LightTx: A Lightweight Transactional Design in Flash-based SSDs to Support Flexible Transactions*

Youyou Lu †, Jiwu Shu† §, Jia Guo †, Shuai Li † ¶ and Onur Mutlu ‡
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
†Tsinghua National Laboratory for Information Science and Technology, Beijing, China
‡Computer Architecture Laboratory, Carnegie Mellon University, Pittsburgh, PA
luyy09@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn, jguo.tshu@gmail.com, lishuai.ujs@163.com, onur@cmu.edu

***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 has been leveraged to efficiently support transactions, a commonly used mechanism in systems to provide consistency. However, existing transaction designs embedded in flash-based Solid State Drives (SSDs) have limited support for *transaction flexibility*, i.e., support for different isolation levels between transactions, which is essential to enable different systems to make tradeoffs between performance and consistency. Since they provide support for only strict isolation between transactions, existing designs lead to a reduced number of on-the-fly requests and therefore cannot exploit the abundant internal parallelism of an SSD. There are two design challenges that need to be overcome to support flexible transactions: (1) enabling a transaction commit protocol that supports parallel execution of transactions; and (2) efficiently tracking the state of transactions that have pages scattered over different locations due to parallel allocation of pages.**

In this paper, we propose LightTx to address these two challenges. LightTx supports transaction flexibility using a lightweight embedded transaction design. The design of LightTx is based on two key techniques. First, LightTx uses a commit protocol that determines the transaction state solely inside each transaction (as opposed to having dependencies between transactions that complicate state tracking) in order to support parallel transaction execution. Second, LightTx periodically retires the dead transactions to reduce transaction state tracking cost. Experiments show that LightTx provides up to 20.6% performance improvement due to transaction flexibility. LightTx also achieves nearly the lowest overhead in garbage collection and mapping persistence compared to existing embedded transaction designs.

***Keywords*—solid state drive, flash memory, transaction support, atomicity, durability.**

I. 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 [20]. 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) [14] and shadow paging [10] 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 the 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 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. However, two obstacles need to be overcome. First is the lack of atomicity of multi-page updates. A transaction usually updates multiple pages, and the mapping entries of these pages in the FTL mapping table may be distributed to different pages. Failure to atomically update all of the mapping entries breaks the atomicity property of transactions. Second is the narrow interface between the system and the storage device. The simple read/write interface does not communicate transaction semantics. This prevents the device from supporting transactions inside the device. Also, the system is unaware of page versions in SSDs and cannot leverage them for transaction support.

Recent research [17], [19] 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 transaction recovery, and thus nearly doubles system performance due to the elimination of duplicated log writes [17], [19]. 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 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 [21]. 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

§Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).
¶Shuai Li joined this work as a research assistant at Tsinghua University, and he was also a graduate student at Jiangsu University.

* This work is supported by the National Natural Science Foundation of China (Grant No. 60925006, 61232003), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), Shanghai Key Laboratory of Scalable Computing and Systems, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, Huawei Technologies Co., Ltd., Intel Science and Technology Center for Cloud Computing, and Tsinghua University Initiative Scientific Research Program.

978-1-4799-2987-0/13/$31.00 ©2013 IEEE

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 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 can require page-level state tracking, which can be costly.[1] 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 are needed in order not to touch the pages used for the commit protocol (as described in [19]). 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.[2]

Our goal in this paper is to support flexible isolation levels in the system with atomicity and durability guarantees provided inside the SSD, all the while achieving low hardware overhead (in terms of 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. Updates to the same pages can be performed *simultaneously* for concurrent transactions, because updates are written to new locations in flash memory instead of over-writing the original location (This is due to the *out-of-place update* property of SSDs that are designed to ensure an erase operation is not on the critical path of a write). Different versions of the page updates are 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) 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 the state determination of transactions are not dependent on each other (i.e., can be done in parallel). The commit protocol of LightTx is designed to be *page-independent*, which supports concurrent transaction execution even with updates to the same page (Section III-B).

2. Transactions have birth and death. A transaction is dead when its pages and their mapping entries are updated atomically and made persistent, in which case committed pages can be accessed through the FTL mapping table while the uncommitted pages cannot. In order to reduce the transaction state tracking cost, 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 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 III-C).

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:

- We extend the SSD interface with transaction semantics and introduce a page-independent commit protocol to support simultaneous update of multiple versions of a page concurrently written by different transactions. This protocol provides flexible isolation level choices to the system.
- We design a new zone-based transaction state tracking scheme that tracks the live transactions and periodically identifies and retires the dead transactions. This reduces transaction state tracking cost, making our proposal a lightweight design.
- Our evaluation of LightTx using database traces shows a transaction throughput increase of up to 20.6% with relaxed isolation levels compared to strict isolation. The overhead of LightTx is nearly the lowest in both garbage collection and mapping persistence time compared to existing embedded transaction designs [17], [19].

II. Background and Related Work

A. Flash-based Solid State Drives

A flash-based SSD is composed of multiple flash packages (chips) connected through different channels. In each chip, there are multiple planes, and each plane has a number of blocks.[3] A block is composed of pages, and a flash page is the read/write unit. A typical flash page size is 4KB with 128B-page metadata, a.k.a. OOB (Out-of-Band) area, and the block size is 256KB [3]. In SSDs, read/write requests are distributed to different blocks in different channels and planes in parallel or in a pipelined fashion [8], [11], providing *internal parallelism* in access.

Programming of the 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). 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 best make use of the internal SSD performance.

B. 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

[1]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.

[2]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.

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

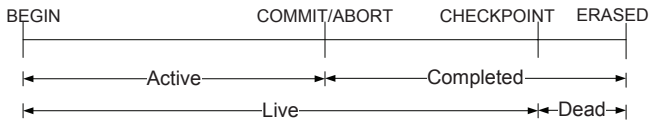

Fig. 1: Transaction Lifetime

control method provides different isolation levels between transactions and determines the consistency level. 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 [21]. Both traditional DBMSs (database management systems) and modern data-intensive applications have significantly varied transaction isolation requirements [21]. 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. The mission of a transaction is to provide a consistent state change. A transaction is born when the system begins the state change and dies after the checkpointing of the system. *Checkpointing* makes the state of the system persistent and unambiguous. That is, the committed pages can be accessed in the FTL mapping table while the uncommitted cannot after checkpointing. Thus, information about dead transactions need not be kept in the SSD. The death of a transaction is different from the completion (commit/abort) of a transaction. A transaction is alive when the transaction state is ambiguous before the checkpointing. As shown in Figure 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.

C. 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 [15], [16], [17], [19], [22] and system optimizations to exploit the flash memory advantages [6], [7], [12], [13]. In this section, we mainly focus on transaction support with flash memory.

Flash memory has been exploited to provide atomicity for file systems. Atomic-Write [17] 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 commit. Atomic Write FTL [18] takes a similar approach but directly appends pages in the log-blocks sequentially. Transactional Flash File System [9] 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 [13] has employed a single updating window to track the recently allocated flash blocks for parallel allocation while providing transaction support. Although these approaches work 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. requirements for different isolation levels.

TxFlash [19] and Flag Commit [16] extend atomicity to general transactions for different applications including DBMSs. TxFlash [19] 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 to straddle the uncommitted, but requires the page with the backpointer to be erased after all the uncommitted pages are 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 tracks only the live transactions and prevents garbage collection in the recently updated blocks, which have low probability to be chosen as victims for erasing, and uses a page-independent commit protocol to support concurrent transaction execution. Flag Commit [16] 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. In contrast, LightTx is designed to function in both SLC and MLC NAND flashes.

III. LightTx Design

To support flexible transactions with a lightweight implementation, LightTx has the following design components and goals:

- *Page-independent commit protocol* to support simultaneous updates of multiple page versions, so as to enable flexible isolation level choices in the system.
- *Zone-based transaction state tracking scheme* to only track the blocks that have live transactions and retire the dead ones to lower the cost of transaction state tracking.

This section describes the design of LightTx, including the interface, the commit protocol, the zone-based transaction state tracking scheme, and the recovery mechanisms.

A. Design Overview

LightTx extends the FTL functions in SSDs to support transaction atomicity and durability. As shown in Figure 2, in addition to the common modules (i.e., the FTL mapping table, the read/write cache, garbage collection, and wear leveling), LightTx introduces three new modules (the active TxTable, commit logic, and recovery logic) and revises the

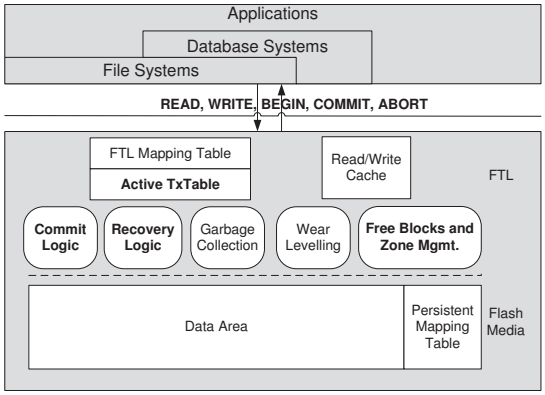

Fig. 2: The LigthTx Architecture

TABLE I: Device Interfaces

Operations	Description
READ(*LBA, len...*)	read data from LBA (logical block address)
WRITE(*LBA, 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

free block management using a zone-based scheme. The commit logic extracts the transaction information from the extended transactional interface shown in Table I and tracks the active transactions using the active TxTable. The active TxTable is a list of active transaction lists, and each active transaction list links the page metadata of all pages of the transaction. Each entry in the active TxTable records the mapping from a page's logical address to its physical address. 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 (TxID). If the TxID is currently used, an error is returned to the system, asking the system to allocate a new TxID. 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 the storage. LightTx only provides transaction recovery function in the SSD for the atomicity and durability of the persistent data, and does not aim to provide read isolation.

B. Page Independent Commit Protocol

Commit Protocol. In the commit protocol design, LightTx aims to minimize dependences 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 (TxID) in the page metadata. Also, LightTx delays the FTL mapping table updates until the commit/abort of each transaction instead of doing updates on a page-by-page basis to reduce conflicts. 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).

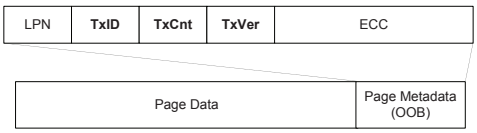

Fig. 3: Page Metadata Format of a Physical Flash Page

As shown in Figure 3, the commit protocol in LightTx uses 12 bytes for transaction information, including the transaction identifier (TxID), 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 TxID, 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.

Operation. LightTx operations defer the mapping table update to the transaction commit time. Instead of directly updating the FLT 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.

C. 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 with classifying flash blocks into different *zones* and tracking them separately in these zones (which we discuss next).

Block Zones. 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

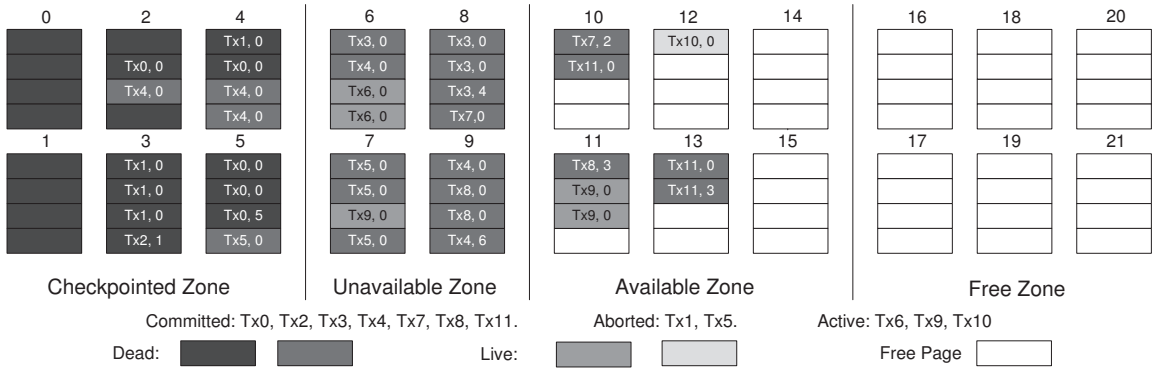

Fig. 4: Zone-based Transaction State Tracking

the page containing the old block is invalidated in the FTL mapping table. This makes it possible to track the recently updated data by tracking the recently allocated flash blocks. Since transactions have birth and death as shown in Figure 1, live transactions (see Figure 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 Figure 4, 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. During the interval between two checkpoints, only the *Available* and *Unavailable Zones* have blocks whose states are potentially modified. 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 after a system crash.

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 because the previous designs have extra restrictions on garbage collection (e.g., forced garbage collection of aborted pages in SCC, prevented garbage collection in BPCC on blocks that have experienced aborts since the last commited version).

Example: *Figure 4 illustrates a snapshot view of an SSD with 22 flash blocks, where each block has four pages. Tag <Txi, 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 III-D. The Available Zone is first scanned, and Tx11 is identified as committed since its number of pages matches the non-zero 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 the mapping entries of pages in committed transactions are stored in the FTL mapping table, while those in the aborted transactions are not).

D. Recovery

With the zone-based transaction state tracking scheme, live transactions are separated from the dead. 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 from the pages in the *Available* and *Unavailable Zones*. Recovery steps are as follows:

1) First, page metadata of all flash pages except the free ones in the *Available Zone* is read to check the states

TABLE II: Evaluated SSD Parameters

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

using 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 of pages 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, the committed transactions identified from the above steps are redone to update the FTL mapping table in the sequence of transaction commit version, TxVer. Since the data pages are already persistent, only the metadata needs to be updated. Following the sequence of TxVer, the persistent mapping table (see Figure 1) is updated to reflect the updates of the committed transactions. Afterwards, the aborted transactions are simply discarded from the TxTable, and the recovery process ends.

IV. Evaluation

In this section, we quantitatively evaluate the benefits of transaction flexibility with different isolation levels and the protocol overhead, including garbage collection overhead and mapping persistence overhead, of LightTx over the existing transaction protocols, Atomic-Write [17] and cyclic commit protocols (SCC/BPCC) [19].

A. Experimental Setup

We implement LightTx on a trace-driven SSD simulator [4] based on DiskSim [5]. We configure the SSD simulator using the parameters listed in Table II, which are taken from the Samsung K9F8G08UXM NAND flash datasheet [3]. We collect the transactional I/O trace from the TPC-C benchmark DBT-2 [1] on PostgreSQL [2] 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.

B. 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 [20]. All the three isolation levels yield the same functional results.

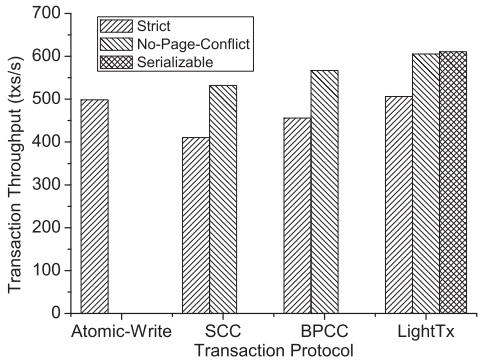

Fig. 5: Effect of Different Isolation Levels on Transaction Throughput

Figure 5 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 II-C). SCC and BPCC do not support serializable isolation because their commit protocols require the states of pages in previous versions to be determined. LightTx supports all three isolation levels. Two conclusions are in order from Figure 5. First, for a given isolation level, LightTx provides as good or better transaction throughput than all previous approaches, due to its commit protocol's better exploitation of internal parallelism. Second, no-page-conflict isolation and serializable isolation respectively improve throughput by 19.6% and 20.6% with LightTx over strict isolation. This improvement comes from the fact that less strict isolation levels can better exploit the internal SSD parallelism.[4] We conclude that LightTx is effective at enabling different isolation levels.

C. Garbage Collection Overhead

In this section, we first show the performance of the four protocols under different abort ratios and then analyze the garbage collection overhead. To provide a fair comparison, we evaluate all the four protocols with the strict isolation level.

Transaction Throughput. Figure 6 compares the transaction throughput of Atomic-Write, SCC, BPCC and LightTx under different abort ratios. In general, LightTx significantly outperforms SCC and BPCC when abort ratio is not zero, while having slightly better performance than Atomic-Write. We make two observations:

1. The abort ratio has no impact on Atomic-Write and LightTx but has a significant impact on SCC and BPCC. This is because more aborted transactions incur higher garbage collection overhead in SCC and BPCC, which will be discussed in the following section. In contrast, LightTx is not sensitive to the abort ratio, as the aborted transactions are simply discarded without any penalty.[5]

2. LightTx outperforms Atomic-Write by 1.6% and is comparable to SCC and BPCC when abort ratio is zero. For the no-abort case, none of the four protocols have

[4]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.

[5]Atomic-Write is also not sensitive to the abort ratio because transactions in Atomic-Write are executed one by one and the identification of transaction state is done independently for each transaction.

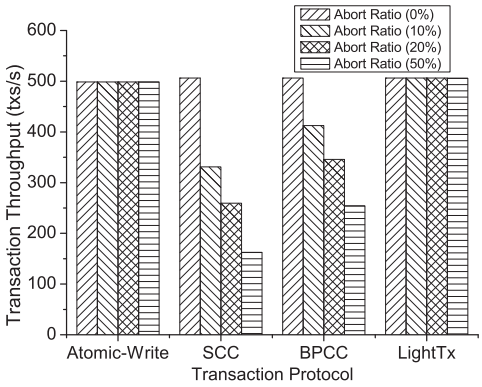

Fig. 6: Transaction Throughput under Different Abort Ratios

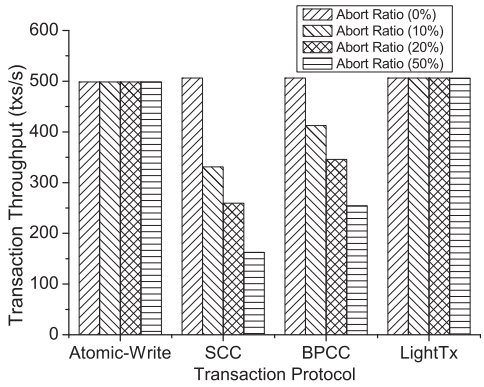

Fig. 7: Garbage Collection Overhead under Different Abort Ratios

extra erase operations caused by aborted transactions. But Atomic-Write and LightTx have more writes due to mapping persistence (write-back of the FTL mapping table into the Flash device), which leads to a higher garbage collection frequency. Comparatively, SCC and BPCC do not keep a persistent mapping table and as a result need to perform additional writes to the flash device. Atomic-Write has to flush the FTL mapping table to the flash memory on each commit operation to make the transaction durable, while LightTx delays the persistence of the FTL mapping table. This is because only mapping entries for pages in the *Available* and *Unavailable Zones* are updated as mentioned in Section III-C, and the FTL mapping table can be updated by scanning the two zones when the system crashes. Fewer writes are incurred due to less frequent mapping persistence updates, leading to a negligible impact on transaction throughput.

Garbage Collection Overhead. To provide more insight into the performance differences shown in Figure 6, we show the normalized garbage collection (GC) overhead (in other words, time used for moving the valid pages from each block to be erased and erasing the invalid blocks) in Figure 7 with different abort ratios. Atomic-Write has 6.1% higher GC overhead than LightTx. This is because the mapping persistence operations needed to make transactions durable in Atomic-Write are frequent, and the free pages are consumed quickly, which incurs more garbage collection operations. For SCC, GC overhead increases proportionally with the abort ratio. When the abort ratio increases, the previous version of a page to be updated has a higher probability to be not committed, and the whole block where the uncommitted page resides has to be erased before the new page is written. Since the forced erase operation has to erase the dedicated block without considering the block utilization, i.e. the percentage of valid pages in the block, garbage collection efficiency is extremely low: the overhead reaches 41.8 times as much as that of LightTx at an abort ratio of 50%. BPCC employs the idea of straddle responsibility set (SRS), which delays the erase of uncommitted pages by attaching the uncommitted pages to some committed page, to avoid a forced erase operation. 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. Such moves cause the overhead of garbage collection to be 4.8 times as much as that of LightTx when the abort ratio is 50%. The main issue that impacts garbage collection in LightTx is that the blocks in the *Available* and *Unavailable Zones* cannot be erased. However, these are also the blocks that are updated the latest and have a high block utilization. As a result, they are seldom selected as the victim blocks during garbage collection, leading to small GC overhead.

We conclude that LightTx has lower garbage collection overhead than the previous approaches mainly because 1) it ensures page versions are independent by using a new page-independent commit protocol, 2) it does not require forced garbage collection, and 3) it keeps block utilization (i.e., the percentage of valid pages in a block, which need to be moved before the block is erased during garbage collection) low.

D. 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 Figure 8, the recovery time in SCC and BPCC is 6.957 seconds, while that in LightTx is less than 0.194 seconds for all zone size settings, which is close to the recovery time in Atomic-Write, 0.126 seconds. The recovery time has two components: page scanning and state checking. During page scanning, the metadata of the pages in the *Available* and *Unavailable Zones* are read. During state checking, the read metadata is processed to determine the state of the pages' corresponding transactions. 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. Atomic-Write has 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 [19]. 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. However, the need to scan the whole device leads to 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 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. Figure 9 shows that mapping persistence overhead in Atomic-

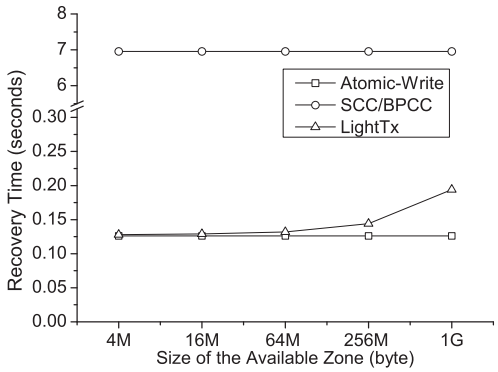

Fig. 8: Recovery Time

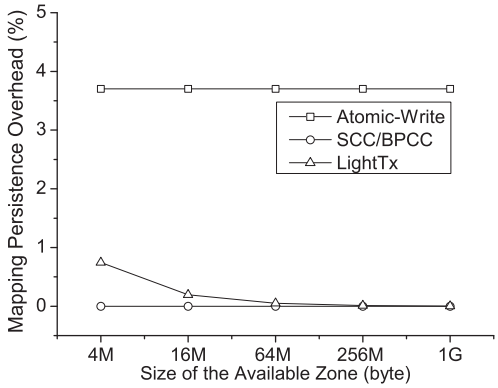

Fig. 9: Mapping Persistence Overhead

Write is 3.70%, while that in LightTx is less than 0.75% 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. Write-Atomic 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 III-C), leading to low overhead.

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

V. Conclusion

Transactional SSDs have leveraged the no-overwrite property of flash memory to free the system from costly transaction management, i.e., duplicated writes in logs. Limited support for transaction flexibility in existing embedded transaction designs not only fail to meet various requirements of emerging applications and DBMSs in the tradeoffs between consistency and performance, but also cannot effectively exploit the internal parallelism present in SSDs. In this paper, we propose LightTx to support flexible isolation levels by increasing the concurrency of writes and reducing the transaction state tracking cost in the Flash Translation Layer. To increase the write concurrency, LightTx extends the system-device interface to enable different transaction semantics and minimizes the dependencies between pages and page versions using a page-independent commit protocol. To reduce the cost of transaction state tracking, LightTx tracks only the live transactions using the new notion of *Available* and *Unavailable Zones* and periodically checkpoints the transactions to identify dead transactions. The tracking of live transactions in the *Available* and *Unavailable Zones* also reduces the frequency of mapping persistence while providing fast recovery. Our evaluations using a real database workload show that LightTx achieves the highest transaction throughput and nearly the lowest Flash Translation Layer operation overhead compared to the state-of-the-art embedded transaction designs, while enabling more isolation levels than past approaches. We conclude that LightTx provides a lightweight and high-performance substrate that enables the system to flexibly make tradeoffs between consistency and performance.

Acknowledgments

The authors would like to thank Song Jiang of Wayne State University, Justin Meza of Carnegie Mellon University and Jishen Zhao of Pennsylvania State University for discussion and feedback. The authors would also like to thank Ying Liang of storage research group in Tsinghua University for her help with the experimental setup.

References

[1] OSDL database test 2. http://sourceforge.net/apps/mediawiki/osdldbt/, 2012.

[2] PostgreSQL. http://www.postgresql.org/, 2012.

[3] Samsung K9F8G08UXM flash memory datasheet. http://www.datasheetarchive.com/K9F8G08U0M-datasheet.html, 2012.

[4] N. Agrawal et al. Design tradeoffs for SSD performance. In *USENIX ATC*, 2008.

[5] J. Bucy et al. The DiskSim simulation environment version 3.0. Technical Report No. CMU-CS-03-102, Carnegie Mellon University, 2003.

[6] A. Caulfield et al. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *SC*, 2010.

[7] F. Chen et al. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS*, 2009.

[8] F. Chen et al. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, 2011.

[9] E. Gal et al. A transactional flash file system for microcontrollers. In *USENIX ATC*, 2005.

[10] J. Gray et al. The recovery manager of the system r database manager. *ACM Computing Surveys*, 1981.

[11] Y. Hu et al. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *ICS*, 2011.

[12] W. K. Josephson et al. DFS: a file system for virtualized flash storage. In *FAST*, 2010.

[13] Y. Lu et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, 2013.

[14] C. Mohan et al. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 1992.

[15] D. Nellans et al. ptrim ()+ exists (): Exposing new FTL primitives to applications. In *NVMW*, 2011.

[16] S. On et al. Flag Commit: Supporting efficient transaction recovery on flash-based DBMSs. *TKDE*, 2011.

[17] X. Ouyang et al. Beyond block I/O: Rethinking traditional storage primitives. In *HPCA*, 2011.

[18] S. Park et al. Atomic write FTL for robust flash file system. In *International Symposium on Consumer Electronics*, 2005.

[19] V. Prabhakaran et al. Transactional flash. In *OSDI*, 2008.

[20] R. Ramakrishnan et al. *Database management systems*. Osborne/McGraw-Hill, 2000.

[21] R. Sears et al. Stasis: flexible transactional storage. In *OSDI*, 2006.

[22] Y. Zhang et al. De-indirection for flash-based ssds with nameless writes. In *FAST*, 2012.