High-Performance and Lightweight Transaction Support in Flash-Based SSDs

Youyou Lu, *Student Member, IEEE*, Jiwu Shu, *Member, IEEE*, Jia Guo, Shuai Li, and Onur Mutlu, *Senior Member, IEEE*

Abstract—Flash memory has accelerated the architectural evolution of storage systems with its unique characteristics compared to magnetic disks. The no-overwrite property of flash memory naturally supports transactions, a commonly used mechanism in systems to provide consistency. However, existing embedded transaction designs in flash-based Solid State Drives (SSDs) either limit the transaction concurrency or introduce high overhead in tracking transaction states. This leads to low or unstable SSD performance. In this paper, we propose a transactional SSD (TxSSD) architecture, LightTx, to enable better concurrency and low overhead. First, LightTx improves transaction concurrency arbitrarily by using a page-independent commit protocol. Second, LightTx tracks the recent updates by leveraging the near-log-structured update property of SSDs and periodically retires dead transactions to reduce the transaction state tracking cost. Experiments show that LightTx achieves nearly the lowest overhead in garbage collection, memory consumption and mapping persistence compared to existing embedded transaction designs. LightTx also provides up to 20.6 percent performance improvement due to improved transaction concurrency.

Index Terms—Solid state drives, flash memory, transaction recovery, transactional SSD, atomicity, durability

1 Introduction

For decades, transactions have been widely used in database management systems (DBMSs), file systems, and applications to provide the ACID (Atomicity, Consistency, Isolation, Durability) properties, but usually at the cost of implementation complexity and degraded performance. Transaction recovery, which ensures atomicity and durability, is a fundamental part of transaction management [1]. In transaction recovery, a write operation keeps the previous version of its destination pages safe before the successful update of the new version, to provide consistency in case of update failure (e.g., due to a system crash). Write ahead logging (WAL) [2] and shadow paging [3] are the two dominant approaches to transaction recovery. In WAL, a write updates the new version in the log and synchronizes it to the disk before over-writing the old version in-place. Additional log writes and the required synchronization make this approach costly. In shadow paging, a write overwrites the index pointer (the metadata to locate pages) to point to the new version after updating the new version in a new location (as opposed to over-writing the old version as done in WAL). This approach causes scattering of data over the storage space, reducing locality of read accesses, which is undesirable for high performance.

Flash memory properties of *no-overwrite* and *high random I/O performance* (comparatively to hard disks) favor the shadow paging approach. A page in flash memory has to be erased before it is written to. To hide the erase latency, flash-based SSDs[1] redirect the write to a free page by invalidating the old one and using a mapping table in the FTL (Flash Translation Layer) to remap the page. So, a page is atomically updated in SSDs by simply updating the mapping entry in the FTL mapping table. Because of this simplicity provided by the FTL mapping table, providing support inside SSDs for transactions is attractive.

Recent research [4], [5], [6] proposes to support transactions inside an SSD by introducing a new interface, *WriteAtomic*, and providing multi-page update atomicity. Transaction support in the SSD (i.e., embedded transaction support) frees the system from high-overhead transaction recovery support, and thus nearly doubles system performance due to the elimination of duplicated log writes [4], [5]. Unfortunately, these proposals support a limited set of *isolation levels*; mainly, *strict isolation*, which requires all transactions to be serialized, i.e., executed one after another. This limits transaction concurrency and hurts the usage of the storage device for transactions for two reasons. First, different systems make different tradeoffs between performance and consistency by allowing different isolation levels among transactions [7]. Not supporting a wide variety of isolation levels makes the system inflexible as it does not give the software the ability to choose the isolation level. Second, *strict isolation* limits the number of concurrent requests in the SSD (as only one transaction can execute at a time) and thus hurts *internal parallelism*, i.e., simultaneous updates of pages in different channels and planes, of the SSD. The need to

- *Y. Lu, J. Shu, J. Guo, and S. Li are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: luyy09@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn, jguo.tshu@gmail.com, lishuai.ujs@163.com.*
- *O. Mutlu is with the Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: onur@cmu.edu.*

Manuscript received 31 Dec. 2013; revised 11 Oct. 2014; accepted 29 Dec. 2014. Date of publication 8 Jan. 2015; date of current version 10 Sept. 2015.
Recommended for acceptance by K. Chakrabarty.
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/TC.2015.2389828

[1] Hereafter, we refer to flash-based SSDs as SSDs.

wait for the completion of previous transactions due to the requirement of strict isolation between transactions causes the SSD to be underutilized.

On the other hand, when supporting transaction recovery inside an SSD, it is important to keep the overhead low (in terms of both cost and performance). Unfortunately, flexible transaction support, transaction aborts and the need for fast recovery lead to high overhead. First, when transactions are concurrently executed to provide support for flexible transactions, determining the state of each transaction requires page-level state tracking, which can be costly and increases *memory overhead*.[2] Second, transaction aborts increase the *garbage collection overhead*, i.e., the time to erase blocks and to move valid pages from the erased blocks, because extra restrictions on garbage collection (GC) are needed in order not to touch the pages used for the commit protocol (as described in [4]). Third, the *mapping persistence overhead*, i.e., the time taken to write the FTL mapping table into the persistent flash device, is high if we would like fast recovery. Fast recovery requires the FTL mapping table to be made persistent at the commit time of each transaction, leading to high overhead.[3]

Our goal in this paper is to design a transactional SSD (TxSSD) to support flexible isolation levels in the system with atomicity and durability guaranteed *inside* the SSD, while achieving low hardware overhead (in terms of memory consumption, garbage collection and mapping persistence) for tracking transactions' states.

Observations and key ideas. We make two major changes to the Flash Translation Layer to achieve the above goal, resulting in what we call LightTx:

1) While the *no-overwrite* property of SSDs allows different versions of a page to be simultaneously updated in different locations, the FTL mapping table determines the correct order by controlling the sequence of mapping table updates. LightTx supports arbitrary transaction concurrency by allowing pages to be updated independently. A version of the page update is not visible until the mapping entry in the FTL mapping table is updated to point to the new location. LightTx updates the FTL mapping table at transaction commit time (instead of at the time a write happens, as done in conventional systems) to support concurrent updates of the same page. In addition, LightTx tags each write with a transaction identifier (TxID) and determines the committed/uncommitted state of transactions solely inside each transaction, which ensures that states of transactions are determined independently. The commit protocol of LightTx is designed to be *page-independent*, which supports concurrent transaction execution even with updates to the same page (Section 3.2).
2) The *near-log-structured update* characteristic of an SSD makes it possible to track recent transactional writes and retire the dead transactions with low overhead. During page allocation, pages are allocated from each clean block in sequential order. This enables page updates to be performed in a log-structured manner. But as multiple clean blocks from different parallel units (i.e., channels) are used for allocation concurrently, page allocation is performed in multiple heads and each head maintains the log-structured manner of updates. We refer to this as the *near-log-structured update* property of an SSD. Leveraging this property, LightTx tracks those blocks that are used for allocation to track recent writes. In addition, a transaction has birth and death. A transaction is dead when its pages and their mapping entries are updated atomically and made persistent. In this case, committed pages can be accessed through the FTL mapping table while the uncommitted pages cannot. With the consideration of the transaction lifetime, LightTx identifies and retires the dead transactions periodically and only tracks the live ones using a new zone-based scheme. This is in contrast to previous proposals where dead transactions are tracked unnecessarily for a long time until they are erased. LightTx's new *zone-based transaction state tracking scheme* enables low-overhead state identification of transactions (Section 3.3).

Contributions. To our knowledge, this is the first paper that allows a low-cost mechanism in SSDs to support flexible isolation levels in transactions. To enable such a mechanism, this paper makes the following specific contributions:

1) We decouple concurrency control and transaction recovery in single transactions. We manage these two properties respectively in software and hardware, with an extended transactional interface to SSDs. Doing so enables this work to support flexible concurrency controls while leveraging SSD advantages in transaction recovery.
2) We propose LightTx, which uses a novel page-independent commit protocol, to improve transaction concurrency by controlling the update order in the FTL mapping table.
3) We design a new zone-based transaction state tracking scheme for LightTx to only track the live transactions and periodically identify and retire the dead ones. This reduces transaction state tracking cost, making LightTx a lightweight design.
4) We evaluate LightTx using database traces in terms of both performance and overhead (including the overhead of garbage collection, memory consumption and mapping persistence). Results show that LightTx achieves nearly the lowest overhead while maintaining high performance compared to existing embedded transaction designs [4], [5].

2 Background and Related Work

2.1 Flash-Based Solid State Drives

A flash-based SSD is composed of multiple flash packages (chips) connected through different channels. In each chip,

[2] This is because different transactions can update pages in the same block, which is a consequence of the scattering of pages of each transaction to different flash blocks to maximize internal SSD parallelism when executing a transaction.

[3] Otherwise, the whole disk should be scanned, and all pages should be checked for recovery, which is even higher overhead than making the FTL mapping table persistent at each transaction commit.

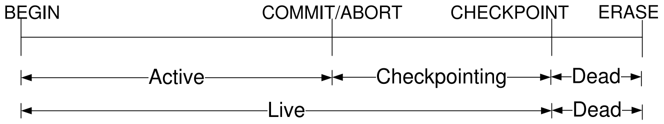


Fig. 1. Transaction lifetime.

there are multiple planes, and each plane has a number of blocks.[4] A block is composed of pages, and a flash page is the read/write unit. A typical flash page size is 4 KB with 128B-page metadata, a.k.a. OOB (Out-of-Band) area, and the block size is 256 KB [8]. In SSDs, read/write requests are distributed to different blocks in different channels and planes in parallel or in a pipelined fashion [9], [10], providing *internal parallelism* in access.

Programming of flash memory is unidirectional. For example, a bit value of 1 can be changed to 0 (via programming), but the reverse directional programming is forbidden (due to the nature of *incremental step pulse programming*, which can only inject charge but cannot remove it from the floating gate [11], [12], [13]). An erase operation is needed before the page can be reprogrammed. To avoid the long latency of an erase operation before a write, the FTL redirects the write to an already-erased page, leaving invalid the original page the write is destined to (this page is to be erased later during garbage collection). This property is known as the *no-overwrite* property; i.e., a page is not overwritten by a write operation. While the no-overwrite property keeps both the old and new page versions and makes it attractive to support transactions embedded inside the SSD, exploiting internal parallelism requires enough concurrent requests to be serviced. Note that concurrency is vital in embedded transaction design to make the best use of the internal SSD performance.

2.2 Transaction Variety and Lifetime

Transactional execution provides 1) consistent state changes for concurrent executions and 2) recovery from system failures. In transaction management, the concurrency control method provides different isolation levels between transactions. The transaction recovery module ensures atomicity and durability. The isolation level determines the parallelism degree of concurrent transactions—for example, *strict isolation* requires the serial execution of transactions. An application chooses the proper isolation level of transactions to make tradeoffs between performance and consistency [7]. Both traditional database management systems and modern data-intensive applications have significantly different transaction isolation requirements [7]. As a result, it is important *to design SSD mechanisms that are flexible enough to enable different isolation level choices at the system level*. In this paper, we aim to provide such flexible isolation level support.

A transaction goes through different periods in its lifetime, which we will exploit in this paper to provide lightweight support for embedded transactions. Generally, a transaction has four operations: *BEGIN*, *COMMIT/ABORT*, *CHECKPOINT* and *ERASE*, as shown in Fig. 1. The *BEGIN* and *COMMIT/ABORT* operations are used by programs to start and complete a transaction. When a transaction is committed or aborted, its state has been determined and can be recovered after system crash, but this requires extra transaction metadata to be kept, e.g., the log data. The *CHECKPOINT* operation writes data to its home location and make the state persistent, so as to make the transactions recoverable without keeping transaction metadata. For instance, the *CHECKPOINT* operation in WAL is the in-place update of the new-version data in a transaction. The *ERASE* operation is an operation that overwrites or erases the transaction data when the data is obsolete. Therefore, a transaction *is born* when it is started and *dies* after the checkpoint operation. The death of a transaction is different from the completion (commit/abort) of the transaction. A transaction is alive when the transaction metadata needs to be kept before the checkpoint. As shown in Fig. 1, we call the transactions that have not committed or aborted *active transactions*, and those that have not yet been checkpointed *live transactions*. In a system that uses write-ahead logging, a transaction completes on commit or abort, and dies after the data is checkpointed in-place from the logs. In this paper, we will exploit the death of transactions to reduce transaction state tracking cost in the FTL.

2.3 Related Work

There has been significant recent research on the architectural evolution of the storage system with flash memory, including interface extensions for intelligent flash management [4], [5], [14], [15], [16] and system optimizations to exploit the flash memory advantages [6], [17], [18], [19], [20]. In this section, we mainly focus on transaction support with flash memory.

Different from atomicity support in HDDs [21], [22], the no-overwrite property of SSDs is leveraged to efficiently keep versions of data for transaction recovery. Atomic-Write (AW) [5] is a typical protocol of this kind. It leverages the log-based FTL and sequentially appends the mappings of transactions to it. The last block in each atomic group is tagged with flag ”1” while leaving the others ”0” to determine boundaries of each group. Atomic-Write requires strict isolation from the system: the system should *not* interleave any two transactions in the log-based FTL. Also, *mapping persistence* is conducted for each transaction commit to provide durability; i.e., the FTL mapping table is written back to the flash device after each transaction commits. Atomic Write FTL [23] takes a similar approach but directly appends pages in the log-blocks sequentially. Transactional Flash File System [24] provides transaction support for file systems in micro-controllers in SSDs, but is designed for NOR flash memory and does not support transactions in DBMSs and other applications.

Recent work [19] has employed a single updating window to track the recently allocated flash blocks for parallel allocation while providing transaction support. Although this approach works well for file systems in which transactions are strictly isolated and serially executed, they are not sufficient for DBMSs and other applications with flexible transaction requirements, i.e., the need for different isolation levels.

[4] In this paper, we refer to the *flash block*, the unit of erase operations in flash memory, simply as the *block*.

TxFlash [4] and Flag Commit [15] extend atomicity to general transactions for different applications including DBMSs. TxFlash [4] proposes two cyclic commit protocols, SCC and BPCC, and links all pages in each transaction in one cyclic list by keeping pointers in the page metadata. The existence of the cyclic list is used to determine the committed/uncommitted state of each transaction. But the pointer-based protocol has two limitations. First, garbage collection should be carefully performed to avoid the ambiguity between the aborted transactions and the partially erased committed transactions, because neither of them has a cyclic list. SCC forces the uncommitted pages to be erased before updating the new version. BPCC uses a backpointer which points to the last committed version. Only when the uncommitted pages between the two committed versions are erased, the page with the backpointer can be erased. Both of them incur extra cost on garbage collection. Second, the dependency between versions of each page prevents concurrent execution of transactions that have accesses to the same page and thus limits the exploitation of internal parallelism. In contrast, LightTx aims to support better transaction concurrency and reduce transaction overhead by avoiding the pointer dependency check and maintenance. Flag Commit [15] tries to reduce the garbage collection overhead associated with the transaction state tracking in SCC and BPCC, and proposes AFC and CFC commit protocols by rewriting the page metadata to reset the pointer. Because MLC NAND flash does not support rewrite operations, AFC and CFC do not apply to MLC NAND flash, which is increasingly widespread as flash technology scales. TxCache [25] employs byte-addressable non-volatile memories (NVMs) as disk cache to reduce the transaction overhead. In contrast, LightTx is designed to function in both SLC and MLC NAND flash memory without requiring byte-addressable NVMs.

MARS [26] also exports flexible interfaces to the software from NVM-based SSDs. Since it is designed for byte-addressable non-volatile memory, MARS uses WAL and requires two copies for each update, one to the log and one to its home location. In contrast, LightTx is designed for flash memory and leverages the page-level no-overwrite property of flash memory to support transactional writes with only one copy.

3 LIGHTTX DESIGN

To support flexible transactions with a lightweight implementation, LightTx, our transactional SSD (TxSSD) design, has the following design components and goals:

- *Page-independent commit protocol*to improve transaction flexibility by allowing arbitrary transaction concurrency.
- *Zone-based transaction state tracking scheme*to lower the cost of LightTx by leveraging the near-log-structured update characteristic of SSDs in combination with the lifetime of transactions.

This section describes the design of LightTx, including its transactional SSD (TxSSD) framework, the commit protocol, the zone-based transaction state tracking scheme, and the recovery mechanism.

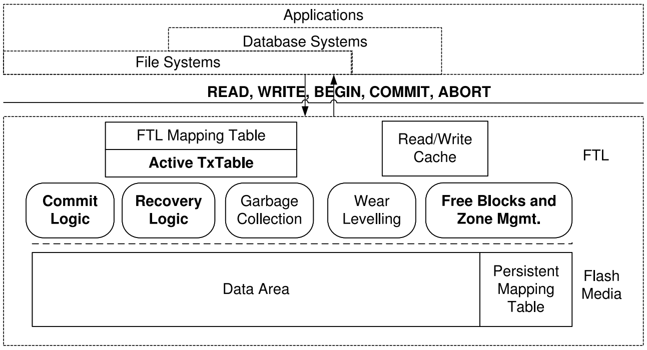


Fig. 2. The TxSSD framework.

3.1 TxSSD Framework

Our TxSSD framework decouples concurrency control and transaction recovery in transaction support. The two functions are separately provided in software and hardware. Hardware transaction recovery can better leverage the no-overwrite property of flash memory for efficient versioning of data, while software concurrency control enables flexible isolation levels.

TxSSD extends the FTL functions in SSDs to support transaction recovery (i.e., atomicity and durability). As shown in Fig. 2, in addition to the modules common to most SSDs (i.e., the FTL mapping table, the read/write cache, garbage collection, and wear leveling), TxSSD introduces three new modules (the Active TxTable, the Commit Logic, and the Recovery Logic) and revises the free block management using a zone-based scheme. The Commit Logic extracts the transaction information from the extended transactional interface shown in Table 1 and tracks the active transactions using the Active TxTable. The Active TxTable keeps the mapping entry from LPN (Logical Page Number) to PPN (Physical Page Number) for each page in active transactions (as shown in the top half of Fig. 4). The Recovery Logic differentiates the committed transactions from the uncommitted and redoes the committed ones during recovery.

Interface design. In order to support the transaction information exchange between the system and the device, we add the BEGIN, COMMIT, and ABORT commands, which are similar to the transaction primitives used in the system, and extend the WRITE command. The BEGIN command checks the availability of a given transaction identifier. If the TxID is currently used, an error is returned to the system, asking the system to allocate a new TxID. For efficiency reasons, the BEGIN command could also be carried with the first WRITE command, instead of being sent explicitly. On a commit or abort operation, the system issues the COMMIT or ABORT command to the device, requesting termination of the transaction of a given TxID. The TxID parameter is also attached to each write operation in the WRITE command. The TxID in each write request identifies the transaction that the page belongs to. The READ command does not need to carry the TxID parameter because isolation between transaction read sets is provided in software, and read requests do not affect the persistent values in storage. Similar to past designs [4], [5], TxSSD only provides transaction recovery function in the SSD for the atomicity and durability of persistent data, and does not aim to provide read isolation.

TABLE 1
The TxSSD Transactional Interface

Operations	Description
READ(*addr*, *len*...)	read data
WRITE(*addr*, *len*, *TxID*...)	write data to the transaction TxID
BEGIN(*TxID*)	check the availability of the TxID and start the transaction
COMMIT(*TxID*)	commit the transaction TxID
ABORT(*TxID*)	abort the transaction TxID

3.2 Page-Independent Commit Protocol

LightTx is designed to support arbitrary transaction concurrency and keep the transaction overhead low. Arbitrary transaction concurrency is supported in LighTx by using a page-independent commit protocol (described next), and low overhead is achieved using a zone-based transaction state tracking scheme discussed in Section 3.3.

Commit protocol. In the commit protocol design, LightTx aims to minimize dependencies between different pages and different versions of the same page. To achieve this goal, LightTx limits the operation of commit logic within each transaction, as each page can be identified to belong to some transaction by storing the transaction identifier in the page metadata, as shown in Fig. 3. Also, LightTx delays the FTL mapping table updates until the commit/abort of each transaction instead of performing updates on a page-by-page basis to reduce conflicts, by introducing the Active TxTable. Transaction versions (TxVer), as opposed to page versions, are used to determine the update sequence of the FTL mapping table. These allow concurrent execution of different transactions even with overlapped updates (i.e., a page accessed by multiple transactions).

In the example shown in Fig. 4, page A is concurrently updated by Tx0, Tx1 and Tx3. The three versions are updated to different flash pages (as shown in the bottom half of Fig. 4), while their mapping entries are kept in the Active TxTable, which keeps track of active transactions and their updated pages (as shown in the top half of Fig. 4). Since the Active TxTable ensures the pages it tracks are not erased, the three versions of page A are kept in flash memory. The use of the Active TxTable allows overlapped pages in different transactions to be updated concurrently.

The commit protocol in LightTx uses page metadata to store transaction metadata. As shown in Fig. 3, the commit protocol uses 12 bytes for transaction metadata, including the transaction identifier, the number of pages in the transaction (TxCnt), and the transaction version (TxVer, the commit sequence), each of 4 bytes. TxID is passed from the system through the interface. TxCnt and TxVer are set to zero for all pages belonging to the transaction except the last-written one, where TxCnt is set to the total number of pages in the transaction and indicates the end of the transaction, and TxVer is set to the latest transaction version to identify the transaction commit sequence and to determine the transaction redo sequence during recovery. To determine the state of a transaction, LightTx counts the number of its pages and checks these pages to see if there exists a page whose TxCnt equals the page count. If so, the transaction is committed. Otherwise, it is not committed.

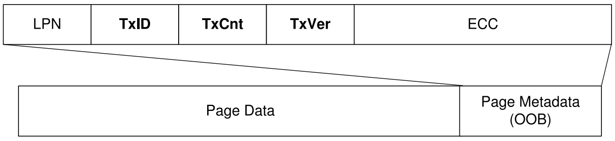


Fig. 3. Transaction metadata in the page metadata.

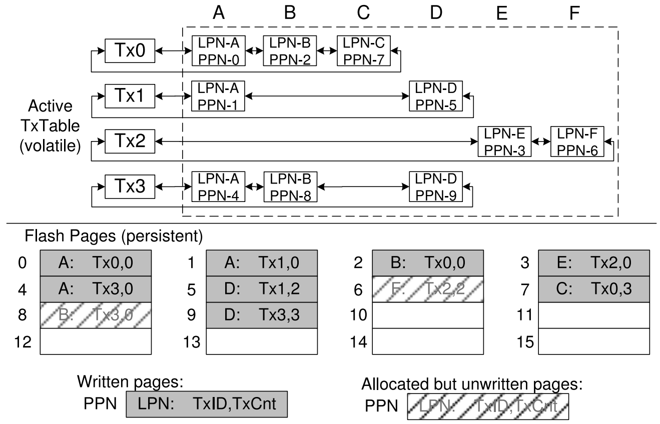


Fig. 4. Illustration of the page-independent commit protocol: Four transactions Tx0 (A, B, C), Tx1 (A, D), Tx2 (E, F) and Tx3 (A, B, D) are executed concurrently. (LPN: Logical Page Number, PPN: Physical Page Number.).

In the example shown in Fig. 4, all pages store the TxID in their page metadata, but only one page in each transaction has non-zero TxCnt (TxVer is not illustrated in this figure). In Tx0, the page at PPN 7 has the TxCnt set to 3. Similarly, pages at PPN 5, 6, 9 have TxCnt set to the number of pages in Tx1, Tx2 and Tx3. During recovery after a system crash or power loss, the TxCnt is read to verify the number of pages that are written successfully in each transaction. In the example, Tx0 and Tx1 are determined to be committed, as each of them has the non-zero TxCnt equal to the number of written pages (indicating all their updated pages are written to persistent flash memory). In contrast, Tx2 and Tx3 are determined to be not-committed. Tx2 is not committed because it has no pages with non-zero TxCnt value. Tx3 is not committed because the number of written pages in persistent flash memory does not equal the non-zero TxCnt value. The mismatch of non-zero TxCnt value indicates not all pages in the transactions are written to persistent flash memory.

Operation. LightTx operations defer the mapping table update to the transaction commit time. Instead of directly updating the FTL mapping table, the write operation updates the mapping entries in the Active TxTable. If the transaction is committed, its mapping entries in the Active TxTable are stored into the mapping table. If not, these mapping entries are simply discarded. For transactional writes, the latest updated page is cached in the SSD DRAM to wait for the next command. If the next command is a commit, the cached page's TxCnt and TxVer are set to non-zero values; if abort, the cached page is discarded; otherwise, the cached page's TxCnt and TxVer are set to zero. For non-transactional writes, the TxID is not set and the TxCnt is set to one, while the TxVer is set to the committed version the same as transactional writes.

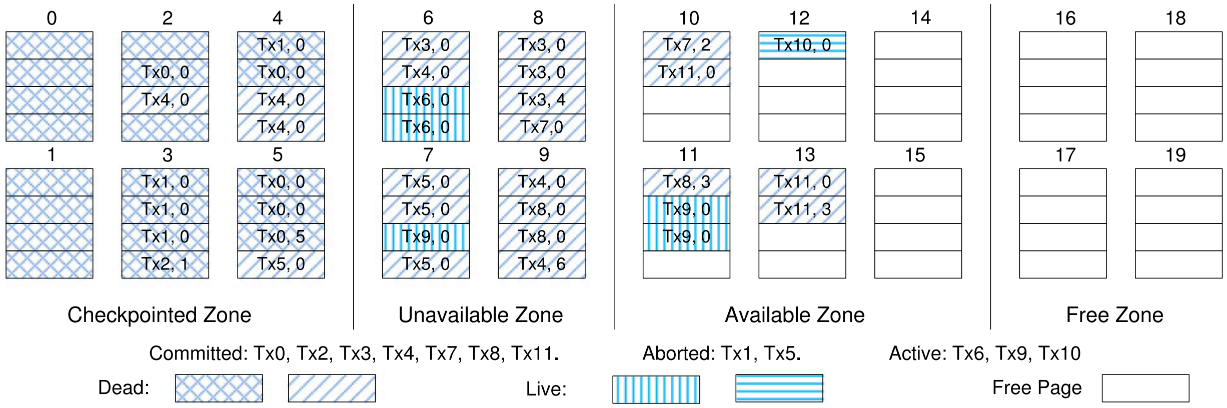


Fig. 5. Zone-based transaction state tracking.

3.3 Zone-Based Transaction State Tracking Scheme

Transaction state tracking identifies the committed and the uncommitted transactions so as to redo the committed and undo the uncommitted during recovery to ensure atomicity. Since pages of each transaction may be scattered across many different flash blocks, state tracking can become costly, if we would like to support flexibility in isolation levels (which requires potentially many transactions' states to be tracked concurrently). To improve efficiency, we use a lightweight design to reduce the tracking overhead in two aspects. First, we track transaction states by tracking the states of flash blocks instead of flash pages. Second, we reduce the number of transactions to be tracked by keeping the live transactions separate from the dead, which is achieved by classifying flash blocks into different *zones* and tracking them separately in these zones (which we discuss next).

3.3.1 Relationship Between Transaction State, Page State and Block State

As shown in Fig. 1, a transaction can be in one of the following three states: *active*, *checkpointing* (i.e., *live but completed*) and *dead*. The state of each flash page is inherited from the transaction it belongs to. So, a flash page can also be in one of the above mentioned three states. Conversely, the state of a transaction can be determined by checking the states of all pages updated by the transaction. But, it is costly to track the state of each page because it requires the tracking metadata to be updated when the page is updated.

Instead of using such costly page-level tracking, LightTx tracks the transaction state at the block-level. LightTx tracks the states of blocks, and then identifies the state of each page to determine the state of a transaction. In traditional flash-based SSDs, flash blocks have three different states: *free*, *available* and *used*. A *free* block has all pages clean. An *available* block is currently used for allocation. A *used* block has all pages written and cannot be used for allocation before it is erased. To differentiate the transactional states of pages, the *used* blocks are further divided into *unavailable* and *checkpointed* blocks in LightTx. All pages in a *checkpointed* block belong to dead transactions. The pages in an *unavailable* block belong to dead, checkpointing or active transactions. States of these pages are further identified using the commit protocol. In this way, the states of transactions can be determined by tracking the states of blocks and identifying the states of pages using the block states.

3.3.2 Block Zones and Zone Sliding

To further reduce the tracking cost, LightTx tracks different flash blocks using different zones. A zone is a logical group that indicates the addresses of its flash blocks.

Block zones. The observation that motivates us to use block zones to reduce the tracking cost is the out-of-place update property of flash-based storage. In flash-based storage, a new write causes an out-of-place update: the data is written to a new physical page in a free flash block (i.e., one that has free pages), and the page containing the old block is invalidated in the FTL mapping table. Since new pages are sequentially allocated from free blocks, it is possible to track the recently updated data by tracking the recently allocated flash blocks. Since transactions have birth and death as shown in Fig. 1, live transactions (see Fig. 1) are more likely to reside in recently allocated flash blocks. We use this observation to keep the states of flash blocks in the FTL in a way that assists the identification of the states of transactions.

A challenge is that one flash block may contain transactions of different states because different transactions may update pages in the same flash block (this is a consequence of designing the system so that we can maximize the internal parallelism of the SSD). As a result of this, there is not a one-to-one correspondence between the state of a block and the state of its transactions. In order to differentiate blocks, we divide blocks into different zones based on each block's state. The flash blocks in our system can be in one of the four states (zones):

- *Free block*, whose pages are all free;
- *Available Block*, whose pages are available for allocation (i.e., available to be written to);
- *Unavailable Block*, whose pages all have been written to but some pages belong to (1) a live transaction or (2) a dead transaction that also has at least one page in an *Available Block*;
- *Checkpointed Block*, whose pages all have been written to and all pages belong to dead transactions.

As shown in Fig. 5, the four kinds of blocks are tracked in four different zones: *Free Zone*, *Available Zone*, *Unavailable Zone*, *Checkpointed Zone*. Conventional FTLs already distinguish between the free, available, and other (i.e., unavailable + checkpointed) blocks. LightTx requires the differentiation of the unavailable and checkpointed blocks, in addition. To enable this differentiation, the addresses of flash blocks in the *Unavailable Zone* are added to a zone metadata page at each checkpoint. The checkpoint operation is periodically performed to move each flash block to its appropriate zone (if the block's zone has changed). We call this *zone sliding*.

Zone sliding. Zone sliding moves those flash blocks that have their states changed to new appropriate zones. Among the four zones, only the *Available* and *Unavailable Zones* have blocks whose states are potentially modified during the interval between two checkpoints. This is because new pages can only be updated in the *Available Zone* and only the *Available* and *Unavailable Zones* have active transactions that can be committed or aborted. Thus, only the states of transactions in these two zones need to be checked during zone sliding.

Zone sliding is triggered when the *Available Zone* runs short of free space. When it is triggered, the flash blocks in the *Available* and *Unavailable Zones* are checked and moved if their states have been changed. Then, free flash blocks are allocated to the *Available Zone*. To reduce the frequency of zone sliding, LightTx preallocates a dedicated number of free blocks from the *Free Zone* to the *Available Zone*. Movement of flash blocks from the *Checkpointed Zone* to the *Free Zone* is performed during garbage collection of the FTL, using conventional garbage collection strategy. As such, with both zone sliding and garbage collection, the flash blocks are moved between the four zones.

Example. Fig. 5 illustrates a snapshot view of an SSD with 20 flash blocks, where each block has four pages. Tag $<\text{Tx}i, j>$ denotes the TxID i and TxCnt j (TxVer is not illustrated in this figure). Using the zone-based tracking scheme, the Available Zone tracks the blocks used for page allocation (blocks 10-15), including the pre-allocated free blocks (blocks 14 and 15). On a checkpointing operation, page mappings in committed transactions are made persistent, and these committed transactions become dead. As a consequence, blocks 2-5 are tracked in the Checkpointed Zone. Blocks 8 and 9, which have pages from Tx7 and Tx8 (which are dead but have pages in the Available Zone), are tracked in the Unavailable Zone as well as blocks 6 and 7, which have active transactions.

During recovery after an unexpected system crash, LightTx follows the steps shown in Section 3.4. The Available Zone is first scanned, and Tx11 is identified as committed since its number of pages matches the nonzero TxCnt according to the commit protocol. Next, the Unavailable Zone is scanned to check the states of Tx7, Tx8, Tx9, and Tx10. Tx7 and Tx8 are committed according to the commit protocol, while Tx9 and Tx10 are not. The other transactions are not checked because their states can be determined by accessing the FTL mapping table (the mapping entries of pages in committed transactions are stored in the FTL mapping table, while those in the aborted transactions are not).

3.3.3 Discussion

With the use of block zones, overheads of mapping persistence and garbage collection are kept low.

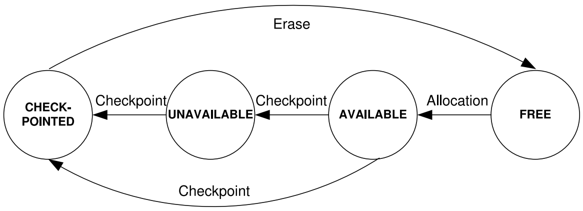


Fig. 6. Block state transition.

Mapping persistence. For durability, the FTL mapping table needs to be written to persistent media when a transaction commits. But this leads to extra flash writes for the FTL mapping table. Instead of persisting the FTL mapping table on each transaction commit, LightTx delays the persistence until the zone sliding.

The delayed mapping persistence does not affect transaction durability. Since mapping persistence are performed on each zone sliding, only the mappings of committed transactions since the last zone sliding are volatile. Fortunately, these transactions can only be active transactions or newly created transactions since last zone sliding, and thus can only be in the *Available* and *Unavailable Zones*. Therefore, even after an unexpected system failure, only the two zones need to be scanned to recover the mappings.

Garbage collection. Once the flash device runs short of free blocks, the garbage collection process is triggered to erase blocks. Only blocks in the *Checkpointed Zone* serve as candidates for garbage collection in LightTx, because the transaction information in both the *Available* and *Unavailable Zones* is protected for transaction state identification during recovery. Since the blocks in *Available* and *Unavailable Zones*, which are used for recent allocation, are unlikely to be chosen for garbage collection, the garbage collection overhead of LightTx is nearly the same as that of conventional FTLs. LightTx's overhead is rather low compared to previous embedded transaction designs [4], [5] because the previous designs have extra restrictions on garbage collection (e.g., forced garbage collection of aborted pages in SCC [4], prevented garbage collection in BPCC [4] on blocks that have experienced aborts since the last committed version).

In addition to garbage collection, LightTx puts few restrictions on wear leveling. For dynamic wear leveling, LightTx has no restrictions. For static wear leveling, no flash blocks in the *Available* and *Unavailable Zones* are selected for wear leveling.

3.4 Recovery

Before discussing the recovery steps, we first present the transitions and properties of block states.

3.4.1 Block State Transition

As discussed above, each block can be in one of the four states (*free*, *available*, *unavailable* and *checkpointed*), and there are three kinds of transition operations (*allocation*, *checkpoint* and *erase*). The block state transition diagram is shown in Fig. 6. The *allocation* and *erase* operations are the same as in traditional SSDs. The only difference between LightTx and traditional SSDs is that LightTx further divides *used* blocks into *unavailable* and *checkpointed* blocks using the *checkpoint* operation. The *checkpoint* operation causes negligible overhead as it only checks the page states and keeps the zone metadata (i.e., the physical addresses of blocks in the zones). As such, the zone-based transaction state tracking scheme is lightweight.

In this scheme, states of pages (and thus transactions) are identified as follows:

- The *free* blocks have no data, and thus no committed or uncommitted pages.
- The pages in the *checkpointed* blocks are committed if they are indexed in the persistent mapping table. Otherwise, they are uncommitted.
- Among pages in the *unavailable* blocks, the pages that are indexed in the persistent mapping table belong to the committed transactions. Pages that are *not* indexed in the persistent mapping table belong to either committed or uncommitted transactions, because some transactions may have pages scattered over the *available* and *unavailable* blocks. Thus, these pages need to be further checked using the commit protocol, with more metadata in the pages of the *available* blocks.
- The states of the transactions in the *available* blocks are identified by comparing the total number of pages in each transaction and the non-zero TxCnt value in some page. If the values match, the pages are committed. Otherwise, the pages from the *unavailable* blocks with the same TxID are read for further check. Only the pages that fail the second check are marked as uncommitted.

3.4.2 Recovery Steps

With the zone-based transaction state tracking scheme, live transactions are separated from dead ones. Only the *Available* and *Unavailable Zones* have live transactions. All transactions in the *Checkpointed Zone* are dead. Therefore, to identify the transaction state, LightTx only needs to read the page metadata in the *Available* and *Unavailable Zones*. Recovery steps are as follows:

1) First, page metadata in the *Available Zone* except those in free pages is read to check the states using the commit protocol. A transaction that has the number of read pages matching its non-zero TxCnt is marked as committed, while other transactions need further identification using page metadata in the *Unavailable Zone*, because some pages in those transactions may reside in the *Unavailable Zone*.
2) Second, page metadata in the *Unavailable Zone* is read to identify the transactions from the *Available Zone*whose states have not been identified. Once the number of read pages matches the non-zero TxCnt, the transaction is marked as committed. The others are uncommitted.
3) Third, committed transactions identified from the above steps are redone to update the FTL mapping table in the sequence of transaction commit version, TxVer. Since their data pages are already persistent, only the metadata need to be updated. Following the sequence of TxVer, the persistent mapping table (see Fig. 2) is updated to reflect the updates of the committed transactions. Afterwards, aborted transactions are simply discarded from the TxTable, and the recovery process ends.

TABLE 2
Evaluated SSD Parameters

Parameter	Default Value
Flash page size	4 KB
Pages per block	64
Planes per package	8
Packages	8
SSD size	32 GB
Garbage collection threshold	5%
Page read latency	0.025 ms
Page write latency	0.200 ms
Block erase latency	1.5 ms

4 Evaluation

In this section, we compare LightTx with existing commit protocols, Atomic-Write [5], SCC/BPCC [4] and Multi-Head Logging (MHL) Commit Protocols (as discussed next), in two aspects: (1) performance benefits from transaction flexibility; and (2) protocol overhead, including the overhead of garbage collection, memory consumption and mapping persistence.

4.1 Experimental Setup

We first present two baseline commit protocols using the multi-head logging technique, which exploit the internal parallelism of SSDs. We then describe our simulator and workload configurations.

Multi-head logging (MHL) commit protocols. The multi-head logging technique keeps multiple logs. Updates to multiple transactions are appended to different logs. Each log clusters all updates of one transaction. This allows multiple transactions to be executed concurrently. MHL-D (MHL with Data Clustering) clusters the data of each transaction to each log. Flash space is divided into multiple logs. Pages in the same transaction are written to consecutive physical addresses in one log. MHL-M (MHL with Metadata Clustering) clusters the metadata (i.e., the mappings) of each transaction to each log. MHL-M is an extension to Atomic-Write [5] with multi-head logging. In MHL-M, multiple transactions are allowed to be executed concurrently, and their mappings are appended to different logs. The MHL technique enables better concurrency and keeps transaction tracking cost low using logs. We compare LightTx to both MHL-D and MHL-M.

Simulator. We implement LightTx on a trace-driven SSD simulator [27] based on DiskSim [28]. The SSD simulator uses a page-level FTL and keeps the whole mapping table in memory. I/O requests are distributed to different packages (elements), providing package-level parallelism. Each package contains multiple planes (parallel units), providing plane-level parallelism. We configure the SSD simulator using the parameters listed in Table 2, which are taken from the Samsung K9F8G08UXM NAND flash datasheet [8]. The SSD simulator is configured using page-level FTL. We use the default garbage collection and wear leveling policies of the SSD simulator [27]. In the evaluation of

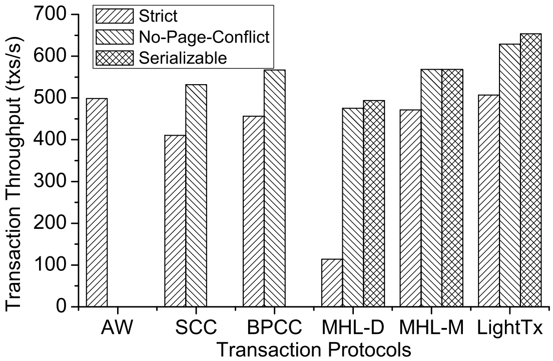


Fig. 7. Effect of different isolation levels on transaction throughput.

MHL-D/MHL-M, the number of log heads are set to 8, which equals the number of packages.

Workloads. We collect the transactional I/O trace from the TPC-C benchmark DBT-2 [29] on PostgreSQL [30] by instrumenting the PostgreSQL source code and recording the XLog operations in the format *(timestamp, TxID, blkno, blkcnt, flags)*. For trace collection, the number of client connections is set to 7 and the number of warehouses is set to 28. The collected trace consists of 1,328,700 requests in total where each transaction updates 27.2 pages on average and 142 pages at maximum. The trace is collected on a 160 GB 7200 rpm hard disk in a server with a 1.60 GHz dual-core processor. In order to saturate the I/O speed of SSDs, the trace is accelerated by 70 times when replayed on the simulator.

4.2 Effect of Transaction Flexibility

To simulate different isolation levels, we use the BARRIER interface to isolate the execution between transactions. We use traces with three isolation levels: strict isolation, no-page-conflict isolation, and serializable isolation. The *strict isolation* trace requires the transactions to be executed serially with a barrier following each commit/abort operation. The *no-page-conflict isolation* trace divides the transactions into segments, and in each segment no two transactions conflict on the same page (i.e., multiple transactions cannot write to the same page). The *serializable isolation* trace allows parallel execution of transactions whose execution sequence is equivalent to a certain serial transaction ordering [1]. All the three isolation levels yield the same functional results.

Fig. 7 shows the transaction throughput obtained with different isolation levels. Atomic-Write supports only strict isolation because it uses a log-structured FTL (as described in Section 2.3). SCC and BPCC do not support serializable isolation because their commit protocols require the states of pages in previous versions to be determined. MHL-D/MHL-M and LightTx support all three isolation levels. Two conclusions are in order from Fig. 7.

1) For a given isolation level, LightTx provides as good or better transaction throughput than all previous approaches, due to its commit protocols better exploitation of internal parallelism. Even though MHL-D and MHL-M can benefit from the multi-head logging technique, the concurrency degree in them are limited by the number of log heads (e.g., 8 in the evaluation).

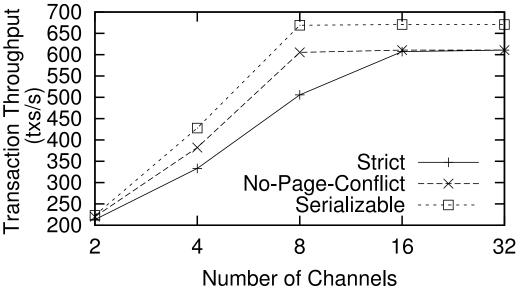


Fig. 8. Sensitivity of internal parallelism to different isolation levels.

2) With LightTx, no-page-conflict isolation and serializable isolation respectively improve throughput by 19.6 and 20.6 percent over strict isolation. This improvement comes from the fact that less strict isolation levels can better exploit the internal SSD parallelism.[5] We conclude that LightTx is effective at enabling the benefits of different isolation levels.

Sensitivity to internal parallelism. To understand the sensitivity to internal parallelism of different isolation levels, we vary the number of channels. Fig. 8 shows transaction throughput of the three isolation levels using different number of channels with LightTx. Generally, transaction throughputs of all the three isolation levels increase or stay the same as the number of channels increases. This means all isolation levels can benefit from better internal parallelism. Second, relaxed isolation levels can better exploit the internal parallelism when the number of channels is small. This reinforces the benefit of flexible isolation levels.

4.3 Garbage Collection Overhead

In this section, we first show the performance of the four protocols under different transaction abort ratios and then analyze the garbage collection overhead.

Transaction throughput. Figs. 9 and 10 show the transaction throughput of all the commit protocols under different abort ratios with strict and no-page-conflict isolation levels, respectively. From the two figures, we make two observations:

1) LightTx and AW have stable performance under different abort ratios, while SCC and BPCC have unstable performance. This is because more aborted transactions incur higher garbage collection overhead in SCC/BPCC, which will be discussed in the following section. In contrast, LightTx and MHL-D simply discard the aborted pages, and AW and MHL-M discard the dirty volatile mapping table on aborts. As a result, these four protocols are not sensitive to the abort ratio.
2) LightTx outperforms all other commit protocols when abort ratio is non-zero and has performance comparable to SCC/BPCC when abort ratio is zero. In Fig. 9, LightTx outperforms AW and MHL-M by 1.5 and 7.4 percent, respectively. The performance cost of AW and MHL-M mainly comes from mapping persistence (write-back of the FTL mapping

[5] Note that transactions are executed serially in strict isolation and as a result the parallelism of the SSD is not fully utilized especially when the transactions are small with few I/Os.

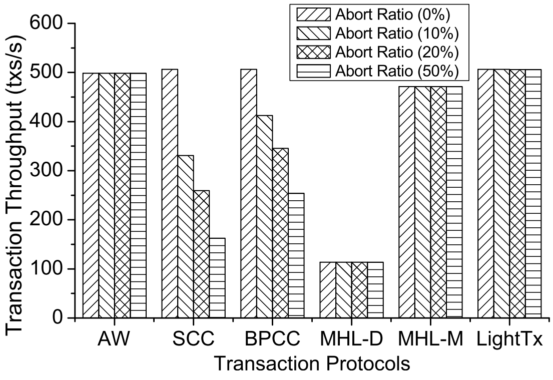


Fig. 9. Transaction throughput under different abort ratios (strict isolation).

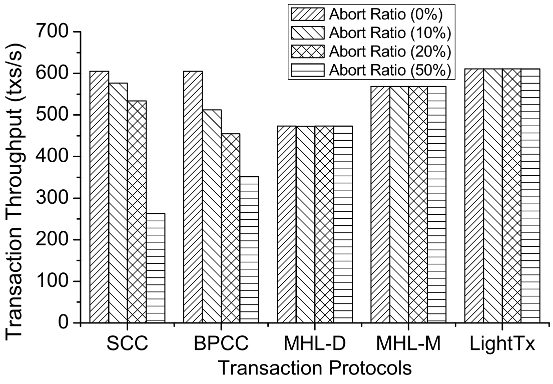


Fig. 10. Transaction throughput under different abort ratios (no-page-conflict isolation).

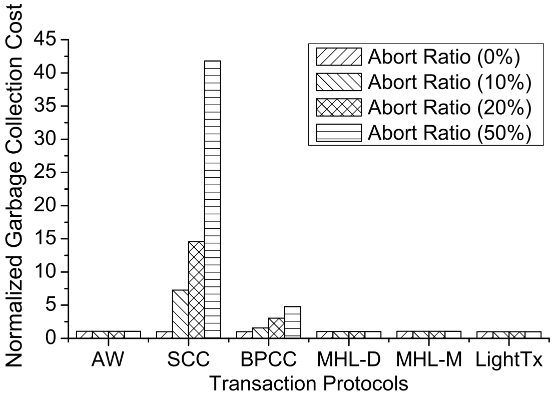


Fig. 11. Garbage collection overhead under different abort ratios (strict isolation).

table into the flash device). MHL-D has poor performance with strict isolation. This is because all pages in one transaction are appended in the same block, but only one transaction is executed at one time in strict isolation, which does not exploit the internal parallelism. The performance overhead of SCC/ BPCC comes from the garbage collection, which will be discussed next. Fig. 10 shows similar results.

Garbage collection overhead. To provide more insight into the performance differences shown in Figs. 9 and 10, we show the normalized garbage collection overhead (in other words, time used for moving the valid pages from each block to be erased and erasing the invalid blocks) in

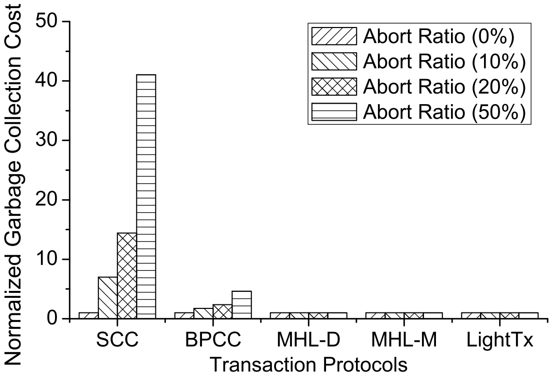


Fig. 12. Garbage collection overhead under different abort ratios (no-page-conflict isolation).

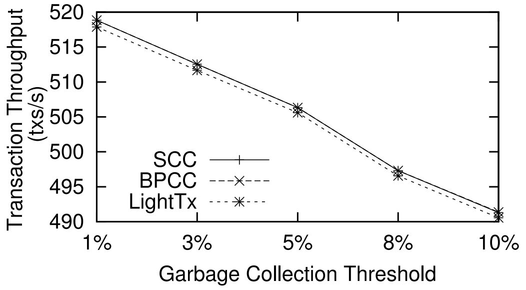


Fig. 13. Transaction throughput versus different garbage collection thresholds (abort ratio = 0%).

Figs. 11 and 12 with different abort ratios. LightTx and MHL-D have the lowest garbage collection overheads among all the protocols. AW and MHL-M have 6.1 percent higher GC overhead than LightTx. This is because the mapping persistence operations needed to make transactions durable in AW and MHL-M are frequent, and the free pages are consumed quickly, which incurs more garbage collection operations. SCC and BPCC have much higher GC overhead than LightTx when abort ratio increases, and the GC overhead in SCC and BPCC is as high as 41.8x and 4.8x of that in LightTx when abort ratio is 50 percent. This is because of the complex pointer maintenance of SCC/ BPCC commit protocols. In SCC, an uncommitted page needs to be erased before the new version is written. This forces a new block that contains an uncommitted page to be erased even when the percentage of valid pages (i.e., pages that are indexed in either the FTL mapping table or the Active TxTable) is high. BPCC employs the idea of straddle responsibility set (SRS), which avoids forced erase operations by attaching the uncommitted pages to some committed page. However, a page whose SRS is not null, even if it is invalid, cannot be erased and has to be moved to the same new location as the valid pages. These constraints incur high GC overhead in SCC/BPCC when the abort ratio is high.

Sensitivity to garbage collection. To further understand the garbage collection cost in LightTx versus SCC/BPCC, we measure transaction throughput and garbage collection time of both LightTx and SCC/BPCC under varied garbage collection thresholds.[6] Figs. 13 and 14 show transaction

[6] Garbage collection threshold is the predefined value of free page percentage, below which the garbage collection process is triggered.

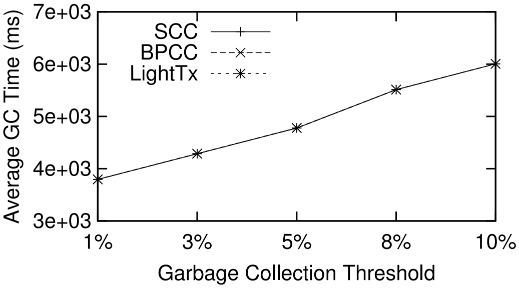


Fig. 14. Garbage collection overhead versus different garbage collection thresholds (abort ratio = 0%).

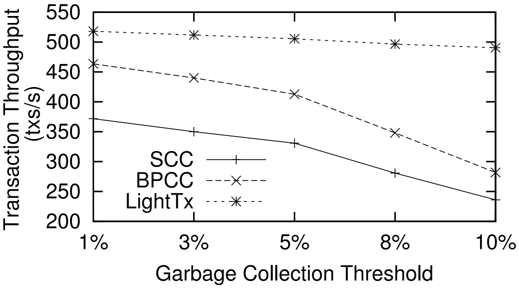


Fig. 15. Transaction throughput versus different garbage collection thresholds (abort ratio = 10%).

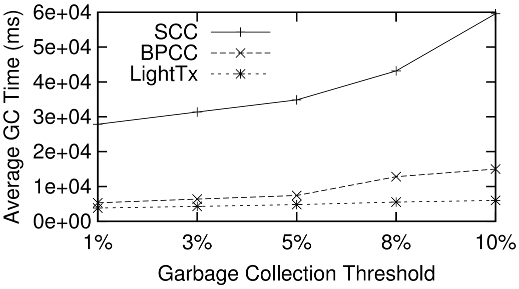


Fig. 16. Garbage collection overhead vs. different garbage collection thresholds (abort ratio = 10%).

throughput and average GC time for LightTx and SCC/BPCC with varied GC thresholds when abort ratio is zero. From the two figures, LightTx has transaction throughput close to that of SCC/BPCC, and the garbage collection time is almost as same as that of SCC/BPCC. Figs. 15 and 16 show the case when abort ratio is 10 percent. In this case, LightTx has better transaction throughput and lower garbage collection time than SCC/BPCC. When GC threshold increases, GC time in SCC/BPCC becomes worse. GC time in LightTx increases very slowly. This is because the restrictions on GC in SCC/BPCC causes inefficiency in garbage collection, either by moving valid pages due to forced erase in SCC or by unnecessary keeping of invalid pages for straddle dependencies in BPCC. The inefficiency causes the free pages to be consumed much faster, and thus leads to poorer GC performance when GC threshold is high. Comparatively, LightTx has few restrictions on GC and is therefore not affected by GC threshold much.

We conclude that LightTx has lower garbage collection overhead than other approaches mainly because 1) it avoids frequent mapping persistence by tracking recent updates in the *Available Zone*, 2) it also avoids extra constraints on garbage collection caused by pointer-based commit protocols in SCC/BPCC, and instead, uses a page-independent commit protocol with a zone-based state tracking scheme.

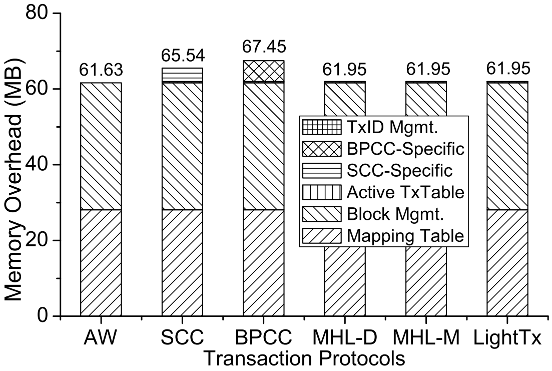


Fig. 17. Memory consumption overhead.

4.4 Memory Consumption Overhead

We measure the FTL memory consumption for main data structures of each protocol and show the composition of the memory overhead in Fig. 17. Except for the FTL mapping table and block management data shared in all protocols, the specific memory breakdown of each protocol varies. SCC and BPCC have higher memory overhead, and the others have low memory overhead. Active TxTable is a shared data structure to keep the metadata of active transactions in all protocols except AW. It consumes 0.31 MB of memory. In addition, SCC keeps the uncommitted pages in memory, which consumes 3.60 MB of memory. BPCC maintains the straddle responsibility set, consuming 5.51 MB of memory. In contrast, MHL-D/MHL-M and LightTx keep only the live transaction IDs, which consumes only several kilobytes.

Specific memory consumption of each protocol becomes more significant when the demand-based FTL (DFTL) technique [31] is used to reduce the memory consumption of the mapping table. DFTL leverages the phenomenon of mapping locality. A small portion of the mapping table can be cached in DRAM with little performance loss while a large portion is kept in the flash media. LightTx and MHL-D can use this kind of mapping table, and SCC and BPCC can also be easily extended with such a page mapping table. In contrast, the log-based FTL used by AW and MHL-M can hardly leverage the mapping locality to reduce the memory usage of mapping table. Block management can be optimized in a similar way. Therefore, we conclude that the memory consumption of previous protocols (AW, SCC and BPCC) and MHL-M can be significant while LightTx and MHL-D achieve low memory overhead.

4.5 Mapping Persistence Overhead

The goal of mapping persistence is to reduce the recovery time. In this section, we will evaluate the recovery time as well as the mapping persistence overhead.

Recovery time. As shown in Fig. 18, the recovery time in SCC/BPCC is 6.957 seconds, while that in LightTx and MHL-D is less than 0.194 seconds for all zone size settings, which is close to the recovery time in AW/MHL-M, 0.126

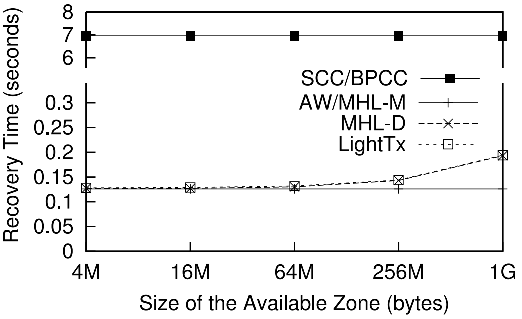


Fig. 18. Recovery time.

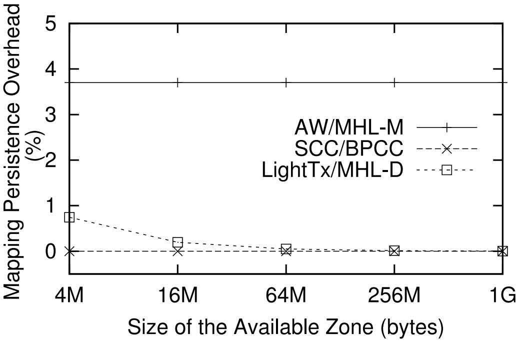


Fig. 19. Mapping persistence overhead.

seconds. The recovery time has two components: page scanning and state checking. As processing in the CPU (required by state checking) is much faster than flash accesses (required by page scanning), the page scanning time dominates the recovery time. AW/MHL-M have the smallest recovery time because the mappings are persistent for each transaction and only the mapping table needs to be read at recovery. Comparatively, SCC and BPCC require a whole device scan because all the pages of the latest version should be found to determine the transaction states [4].[7] However, the need to scan the whole device leads to a large overhead (not to mention it increases linearly with device capacity). LightTx tracks the pages whose mappings have not been persistent in the *Available* and *Unavailable* Zones, so that only pages in the two zones need to be read for recovery in addition to the persistent mapping table read. As a result, the recovery time overhead of LightTx/MHL-D depends on the number of pages in the *Available* and *Unavailable Zones*, which is much smaller than that in the entire device, leading to significantly lower recovery time than SCC and BPCC.

Mapping persistence overhead. We measure the mapping persistence overhead using the metric of mapping persistence write ratio, which is the number of writes to ensure mapping persistence divided by the total number of writes in the trace. Fig. 19 shows that mapping persistence overhead in AW/MHL-M is 3.70 percent, while that in LightTx is less than 0.75 percent for all zone size settings. SCC and BPCC trade recovery time for mapping persistence overhead and have no persistent mapping table, which leads to zero overhead for mapping persistence. AW/MHL-M uses the other extreme approach, which makes mappings persistent for each transaction, leading to the highest overhead. As stated above, LightTx relaxes the mapping persistence of pages by tracking them in zones and requires writes to maintain persistence only during zone sliding that happens at the end of a checkpoint (as described in Section 3.3), leading to low overhead.

TABLE 3
Evaluation Summary of Commit Protocols

Commit Protocol	Flexibility (Section 4.2)	Performance (Section 4.2)	GC Overhead (Section 4.3)	Mem. Overhead (Section 4.4)	Recovery Speed (Section 4.5)	Mapping Persist. Overhead (Section 4.5)
AW	strict	medium	good	✓	✓	×
SCC	NPC	unstable	unstable	×	×	✓
BPCC	NPC	unstable	unstable	×	×	✓
MHL-D	bounded	poor	good	✓	✓	✓
MHL-M	bounded	good	good	✓	✓	×
LightTx	arbitrary	good	good	✓	✓	✓

(1) ✓ - good, × - poor. (2) NPC: no-page-conflict, bounded: bounded by the number of log heads, unstable: unstable under different abort ratios.

Based on these evaluations, we conclude that LightTx achieves fast recovery with low mapping persistence overhead.

4.6 Evaluation Summary

Table 3 shows the evaluation summary of MHL-D, MHL-M and LightTx compared with existing protocols (AW, SCC and BPCC). While MHL-D and MHL-M support better transaction concurrency, they are still bounded by the number of log heads. Also, MHL-D has poor performance because the pages in one transaction need to be consecutively written to the same block, which hurts the internal parallelism. MHL-M has high mapping persistence overhead because it has to make the mapping table persistent on each transaction commit. Comparatively, LightTx supports arbitrary transaction concurrency and achieves the highest performance and nearly the lowest overhead in all evaluated aspects.

5 Conclusion

Providing flexible support for transactions at low overhead is a key challenge in transactional SSD (TxSSD) design. In this paper, we propose a novel TxSSD design, called LightTx, to achieve high transaction flexibility at low overhead. To improve transaction flexibility, LightTx decouples concurrency control and transaction recovery of transactions and manages them respectively in software and hardware, by extending the transactional interface of SSDs. Inside the SSD, LightTx reduces page dependencies to improve concurrency using a page-independent commit protocol. To reduce transactional overhead, LightTx introduces a zone-based transaction state tracking scheme to track the recent updates, so as to reduce the mapping persistence overhead while providing fast recovery. The zone-based scheme also retires the dead transactions to reduce the transaction tracking overhead,

[7] To reduce the scan cost, page metadata in all pages of one block is stored in the last page of the block (called the summary page). This optimization is used in both SCC/BPCC and LightTx/MHL-D.

including the overhead of garbage collection and memory consumption. Evaluations show that LightTx is an effective, efficient and flexible transactional SSD design that achieves high performance at low overhead.

ACKNOWLEDGMENTS

The authors would like to thank Song Jiang (Wayne State), Justin Meza (CMU) and Jishen Zhao (PSU) for their valuable feedback, and Ying Liang (Tsinghua) and Peng Zhu (Tsinghua) for their help with experimental setup. This work is supported by the National Major Project of Scientific Instrument of National Natural Science Foundation of China (Grant No. 61327902), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, Intel Science and Technology Center for Cloud Computing, US National Science Foundation (Grants No. 1212962 and 1320531) and Tsinghua University Initiative Scientific Research Program. This paper is an extended and revised version of [32]. J. Shu is the corresponding author.

REFERENCES

[1] R. Ramakrishnan and J. Gehrke, *Database Management Systems.* New York, NY, USA: Osborne/McGraw-Hill, 2000.

[2] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[3] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 12, pp. 223–242, 1981.

[4] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, pp. 147–160.

[5] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *Proc. 17th IEEE Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 301–311.

[6] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A file system for virtualized flash storage," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, pp. 85–99.

[7] R. Sears and E. Brewer, "Stasis: Flexible transactional storage," in *Proc. 7th Symp. Oper. Syst. Des. Implementation*, 2006, pp. 29–44.

[8] Samsung K9F8G08UXM flash memory datasheet. (2012). [Online]. Available: http://www.datasheetarchive.com/K9F8G08U0M-datasheet.html

[9] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 266–277.

[10] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 96–107.

[11] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, A. Cristal, O. S. Unsal, and K. Mai, "Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime," in *Proc. 30th IEEE Int. Conf. Comput. Des.*, 2012, pp. 94–101.

[12] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling," in *Proc. Conf. Des., Autom. Test Eur.*, 2013, pp. 1285–1290.

[13] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai, "Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation," in *Proc. 31st IEEE Int. Conf. Comput. Des.*, 2013, pp. 123–130.

[14] D. Nellans, M. Zappe, J. Axboe, and D. Flynn, "ptrim ()+ exists (): Exposing new FTL primitives to applications," in *Proc. 2nd Annu. Non-Volatile Memory Workshop*, 2011, pp. 1–2.

[15] S. T. On, J. Xu, B. Choi, H. Hu, and B. He, "Flag commit: Supporting efficient transaction recovery on flash-based DBMSs," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 9, pp. 1624–1639, Sep. 2011.

[16] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based SSDs with nameless writes," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–16.

[17] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2010, pp. 1–11.

[18] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 181–192.

[19] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 257–270.

[20] Y. Lu, J. Shu, and W. Wang, "ReconFS: A reconstructable file system on flash storage," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 75–88.

[21] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, "The logical disk: A new approach to improving file systems," in *Proc. 14th ACM Symp. Oper. Syst. Principles*, 1993, pp. 15–28.

[22] R. Grimm, W. Hsieh, M. Kaashoek, and W. De Jonge, "Atomic recovery units: Failure atomicity for logical disks," in *Proc. 16th Int. Conf. Distrib. Comput. Syst.*, May 1996, pp. 26–36.

[23] S. Park, J. H. Yu, and S. Y. Ohm, "Atomic write FTL for robust flash file system," in *Proc. 9th Int. Symp. Consumer Electron.*, 2005, pp. 155–160.

[24] E. Gal and S. Toledo, "A transactional flash file system for microcontrollers," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 89–104.

[25] Y. Lu, J. Shu, and P. Zhu, "TxCache: Transactional cache using byte-addressable non-volatile memories in SSDs," in *Proc. 3rd IEEE Nonvolatile Memory Syst. Appl. Symp.*, 2014, pp. 1–6.

[26] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction support for next-generation solid-state drives," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 197–212.

[27] 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.

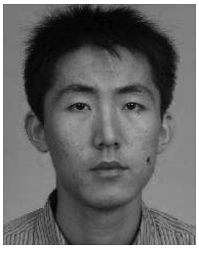
[28] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The *DiskSim* simulation environment version 4.0 reference manual," Carnegie Mellon University, 2008.

[29] Dbt2 test suite. [Online]. Available: http://sourceforge.net/apps/mediawiki/osdldbt, 2012.

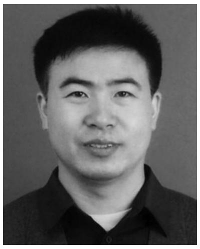
[30] PostgreSQL. (2012). [Online]. Available: http://www.postgresql.org/

[31] 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.

[32] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *Proc. 31st IEEE Int. Conf. Comput. Des.*, 2013, pp. 115–122.

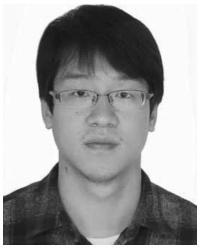


Youyou Lu received the BS degree in computer science from Nanjing University in 2009. He is working towards the PhD degree in the Department of Computer Science and Technology at Tsinghua. His current research interests include non-volatile memories and file systems. He is a student member of the IEEE.



Jiwu Shu received the PhD degree in computer science from Nanjing University in 1998, and finished his postdoctoral position research at Tsinghua University in 2000. Since then, he has been teaching at Tsinghua University. He is a professor in the Department of Computer Science and Technology at Tsinghua University. His current research interests include storage security and reliability, non-volatile memory-based storage systems, and parallel and distributed computing. He is a member of the IEEE.

Jia Guo received the BS degree in computer science from Nanjing University in 2010, and the MS degree in computer science from Tsinghua University in 2013. His research interests include flash-based storage systems.



Shuai Li received the BS and MS degrees in computer science from Jiangsu University in 2010 and 2013, respectively. He was a research assistant at Tsinghua University from 2011 to 2013. His research interests include distributed storage systems.

Onur Mutlu received the BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, and the PhD and MS degrees in ECE from the University of Texas at Austin, in 2006. He is the Dr. William D. and Nancy W. Strecker early career professor at Carnegie Mellon University. His broader research interests are in computer architecture and systems, especially in the interactions between languages, operating systems, compilers, and microarchitecture. Prior to Carnegie Mellon, he worked at Microsoft Research (2006-2009), Intel Corporation, and Advanced Micro Devices. He recently received the IEEE Computer Society Young Computer Architect Award, CMU College of Engineering George Tallman Ladd Research Award, Intel Early Career Faculty Honor Award, IBM Faculty Partnership Award, best paper awards at ASPLOS, VTS and ICCD, and a number of “computer architecture top pick” paper selections by the *IEEE Micro* magazine. For more information, please see his webpage at http://www.ece.cmu.edu/∼omutlu. He is a senior member of the IEEE.