 blocks for the commitLog blocks but it seemed to be sufficient. Also, we believe the increased storing the L2P tables were almost negligible compare to the MV-FTL's reducing the number of data pages written to NAND by almost up to $1/2$.

3.4 Snapshot Isolation Based on MV-FTL

MV-FTL provides snapshot isolation [11] using CDL. When a read-only transaction starts, the transaction first ask to MV-FTL about the ID of the most recently committed diffL2P. Accordingly, MV-FTL returns the ID of the tail diffL2P of the CDL, incrementing the reference count (refcount) of the diffL2P by 1. From then on, as in the update case, whenever the transaction reads from the MV-FTL, it tags the access with the ID.

On receiving a read access, the MV-FTL first finds the diffL2P that corresponds to the ID and checks the diffL2P to find whether it contains an L2P mapping that matches the read access. If it matches, then the MV-FTL reads from the NAND using L2P mapping; if not, the MV-FTL goes to the previous diffL2P on the CDL and repeats the checking. If no matching L2P mapping is found in any of the preceding diffL2Ps on the CDL, the MV-FTL reads the NAND page using the main L2P table. Algorithm 2 shows the detailed procedure of the read operation in MV-FTL.

After a while, if the transaction completes its job, it notifies the MV-FTL that it will no longer use the assigned diffL2P ID. Then, MV-FTL decrements the refcount of the

2. Up to 1023 L2P mappings when a NAND page is 8-Kbytes long.

3. To guarantee the atomic commit, MV-FTL should write a page even when there are only a few L2P mappings to store.

designated diffL2P, to check if the diffL2P needs to be purged or not. (See Section 3.5.)

A read-write transaction also reads from the MV-FTL, and therefore, it requires snapshot isolation. As discussed earlier in this section, MV-FTL gives the ID of a new diffL2P to write on to the read-write transaction, instead of the most recently committed diffL2P's ID. To smoothly provide snapshot isolation for the read-write transaction, MV-FTL puts a pointer to the tail of the CDL into the newly-created diffL2P, incrementing the tail's refcount by 1—this pointer will be removed when the corresponding transaction commits, decrementing the refcount of the pointed diffL2P by 1. By following the pointer, a read-write transaction can also make full use of snapshot isolation, using the same read algorithm.

Algorithm 2. ReadPage(did, logicalAddr)

```
    Input:
    logicalAddr : logical Address of data to read
    did : ID of diffL2P referred by the current transaction
    Output:
    data : data stored in logicalAddr
 1: iterDiffL2P ← diffL2P[did];
 2: while iterDiffL2P ! = NULL do
 3:     foreach L2PEntry e in iterDiffL2P.entries do
 4:         if e.logicalAddr == logicalAddr then
 5:             return nandRead(e.physicalAddr, PAGE_SIZE);
 6:         end
 7:     end
 8:     iterDiffL2P ← iterDiffL2P.prev;
 9: end
10: foreach L2PEntry e In L2PTable.entries do
11:     if e.logicalAddr == logicalAddr then
12:         return nandRead(e.physicalAddr, PAGE_SIZE);
13:     end
14: end
```

Based on the snapshot isolation, MV-FTL becomes free from read anomalies [11]. However, at the same time, MV-FTL still needs help from DBMSs, for MV-FTL cannot prevent concurrency anomalies such as write skew [23]. In Section 4.1.2, we show how we modified SQLite to provide such aids.

3.5 Version Purging

MV-FTL monitors the reference count (refcount) of every committed diffL2P. Specifically, MV-FTL checks the refcount of diffL2P whenever the status of the related transaction changes, i.e., at every transaction end, commit, or abort time. If the refcount of a committed diffL2P becomes 0, the MV-FTL triggers the diffL2P purge operation, which is as shown in Algorithm 3.

When a diffL2P's refcount becomes 0, the MV-FTL does not have to keep the version. MV-FTL deals with the diffL2P according to the relative position on CDL, which can be categorized as follows:

- *The diffL2P is at the tail of CDL* (lines 2-3). In this case, the diffL2P is the most recently committed one, and MV-FTL simply does nothing and returns.
- *The diffL2P is at the head of CDL* (lines 4-10, 20-21). In this case, the diffL2P is the oldest committed one, and all the other transactions will get a newer version. In other words, no other transaction will be affected even when the diffL2P and the main L2P table are merged. Hence, MV-FTL applies the L2P mappings on the diffL2P to the main L2P table (lines 6-10), and then frees the diffL2P (lines 20-21).
- *The diffL2P is neither head nor tail of CDL* (lines 11-21). In this case, there exists some transactions that access older versions. Hence, MV-FTL cannot apply the L2P mapping to the main L2P table; otherwise, it will affect the older versions. Instead, MV-FTL merges the diffL2P with the one next to it along the CDL (lines 13-19). While merging these two diffL2Ps, there may be L2P mappings with the same logical address. Then, MV-FTL will invalidate the older one (lines 15-16), as the older one will no longer be needed. After merging, MV-FTL frees the diffL2P (lines 20-21).

Algorithm 3. PurgeDiffL2P(did)

```
    Input: did : diffL2P referenced by the current transaction
    Output: None
 1: victim ← diffL2P[did];
 2: if victim is tail of CDL then
 3:     return;
 4: else
 5:     if victim is head of CDL then
            /* apply all L2Ps in victim to the L2P table        */
 6:         foreach L2PEntry e in victim do
 7:             tEntry ← L2PTable.entries[e.logicalAddr];
 8:             InvalidatePage(tEntry.physicalAddr);
 9:             tEntry.physicalAddr ← e.physicalAddr;
10:         end
11:     else
            /* victim is neither head nor tail of CDL        */
12:         next ← victim.next;
13:         foreach L2PEntry e in victim do
                /* check if next already has a L2P mapping for
                   e.logicalAddr   */
14:             en ⇐ next.getEntry(e.logicalAddr);
15:             if en != NULL then
16:                 invalidatePage(e.physicalAddr);
17:             else
                    /* add e only when next does not have it     */
18:                 next.insertEntry(e);
19:         end
20:     RemoveFromCDL(victim);
21:     FreeDiffL2P(victim);
```

We designed MV-FTL to not remove the tail diffL2P. This design decision is to assure that there is at least one diffL2P in CDL; by never removing the tail diffL2P, the condition will be sufficiently guaranteed. Of course, MV-FTL definitely purges the tail diffL2P, as time goes by; that is, when a transaction commits, the corresponding diffL2P will soon take the tail position. Therefore, the previously-been-at-the-tail diffL2P can now be removed as an intermediate node—neither tail nor head.

The main advantage of MV-FTL's purging operation is that it is almost free, especially when compared to the garbage-collection operations in MVCC DBMSs. That is, while a DBMS need to read and write back the data pages to

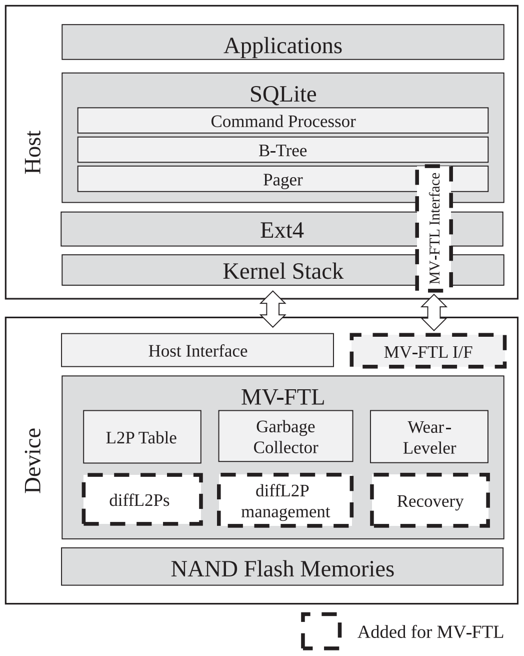


Fig. 5. Overall architecture of MV-FTL/SQLite system.

physically erase expired versions for a garbage-collection, MV-FTL have only to *invalidates the L2P mappings* of the expired versions. As an L2P mapping is far smaller than a data page itself—practically thousands of times smaller, the cost will be almost negligible. Moreover, MV-FTL rarely has to write the L2P mappings even as long as it has sufficiently large DRAMs enough to accomodate all its data structures—it was true even in the OpenSSD platform [26], which has relatively limited HW resources compared to commodity SSDs. (We will discuss further in the next section.)

The fact that purging operations are almost free in MV-FTL additionally enhances both MV-FTL itself and the storage system based on MV-FTL. First, MV-FTL can *aggresively* invoke a purge operation whenever it finds a diffL2P with zero refcount, thereby minimizing the number of diffL2Ps and keeping the memory requirement of MV-FTL as small as possible. Furthermore, from the storage system's point of view, it no longer has to suffer from *garbage-collection pause* or such.

4 Case Studies 1: SQLite

4.1 Implementation

Being intensely focused on its motto, "*Small. Fast. Reliable.*",[4] SQLite implements just a fundamental level of storage management. Although this makes SQLite beloved especially in the mobile area, many researchers have reported that the storage management in SQLite causes degradation in the performance of its transaction processing [16], [24]. In addition, SQLite is a good testbed to show the effectiveness of MV-FTL because it has a straightforward architecture.

To evaluate MV-FTL, we first modified SQLite [12]. More specifically, we simply turned off WAL mode, the original MVCC mode of SQLite and, instead, added an additional small amount of codes to communicate with MV-FTL, In addition, we implemented an MV-FTL on *OpenSSD* (described in Section 4.1.1) and modified SQLite to cooperate with MV-FTL (Section 4.1.2). Furthermore, for communication between OpenSSD and SQLite, we implemented additional interfaces from OpenSSD to SQLite in the Linux kernel (Section 4.1.3). The overall architecture of our system is shown in Fig. 5.

4. This motto can be seen on the top-right of the SQLite homepage [12].

4.1.1 MV-FTL Implementation on OpenSSD

We implemented MV-FTL by modifying the open source greedy FTL of OpenSSD.

We first created modules that implement the idea described in Section 3 and incorporated them with the greedy FTL. In particular, we allocated 16 KB of DRAM memory for storing the diffL2Ps. As the size of each diffL2P entry is 16 bytes, the MV-FTL can store approximately 900 diffL2P entries in the DRAM. If there are too many transactional L2P versions, i.e., more than 900 diffL2P entries, the MV-FTL can accommodate no more diffL2P entries. In such cases, MV-FTL denies any transactional update and notifies the host about the update failure. However, in practice, the chance of such failures is rare, because the transactions are usually very short. Empirically, although we conducted a variety of experiments, including TPC-C and the RL Benchmark, several times, we never experienced such a failure, and the total number of diffL2P entries never exceeded 600.

To support transactional operations for MV-FTL, we extended the SATA interfaces as in X-FTL [16]. More precisely, we extended the SATA read/write commands such that they can carry the ID of transactions. Furthermore, we extended the SATA TRIM command such that the host can notify MV-FTL about the start, commit/abort, and end of a transaction through the command.

4.1.2 Modifications to SQLite

To fully utilize the version management features provided by MV-FTL, we modified two SQLite modules, namely the Pager, which oversees storage management, and the OS Interface. We extended the procedures related to the transaction start/end in the Pager such that they transmit proper information to MV-FTL in accordance with the stage of each transaction.

Moreover, because MV-FTL does not require a locking for any read-only transaction, we just turned off the read locks in SQLite. The fact that readers are free from locking elavates the degree of concurrency for read transactions higher—theoretically even higher than SQLite WAL mode. In fact, SQLite WAL mode still requires every read transaction to acquire a read lock in advance, which is to simplify the version management.

Remark. Although MV-FTL does not need any lock for read-only transactions, read-write transactions still need to acquire an exclusive lock for DB in advance. However, this does not hurt the concurrency, as any mode in the SQLite does the same.

4.1.3 Modifications on Other Layers

In addition to SQLite and MV-FTL, the middle layers, such as the file system, block layer, and device driver, also participate in transactional communication, which means that all the layers require an extension to enable the communication.

TABLE 1
Evaluation Setup

CPU	Intel i5-3470
RAM	8 GB
OS Kernel	Linux kernel 3.16.1 (*modified*)
File System	ext4 (*modified*)
Database	SQLite v3.7.13 (*modified*)
Storage (OS, main programs)	Samsung 850 SSD 128 GB
Storage (being tested)	Indilinx Jasmine Barefoot OpenSSD with 4 Samsung MLC 64 Gb NAND flash memories (32 GB in total) Page size: 8 KB, Block Size: 1 MB

TABLE 2
RL Benchmark Transaction Sequences

Txn#	Transaction
1-1000	Insert 1 item
1001-1002	Insert 25000 items
1003	1000 range updates
1004	Update 2500 items
1005	Insert into t1 from t2; Insert into t2 from t1;
1006	Deletion with "LIKE" statement
1007	Deletion using Range
1008-1010	Drop t1, t2 and t3, respectively.

In this paper, again as in X-FTL [16], we extend the interface of each layer such that each layer simply passes the transactional requests to the lower layer until they reach the MV-FTL.

Meanwhile, the Linux kernel uses the page cache to accelerate read/write accesses to the storage device. The page cache replaces page I/O operations with the quicker main-memory I/O operations. At this point, in order to incorporate the page cache with our MV-FTL, the former should also be aware of the concept of versions. That is, if SQLite requests the page of a specific version, the page cache first checks whether the page is in the cache, and then checks whether the page is of the requested version. However, because this modification was beyond our capabilities, we omitted the page cache modification for the present study, leaving this for a future work; instead, we modified SQLite to use direct I/O so that we can skip the page cache. Even though we limited MV-FTL to use direct I/O only, we hope that soon we could present the evaluation result using buffered I/O.

4.2 Evaluation

Based on the implemented system, we evaluated the effectiveness of storage-level multi-version concurrency control by comparing the performance of MV-FTL and modified SQLite with those of SQLite's two original journal modes [12], RBJ and WAL. We also compared them with X-FTL [16], one of the previous studies that enhanced the overall performance by exploiting the unique update scheme of flash storages. For evaluation, we first executed SQL transactions captured from the RL Benchmark [17], and while executing the workload, we added several concurrent read accesses such that those read accesses could inhibit the execution of the RL Benchmark. For a more realistic evaluation, we executed the DBT2 [25] benchmark, a TPC-C-like benchmark for SQLite. In this experiment, we first added concurrent read access, as in the RL Benchmark experiment.

In addition, we compared how many pages the host writes to the MV-FTL while executing these workloads in each mode. The results suggest that using MV-FTL reduces the number of pages that needs to be written for executing transactions, which can prolong the overall lifetime of a flash storage and improve the overall system performance.

4.2.1 Evaluation Setup

We implemented MV-FTL on top of the Jasmine OpenSSD Platform [26]. The platform consists of a Barefoot controller that contains an 87.5 MHz ARM7 processor, 96 KB of SRAM for firmware codes, 64 MB mobile SDRAM for metadata, and 4 NAND flash memories of 8 GB each from Samsung. This platform supports SATA 2.0 on the host interface.

At the host side, we modified SQLite to co-operate with the MV-FTL, file system, and Linux kernel in order to facilitate the communication between SQLite and MV-FTL.

For a competitor, we implemented X-FTL on the same OpenSSD and modified SQLite, the Linux kernel, and so on, based on the description of the paper [16]. We also compared the evaluation results with the original RBJ and WAL modes. For those modes, we used the unmodified version of OpenSSD, SQLite, and the Linux kernel.

All the evaluations were conducted on the same machine. The machine's detailed information is summarized in Table 1.

4.2.2 Concurrency Evaluation

To evaluate MV-FTL and other modes, we used two popular benchmark, each of which is described below:

- We first captured SQL queries using RL Benchmark [17], which is one of the most popular benchmark applications tatevaluates the performance of SQLite on Android phones. RL Benchmark consists of tens of thousands of insert/update SQL queries that consist of transactions of various lengths–some consist of just a single statement while others contain thousands of statements. (See Table 2 for details) We extracted the SQL queries while running the RL Benchmark application on an Android smartphone and executed them on our system.
- We used DBT2 [25], a TPC-C [18]-like benchmark that emulates online transaction processing (OLTP) workloads. A DBT2 worker selects and executes one of the five transactions based on the predefined probabilities as listed in Table 3. We measured the number of completed new-order transactions per minute (NOTPM) as a key factor of performance.

Table 4 summarizes the key features of each mode to be evaluated. This suggests that SQLite using the RBJ and WAL modes are expected to suffer from overhead for guaranteeing transaction atomicity in DBMS-level,[5] whereas SQLite on X-FTL and MV-FTL are free from such overheads because they are effectively exploiting the OoP characteristic for the

5. In fact, RBJ suffer more than WAL because RBJ generates more blocking operations, i.e., more **fsync()** operations.

TABLE 3
Types of Transactions for DBT2

Name	Weight	I/O type	Freq(%)
Delivery	Batch	Read-write	4%
Order-status	Mid-weight	Read-only	4%
Payment	Light-weight	Read-write	43%
Stock-level	Heavy-weight	Read-only	4%
New-Order	Mid-weight	Read-write	45%

TABLE 4
Features of Each Mode to Evaluate

Mode	FTL-Assisted?	Multi-version?
RBJ	No	No
WAL	No	Yes
X-FTL	Yes	No
MV-FTL	Yes	Yes

atomicity. On the contrary, in terms of concurrency, the performance is expected to degrade for RBJ and X-FTL, because there was only a single version for each dataset, whereas WAL and MV-FTL can process multiple transactions more fluently using multiple versions. We proved that these expectations are practically valid in the evaluations.

RL Benchmark with Concurrent Read Accesses. Although RL Benchmark is widely used to evaluate the performance of SQLite, RL Benchmark itself does not evaluate transaction concurrency. RL Benchmark is single-threaded, and thus there is no concurrent access. However, in this paper, to evaluate concurrency, we supposed an *artificial but probable* scenario in which a read-only process repeatedly checks the status of the database while another process is executing RL Benchmark on the same database.[6] The read-only process executes a read-only status-checking query N times every second. For example, if N is zero, the read-only process never executes the query; if N is ten, the process executes ten queries per second,—in other words, it executes one status-checking query every 100 ms. By adjusting N, we can implicitly control the *degree of concurrency*. As N grows, the status check operation obstructs RL Benchmark execution more by causing *readers-writers conflict*.

Fig. 6 shows the elapsed times of the RL Benchmark with the above check processes fully implemented, varying the number of check operations per second, N, from 0 to 25. Various cases of N are described as follows:

- *When N is 0:* X-FTL and MV-FTL execute approximately 1.3 to 1.4 times faster than the WAL modes, and three times faster than the RBJ mode. The performance improvements result from exploiting the append-only update characteristic of FTL for guaranteeing atomicity of transaction. In this case, both X-FTL and MV-FTL reduce the need to write additional data for assuring transaction atomicity.
- *When N is greater than zero:* While the execution times of MV-FTL are almost the same, those of X-FTL increase rapidly with N. The execution times of X-FTL become even longer than those of the WAL mode, when N is greater than 13; eventually, X-FTL fails to execute RL Benchmark when N is greater than 20, because the read accesses in the check query repeatedly blocks the execution of RL Benchmark.

In all the cases, SQLite's RBJ mode performs the worst for the following reasons: 1) SQLite in the RBJ mode suffers from the amplified number of writes, which can be twice as many page writes in the worst case, caused by copying the original pages to the RBJ file and then updating the database file; 2) SQLite in the RBJ mode does not provide any concurrency, because the database in the RBJ mode has only a single version for each data.

6. We implemented status checking simply to avoid possible noise. More specifically, a status check operation first reads a table with a single entry and then sleeps for 50 ms of *thinking* time.

We observed that X-FTL is slightly better than MV-FTL when N is between zero and one. The main reasons for these slight inferiorities are: 1) the L2P mapping algorithm in MV-FTL is slightly more complex than that in X-FTL, and 2) X-FTL is free from version-purging overhead. However, we argue that such disadvantage could be negligible, because MV-FTL is better than X-FTL when there are more than three check operations per second.

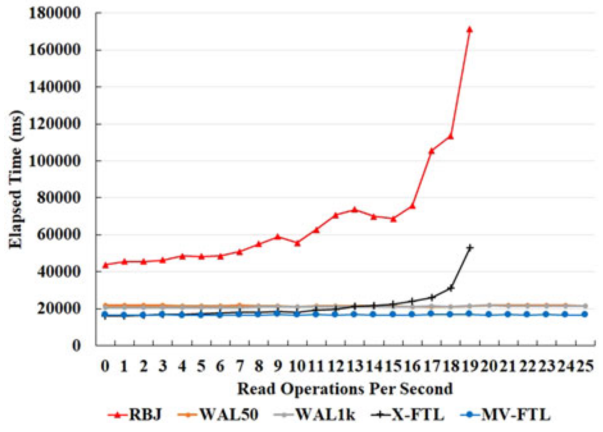
DBT2 with Concurrent Read Access. Next, we conducted an experiment using a configuration similar to that of the previous experiment: while running DBT2, an independent process concurrently checks the status of the database, as described in Section 4.2.2. We again varied the number of check operations per second, N, from 0 to 25.

Fig. 7 shows that 1) MV-FTL recorded 689 NOTPM on average, which is almost 2.6 times more than that of WAL1k (avg. 269 NOTPM) and 2.8 times more than that of WAL50 (avg. 245 NOTPM); 2) whereas the NOTPM of X-FTL degrades gradually as N increases. The NOTPM of MV-FTL sustains regardless of N.

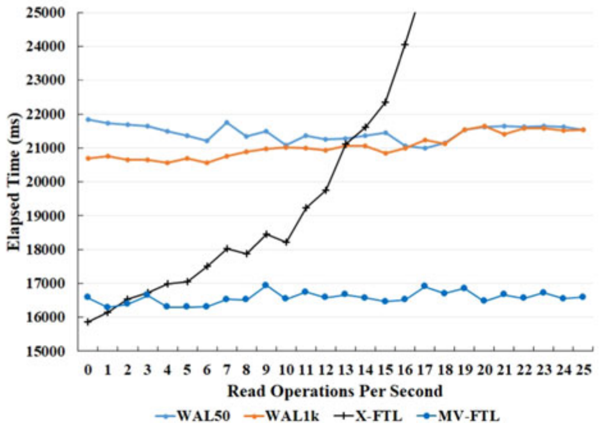
These findings are consistent with the RL Benchmark evaluation. In fact, the performance gain of MV-FTL is much greater than that of WAL modes, in the DBT2 evaluation. We surmise that this is due to the basic differences between the two benchmarks. Whereas RL Benchmark starts with an empty database, DBT2 starts with a relatively large database, which initially takes more than 85 MBs just to store a single warehouse. Moreover, whereas update queries in RL Benchmark are quite elementary, those in DBT2 are more complex, causing the garbage collection overhead to become aggravated.

DBT2 with Multiple Concurrent Workers. Next, we evaluated the four modes using DBT2 by varying the number of concurrent workers from 1 to 20.

In this configuration, each worker independently selects and executes DBT2 queries on the same database. Therefore, unlike the previous experiments where we can control the frequency of simultaneous read accesses, here we have no way to increase only the concurrent read accesses only, such that the increased degree of concurrency degrades the performances of the multi-versioned modes (i.e., the WAL mode of SQLite and MV-FTL). However, the degradations are much less in the multi-versioned modes, because they are free from conflicts between read and write operations, whereas RBJ and X-FTL still suffer from these conflicts.



(a) Full, including RBJ



(b) Partial

Fig. 6. RL Benchmark execution times when concurrent processes are reading from the same database.

Fig. 8 shows the results of our evaluation, which indicates that the performance of each mode, except for RBJ, converges to some degree as the number of workers increases. However, as expected, the performance drop is much slower in MV-FTL than in X-FTL. More precisely, the performance of X-FTL dropped by less than 50% even with two clients; however, in the case of MV-FTL, the performance drops are more gradual. Our results suggest that, although the performance drops are inevitable as the degree of concurrency increased, MV-FTL handles the conflicts caused by the concurrency more gracefully. Therefore, MV-FTL can be useful not only when there are many concurrent read accesses, but also when there are many concurrent read/write accesses.

4.2.3 Write Count Analysis

During the evaluations, we measured the number of pages written per transaction when there is no concurrent read access—that is, $N = 0$. We obtained these values using the `blktrace` command; i.e., while executing each workload, we first captured the block layer IO trace to the flash storage. Then, from the trace, we simply counted the number of pages written. We also counted the number of read-write transactions that were successfully committed. Using these two kinds of values, we calculated the average number of write operations per transaction.

Fig. 9 shows the number of write operations in each mode. As shown, the number of write operations in MV-FTL and X-FTL is reduced by 41 and 52 percent in RL Benchmark and DBT2, respectively, when compared with WAL1k, the WAL mode that invokes a checkpoint operation per 1,000-page update (the default option of the SQLite WAL mode). The number of write operations in WAL mode is nearly twice as many as those in MV-FTL and X-FTL. The results verify our hypothesis, i.e., the WAL mode almost doubles the number of write operations due to the garbage collections. In fact, the WAL mode increases the number to more than twice as that in the DBT2 experiment, because garbage collections accompany additional updates of metadata on the file system.

This reduction of write operations in MV-FTL explains how and why MV-FTL outperforms RBJ and WAL, i.e., the number of write operations are significantly reduced, and therefore, the overall transaction processing performance increases. In addition, in terms of lifetime, MV-FTL is expected to prolong the lifespan of a flash storage, given that the lifespan of a flash storage highly correlates with the number of pages written to it [27].

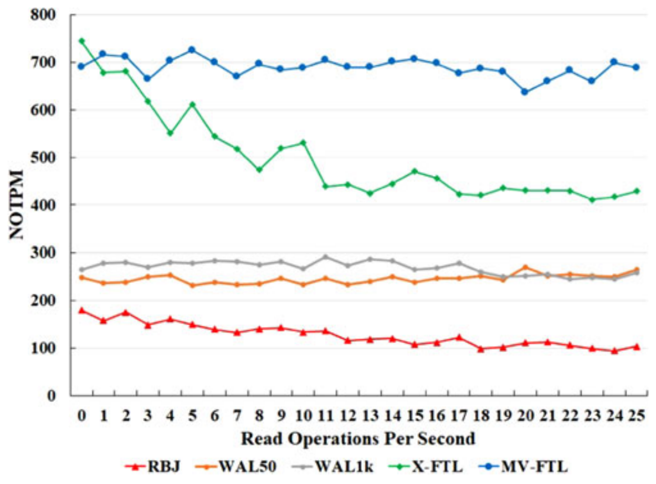


Fig. 7. DBT2 performance with concurrent processes checking the same database; measured in NOTPM (higher values are better).

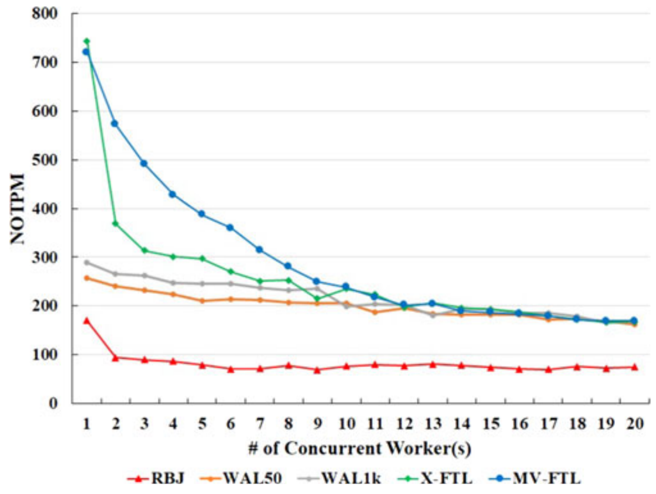
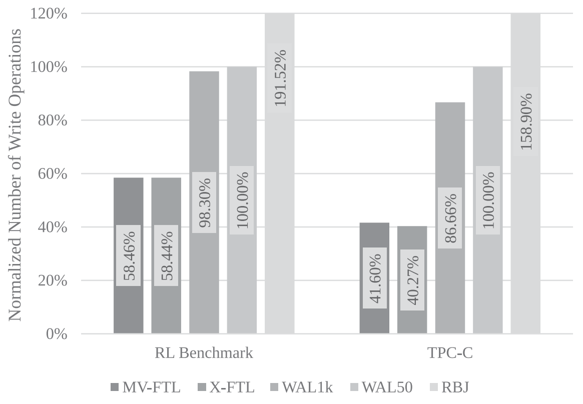


Fig. 8. DBT2 performances with multiple concurrent workers; measured in NOTPM (higher values are better).

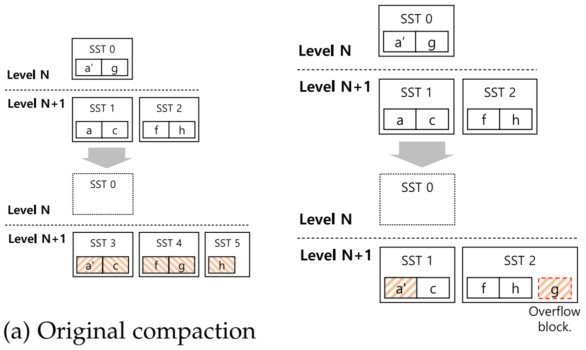
Workload	MV-FTL	X-FTL	WAL$_{1k}$	WAL$_{50}$	RBJ
RL Bench.	12.92	12.91	21.72	22.10	42.32
DBT2	17.08	16.48	35.47	40.93	65.03

(a) the number of write operations



(b) normalized w. r. t. WAL50

Fig. 9. Number of writes per transaction (and normalized) at each mode. WAL50 and WAL1k represent the results of the WAL mode whose checkpoint threshold is 50- and 1,000-page updates, respectively.



(a) Original compaction (b) Patch compaction

Fig. 10. Comparison between the original compaction and patching compaction. Blocks with diagonal stripes represent dirty blocks.

5 Case Study 2: RocksDB

Another system that can benefit from MV-FTL is RocksDB [13]. RocksDB is an open-source key-value store developed and widely-adopted by Facebook. It is becoming more and more famous, since many DBMSs like *MyRocks*, a Facebook's variant of MySQL, began to use RocksDB as their backend storage engine. According to Dong et al. [13], "the underlying storage engine for Facebook's MySQL instances is increasingly being switched over from InnoDB to MyRocks." For this reason, RocksDB is drawing considerable attention from many researchers.

RocksDB is based on Log-Structure Merge (LSM) tree [28]. It stores data in a file called *Sorted Sequence Table* (SST) file. As the name suggests, each SST file is read-only after it is created. When RocksDB needs to update a data item, it first adds the item to an in-memory table. When the in-memory table grows bigger than a certain threshold, RocksDB flushes it into a new SST file. In the meantime, older SST files are untouched regardless of data update or creation of a new SST file. After a while, when a large number of SST files have been created, RocksDB triggers a procedure called *compaction*.

For compaction, RocksDB first chooses the SST files to be compacted, usually one from lower levels and the others from higher levels[7] Then, RocksDB simply reads and merge-sorts the SST files, writing the merge-sort output into new SST files at higher levels. Finally, after the creation of the new SST file is over, RocksDB deletes the obsoleted original SST files. Fig. 10a depicts the overall compaction sequence.

One of the difficulties in RocksDB compaction is that a huge majority of the bytes written during compaction is *merely* to copy the data, which exists at the same output level [13]. RocksDB reads and writes large files at levels higher than 2, only applying a small amount of changes from a lower-level file. This comes from an intrinsic design property of the LSM tree, that is, the total size of SST files at level $n+1$ is designed to be r times bigger than that at level n, where r is usually 10 in RocksDB. This property effectively limits the space amplification of the LSM tree to no more than $r/(r-1)$, but, at the same time, results in severe write amplifications for compacting SST files. In other words, RocksDB reads and re-writes r records at the level $(n+1)$ on average to compactize a single record from level n.

To cope with this write amplification, we propose a new compaction algorithm called *patch compaction*. In patch compaction, instead of copying the data at the same level, RocksDB simply writes new or updated parts to the original SST file. (See Fig. 10a) In this way, the use of patch compaction in RocksDBs can significantly reduce the number of bytes to be written. For example, in the case of Fig. 10, RocksDB should originally copy all the 5 data items into a new file. However, using patch compaction, RocksDB needs to write only a' and g to the new SST file.[8]

This patch compaction itself, however, can harm the concurrency of the whole system. That is, RocksDB cannot not access an SST file when the file is under patch compacted. Moreover, RocksDB suffer from recovery if some failure occurs during patch compaction, as it directly updates the original file.

Using MV-FTL, however, these concerns get simply banished. By using MV-FTL, RocksDB can freely read the older versions of the SST file even when the SST file is under patch compaction. Likewise, MV-FTL addresses much of the recovery concerns caused by the patch compaction.

For evaluation, we implemented a file-based emulator that imitates the version management of MV-FTL; that is, when a transaction updates an SST file, the emulator first makes a copy of the existing SST file and then updates the copy, keeping the existing file intact. In addition, we implemented the patch compaction on RocksDB to cooperate with the MV-FTL emulator. Then, we executed LinkBench and TPC-C on MyRocks [22] and measured how many bytes

7. Here, lower level means the level with lower level number–that is, L2 is *lower* than L3.

8. Fig. 10b may seem strange as SST file 2 after compaction seems to be not *sorted* anymore. However, there is no problem, as the SST files in RocksDB have already been equipped with *index*; that is, SST files do not have to be literally sorted but patch compaction should be performed to modify the index properly.

TABLE 5
Comparison on Number of Written Bytes between the Original RocksDB Compaction without MV-FTL(Original) and the *Patch Compaction* with MV-FTL (MV-FTL)

	Bytes written(GB)		
Workload	Original	MV-FTL	Reduced(%)
TPC-C	12.88	10.65	17.30
LinkBench	7.10	4.23	40.42

were written to the storage[9] with and without patch compaction. For fair comparison, we executed the same number of queries in both the cases—specifically, 250 K New-Order transactions on TPC-C and 10 M requests on LinkBench.

Table 5 shows the number of bytes written while running TPC-C and LinkBench. Clearly, patch compaction based on MV-FTL reduced the number by 40% in LinkBench and 17% in TPC-C. Although these results do not explicitly guarantee performance improvements, these may imply the possibility of improvements using MV-FTL. At the least, MV-FTL will prolong the lifespan of an SSD by writing fewer bytes to SSDs, given that only a limited number of bytes can be written in an SSD during its lifetime.

6 Related Works

6.1 Studies Utilizing Commodity SSDs

Since their first appearance, SSDs have received much attention from researchers. Researchers have mainly focused on SSDs superior performance in handling random access, and they have tried to adapt SSD to conventional systems, which were mostly designed for traditional storage such as HDDs.

Zhang et al. [5] compared several types of system design for multi-tenancy workload on SSD-based I/O subsystems and noticed that the effectiveness of SSD on random data access affects the performance of the multi-tenancy system. Lee et al. [29] evaluated SSD performance and presented the benefits of leveraging SSD in enterprise database applications. They showed that exploiting SSD in operations related with transaction logs, multi-version control, and temporary table spaces is best suited for driving maximum application performance. Chen et al. [30] maximized SSD performance in online analytical processing (OLAP) operations by avoiding unnecessary writes of small units.

Although these studies cleverly exploited the performance characteristics of commodity SSDs, they did not utilize the intrinsic characteristics of SSDs, which contains a great potential that may further enhance DBMSs.

6.2 Studies Extending SSDs

Some researchers have proposed extending flash-based storage devices and offloading DBMS tasks to the device to improve the performance of the entire system. Smart or intelligent SSDs [3], [31], [32], Willow [33], and multi-streamed SSD [34] are notable examples of such approaches. They proved that system-level performance can be enhanced by giving more information and delegating processing tasks to SSDs. However, they are not intended to enhance the degree of concurrency as in our MV-FTL.

Meanwhile, some researchers have proposed some revolutionary flash-based storage solutions that support transactional updates using the OoP characteristic [6], [35], [36]. Their approaches successfully enhanced both the performance and the concurrency of DBMSs. Still, since they are too revolutionary—they require the overall storage subsystem to be redesigned [6], [35], [36]; they rely on a special functionality, partial page programming, that is rarely available on mobile flash-based storage [6], [35]; or they need dedicated servers and computing power [36], [37].

6.3 MVCC on Enterprise-Level DBMSs

Today, there are many DBMSs in the enterprise market. Such DBMSs usually have complicated MVCC schemes, which manage versions in more fine-grained units, such as tuples or record units.

PostgreSQL [10] is a great example. When PostgreSQL updates a tuple, instead of immediately overwriting the existing tuple, it appends the updated tuple into storage, keeping the old ones untouched. In addition, PostgreSQL adds a timestamp and a previous-version pointer to the tuple; the timestamp is used to represent the specific version, and the pointer is used to retrieve the previous version.

This fine-grained lock has many advantages. In particular, tuple-level version management enables PostgreSQL to manage tuple-wise locks. When a transaction wants to update some tuples, it need only acquire locks for each of the tuples, and therefore other transactions are blocked only when they are to access tuples modified by the updating transaction. In this way, the tuple-grained version management significantly improves the DBMS concurrency level. In addition, the appending operations are well optimized, harmonizing with the buffer manager.

However, because PostgreSQL has been optimized for traditional block devices such as HDDs, PostgreSQL does not exploit the OoP characteristic of SSDs. For example, PostgreSQL has an independent auto-vacuum daemon that periodically purges unnecessary old versions. The daemon reduces the space overhead of managing multiple versions and gives the database a more compact form. However, from an SSD point of view, the auto-vacuum seems somewhat burdensome because SSDs have already been equipped with well-optimized garbage collection mechanisms for handling updates in an OoP, append-only manner.

6.4 Studies Extending MVCC

There are also many studies on reducing MVCC overhead. First, Neumann et al. [38] implemented MVCC on a main-memory database system called Hyper [39]. They implemented an in-place update scheme with undo log buffers for version control and adapted a precision locking method [40] to support serializability for snapshot isolation.

Jones et al. [41] extended H-store [42] to support a low-overhead concurrency control scheme. To take advantage of multiple machines and CPUs, H-Store divides data into partitions so that transactions are distributed to partitions as much as possible. In this manner, most transactions can work independently without the overhead generated from

9. While measuring, we excluded the overheads caused by the emulator itself.

concurrency control. With transactions that have data dependencies, they proposed speculative execution to improve throughput by hiding two-phase commit latency.

Saxena et al. [43] implemented a prototype of a main-memory transactional store by extending TinySTM [44]. In addition, they proposed a partitioned logging method for durability, which assigns transactional logs to cores so that logs are independently flushed to separate locations in the SSD. Partitioned logging leverages multi-core systems and the internal parallelism of SSDs by saturating the SSD with outstanding requests. Because MV-FTL handles transactional control based on the SSD, unlike studies suggesting transactional support in main memory, MV-FTL focuses on reducing the storage I/O costs, which are dominant over the costs of memory access.

7 Conclusion and Discussion

So far, we have introduced an FTL that provides page-level multi-version management. Because it is based on the intrinsic OoP characteristic of SSDs, MV-FTL manages multiple page versions in a highly efficient way, thereby relieving the burdens of multi-version management of DBMSs.

On the basis of MV-FTL, we implemented a lightweight concurrency control system with SQLite. The system showed better transaction-processing performance, which is equivalent to X-FTL, one of the best works to our knowledge, while the performance were not affected by any conflicting read operations. Additionaly, from a code complexity point of view, SQLite could become *lighter*, freed from complex RBJ and WAL codes.

Furthemore, we designed a new compaction algorithm, Patch Compaction for RocksDB, which efficiently utilizes MV-FTL. We conducted LinkBench and TPC-C benchmark on the MyRocks, using an MV-FTL emulator that provides the storage-level multi-versioning. From the evaluations, we observed that the total number of bytes written was significantly reduced by using MV-FTL. Although these results are merely based on emulation, they suggest that the write overhead on compaction could be significantly reduced by using MV-FTL. Consequently, we confirmed the feasibility that MV-FTL could also be useful in more sophisticated DBMSs – at the least, reducing the number of writes directly lowers the maintenance cost of the SSD-based enterprise systems.

MV-FTL still has room for further improvement. First, currently there exist a big gap in the version management units in MV-FTL and the cutting-edge DBMSs. For MV-FTL to be adopted in more complex, enterprise-level DBMSs, it is necessary to manage versions in smaller units. To do so, additional supports from DBMSs would be necessary.

Moreover, although MV-FTL can control the conflicts between reads and writes, it cannot resove the conflicts between write transactions. For this reason, MV-FTL still needs additional supports from the DBMS for handling conflicts between writes—practically, we relied on the SQLite's original write lock mechanism on our case study. One possible method for addressing this issue is to introduce optimistic concurrency control [45] to MV-FTL. As a vision moving forward, the *melding tree* [37], [46] of Hyder could be a direction to take for such extension.

Meanwhile, our patch compaction codes for RocksDB are still premature, and some logical inefficiencies are apparent. We believe we can soon improve the algorithm to be even more efficient, such that the benefit from MV-FTL increases.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2015R1A2A1A05001845).

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance." in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.
[2] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 181–192, 2009.
[3] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: Opportunities and challenges," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 1221–1230.
[4] I. Jo, et al., "Yoursql: A high-performance database system leveraging in-storage computing," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 924–935, 2016.
[5] N. Zhang, J. Tatemura, J. Patel, and H. Hacigumus, "Re-evaluating designs for multi-tenant OLTP workloads on SSD-basedI/O subsystems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2014, pp. 1383–1394.
[6] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2007, pp. 55–66.
[7] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang Oper. Syst.*, 2009 pp. 229–240.
[8] "Oracle database concepts: Chapter 13 data concurrency and consistency multiversion concurency control." (2017, Oct. 6). [Online]. Available: http://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm#i17881
[9] "MySQL 5.7 Reference Manual." (2017, Oct. 6). [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/
[10] D. R. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1850–1861, 2012.
[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 1995, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/223784.223785
[12] "SQLite Home Page." (2017, Oct. 6). [Online]. Available: https://www.sqlite.org/
[13] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Stumm, "Optimizing space amplification in RocksDB," *Conf. Innovative Data Systems Research (CIDR)*, 2017.
[14] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 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 *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 301–311.
[16] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite databases," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 97–108.
[17] "RL Benchmark: SQLite." (2017, Oct. 6). [Online]. Available: https://goo.gl/BsVCMY
[18] "TPC-C Specification." (2017, Oct. 6). [Online]. Available: http://www.tpc.org/tpcc/
[19] "Linkbench." (2017, Oct. 6). [Online]. Available: https://github.com/facebookarchive/linkbench
[20] 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 Trans. Embedded Comput. Syst.*, vol. 6, no. 3, 2007, Art. no. 18.

[21] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, vol. 4. New York, NY, USA: McGraw-Hill, 1997.

[22] D. Borthakur, "Under the hood: Building and open-sourcing RocksDB," 2013. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920/

[23] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, "Making snapshot isolation serializable," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 492–528, 2005.

[24] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 309–320.

[25] "OSDL Database Test 2(DBT-2)." (2017, Oct. 6). [Online]. Available: http://osdldbt.sourceforge.net

[26] "OpenSSD Project." (2017, Oct. 6). [Online]. Available: http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform

[27] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 286–297, 2011.

[28] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[29] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2008, pp. 1075–1086.

[30] Z. Chen and C. Ordonez, "Optimizing OLAP cube processing on solid state drives," in *Proc. 16th Int. Workshop Data Warehousing OLAP*, 2013, pp. 79–84.

[31] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–12.

[32] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, and C. Park, "Intelligent SSD: A turbo for big data mining," in *Proc. 22nd ACM Int. Conf. Inf. Knowl. Manag.*, 2013, pp. 1573–1576.

[33] S. Seshadri, et al., "Willow: A user-programmable SSD," in *Proc. 11th USENIX Symp. Oper. Syst. Des. Implementation*, 2014, pp. 67–80.

[34] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. 6th USENIX Workshop Hot Top. Storage File Syst.*, 2014, pp. 13–13.

[35] S.-W. Lee and B. Moon, "Transactional in-page logging for multi-version read consistency and recovery," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 876–887.

[36] P. A. Bernstein, C. W. Reid, and S. Das, "Hyder-a transactional record manager for shared flash," in *Proc. CIDR*, 2011, vol. 11, pp. 9–20.

[37] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan, "Optimistic concurrency control by melding trees," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 944–955, 2011.

[38] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2015, pp. 677–689.

[39] A. Kemper and T. Neumann, "Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 195–206.

[40] J. Jordan, J. Banerjee, and R. Batman, "Precision locks," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 1981, pp. 143–147.

[41] E. P. Jones, D. J. Abadi, and S. Madden, "Low overhead concurrency control for partitioned main memory databases," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 603–614.

[42] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (It's time for a complete rewrite)," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 1150–1160.

[43] M. Saxena, M. A. Shah, S. Harizopoulos, M. M. Swift, and A. Merchant, "Hathi: Durable transactions for memory using flash," in *Proc. 8th Int. Workshop Data Manag. New Hardware*, 2012, pp. 33–38.

[44] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proc. 13th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2008, pp. 237–246.

[45] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[46] P. A. Bernstein, S. Das, B. Ding, and M. Pilman, "Optimizing optimistic concurrency control for tree-structured, log-structured databases," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2015, pp. 1295–1309.

Doogie Lee received the BS and MS degrees in computer science and engineering from the Pohang University of Science and Technology (POSTECH) in 2003 and 2005, respectively. He is currently a PhD candidate in the Department of Computer Science, Yonsei University. He is also a senior engineer at Samsung Electronics Co. His current research interests include the database system, flash memory, and NVRAMs.

Mincheol Shin received the BS degree in computer science from Yonsei University in 2011. He is currently a PhD candidate in the Department of Computer Science, Yonsei University. His current research interests include network processor, database system, flash memory including SSD, and nonvolatile memory.

Wongi Choi received the BS degree in computer science from Yonsei University in 2014. He is currently a PhD candidate in the Department of Computer Science, Yonsei University. His current research interests include database systems, flash memory, and NVRAMs.

Hongchan Roh received the BS degree in 2006, the MS degree in 2008, and the PhD degree in 2014, from the Department of Computer Science, Yonsei University, Seoul, Korea. He is currently a research fellow at SK Telecom. His current research interests include network processor, database system, flash memory, and SSD.

Sanghyun Park received the BS and MS degrees in computer engineering from Seoul National University in 1989 and 1991, respectively. He received the PhD degree from the Department of Computer Science, the University of California at Los Angeles (UCLA) in 2001. He is currently a professor in the Department of Computer Science, Yonsei University, Seoul Korea. His current research interests include the database, data mining, bioinformatics, and flash memory.