Supporting System Consistency with Differential Transactions in Flash-Based SSDs

Youyou Lu, *Member, IEEE*, Jiwu Shu, *Member, IEEE*, Jia Guo, and Peng Zhu

Abstract—Embedded transaction design inside solid state drives (SSDs) is an attractive way to support system consistency with low overhead due to the no-overwrite property of flash memory. Recent research proposes shadow paging variants to reduce write traffic to SSDs for longer lifetime while leveraging the random performance of SSDs. However, writes in transactions usually are small, and the page-aligned shadow paging protocols still incur high write amplification, which hurts SSD lifetime. In this paper, we propose an embedded transaction protocol, DiffTx, which differentially logs partial page updates in a write-ahead logging way and writes full page updates in a shadow paging way, aiming at low write amplification. DiffTx further improves transaction performance using two techniques. First, by clustering mapping metadata of full page updates with differential data of partial page updates in a single record, DiffTx tracks pages of each transaction at low overhead. Second, DiffTx removes the write ordering on commit by delaying the completeness check of writes, leveraging the clean-state write property of flash memory. Experiments show that DiffTx improves throughput by 25.9 percent and halves write traffic on average compared to TxFlash, a typical embedded transaction protocol for flash memory.

Index Terms—Solid state drive, flash memory, transactional SSD, differential transaction, crash recovery, consistency

1 Introduction

Flash memory is getting widely used in both embedded and enterprise systems in recent years. Different from magnetic disks, flash memory cannot be overwritten [1], [2]. Page updates in a flash-based solid state drive (SSD) are redirected to free pages while the old pages are left for garbage collection. This *no-overwrite* property naturally keeps both old and new data versions inside an SSD, which makes it attractive to design transactions inside SSDs for system consistency.

Traditional transaction recovery protocols, including both write-ahead logging (WAL) [3] and shadow paging [4], incur high overhead to support system consistency. *WAL* (a.k.a *journaling* in file systems) writes the new version of updated data pages to the log area, and then writes back to their home locations to overwrite the old-version data. With the use of a log area, old-version pages are protected from being overwritten before the new-version pages are written successfully. *Shadow paging* writes the updated data pages to new locations and then updates the indexing pointers. Since updated pages are written to newly allocated pages, disk space becomes more fragmented. This leads to more random I/Os, which are costly in hard disk drives (HDDs). Comparatively, although WAL doubles the write traffic, I/Os are performed in a sequential way. As such, WAL is more suitable for HDD-based storage systems.

In SSD-based storage systems, shadow paging becomes more promising than WAL for three reasons. First, the *no-overwrite* property of flash memory naturally keeps the old and new data versions. This favors the shadow paging protocol without explicitly writing new versions. Second, flash-based SSDs have better random I/O performance than HDDs. Random access problem in shadow paging is mitigated. Third, a flash memory cell can endure limited program/erase (P/E) cycles, i.e., the *endurance* problem. WAL doubles the write traffic, which accelerates the wear process, and shortens the lifetime of an SSD. Different from WAL, shadow paging writes only once. Therefore, shadow paging is favored in SSD-based storage systems.

Recent research has proposed different shadow paging variants inside SSDs to provide system consistency [5], [6], [7], [8], [9]. To leverage the no-overwrite property of flash memory, these protocols are designed inside SSDs and can access both the old and new versions of a page. These transaction designs inside SSDs are called *embedded transaction designs*. TxFlash [5] is a typical embedded transaction design, which significantly improves performance of traditional storage systems that uses WAL. The improvement comes from two aspects. First, TxFlash writes the updated pages only once in a shadow paging way, which avoids the duplicated writes as in WAL. Second, TxFlash eliminates the use of commit record, which is used to indicate the success of a transaction commit, and thus removes the *write ordering on commit*. This write ordering is required to make sure all updated pages have been persisted when issuing the commit record [5], [7], [10], [11], [12]. To achieve this goal, TxFlash introduces two new commit protocols, SCC and BPCC, and uses pointers in each flash page to link all pages in a transaction into a cyclic list. Different commit protocols are proposed to make the shadow paging protocol more suitable for flash-based SSDs [6], [7], [9].

• *The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {luyouyou, shujw}@tsinghua.edu.cn, {jguo.tshu, zhupeng1011}@gmail.com.*

Manuscript received 22 Apr. 2014; revised 25 Mar. 2015; accepted 26 Mar. 2015. Date of publication 2 Apr. 2015; date of current version 15 Jan. 2016. Recommended for acceptance by D. Atienza.
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.2015.2419664

However, two problems remain in embedded transaction designs. First, while the properties of flash memory have been exploited for better transaction designs, access patterns from applications have not been well researched in transaction designs. In transaction workloads, a large portion of transactional writes are small, and their write sizes are much smaller than the page size. Metadata operations in file systems update multiple metadata pages, but only a few bytes in each page are updated [13]. Similarly, database applications update records from multiple pages, and each record probably is of several bytes and far less than the page size (e.g., 4 KB) [14]. Thus, we refer *partial page updates* to those page writes that update only a small part in each flash page, while the others are *full page updates*. Differential update techniques [15], [16], [17], [18] by comparing the new and old versions have been used to reduce the write size. But these techniques have not been well incorporated into embedded transaction designs.

Second, it remains a challenge to keep a low transaction overhead when exploiting internal parallelism of SSDs in transaction support. Pages in each transaction need to be clustered (i.e., *page clustering*), so that they can be found for recovery after failures. While the internal parallelism of an SSD tends to distribute writes to different channels or chips, it always incurs high overhead to track these pages for one transaction. Also, the *write ordering* forces to wait for the persistence of all log records before issuing the write of a commit record. This write ordering prevents parallel executions of multiple writes, and thus hurts the internal parallelism of SSDs.

To address the above two problems, we propose a differential transaction design, DiffTx, to differentially log the partial page updates while updating the full pages in a shadow updating way. DiffTx is designed to reduce write traffic and keep transaction overhead low while exploiting the internal parallelism of SSDs. DiffTx achieves the goals from the following three aspects. First, DiffTx logs the differential data of partial page updates instead of the whole pages to reduce write traffic. Due to data locality, differential data can be merged for the home-location writes. Even the differential logging writes data twice, one in the log area and the other in the home location, the total write size is still smaller than that in the shadow paging, which only writes data once. Second, to keep the page clustering overhead low, DiffTx stores the mapping metadata of the full page updates along with the differential data of partial page updates using a log record. Third, DiffTx removes the write ordering on commit by delaying the completeness check on recovery. Because writes are only performed on free pages, pages that are written do not contain obsolete data. And thus, pages either are clean or have new-version data. Incomplete writes can be simply identified by checking error correction code (ECC) of each page. This favors the delayed completeness check.

Our contributions are summarized as follows:

1) We collect transactional write traces from both file system and database workloads, and observe that a large portion of transactional writes are small.
2) We propose an embedded transaction design, DiffTx, to differentially log the partial page updates while updating the full page updates in a shadow updating way. This reduces write traffic and improves transaction performance.
3) We efficiently cluster pages of a transaction by keeping mapping metadata of full page updates along with differential data of partial page updates, and removes write ordering on commit by delaying the completeness check of transaction writes. The two techniques achieve low overhead while exploiting the internal parallelism of SSDs.
4) We evaluate DiffTx using both file system and database workloads. Results show that DiffTx improves throughput by 25.9 percent and halves the write size on average compared to TxFlash [5], a shadow paging variant for flash-based SSDs.

The rest of this paper is organized as follows. Section 2 discusses transaction recovery protocols and reveals the access pattern of transaction workloads. Section 3 describes the DiffTx design, including the interface and component extensions as well as the commit protocol. We evaluate DiffTx in Section 4 and give related work in Section 5. And Section 6 concludes.

2 Background

2.1 Flash Memory

Flash memory is a kind of solid state storage media which provides low read and write latency. Flash memory cells can not be overwritten [1], [2]. Flash pages can only be written after they are erased, which is known as *clean-state update*. A page has two parts: page data (e.g., 4 KB) and page metadata (e.g., 128 B). Page metadata, which is used to keep ECC and other information, is also called out-of-band (OOB) area. Flash memory is read or written in the unit of pages, but is erased in the unit of flash blocks (e.g., 512 KB). To hide the long latency of erase operations, pages are redirected to free pages while the old pages are marked as invalid for later erase. The FTL (Flash Transaction Layer) in an SSD keeps a mapping table for the page write redirection. This *no-overwrite* property in the FTL can be leveraged to support transactions using the shadow paging protocol.

In an SSD, the flash memory is connected to the flash controller through multiple channels. There are multiple chips in each channel, and each chip consists of multiple planes. Reads and writes can be distributed to different planes or channels for parallelism. This *internal parallelism* of an SSD provides high aggregated bandwidth.

In addition, each flash memory cell endures limited program/erase (P/E) cycles. Reliability of a cell is weakened as the P/E cycle approaches the limit. This is known as the *endurance* problem of SSDs.

2.2 Transaction Recovery

Transaction provides ACID (atomicity, consistency, isolation and durability) properties to a series of read/write operations. While isolation is ensured by concurrency control module, atomicity and durability are provided by transaction recovery module. Both concurrency control and transaction recovery modules are needed to ensure system consistency. In this paper, we focus on transaction recovery

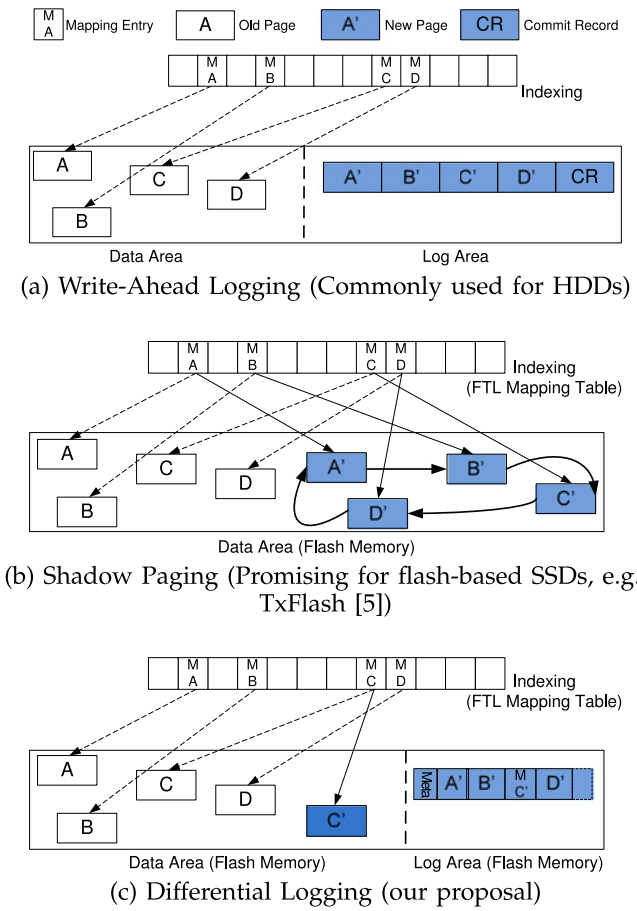


Fig. 1. Transaction recovery protocols.

using flash memory, and assume that concurrency control is performed in programs, similar to [6], [7], [9].

In transaction recovery, there are two main techniques, write-ahead logging [3] and shadow paging [4]. Fig. 1a shows the WAL protocol. In WAL, pages in a transaction are written to the log area followed by a commit record (as shown in the dark boxes). The commit record is not issued until all log writes are persistent. The persistence of the commit record indicates success of the transaction commit. These log pages are checkpointed to their home locations (shown as the light boxes) in the data area, when the log area runs short of space. Since the new-version data will be checkponited to the data area, the indexing pointers, which keep the mapping entries from logical addresses to physical addresses, always point to the data pages in the data area (shown as the dotted lines). Fig. 1b shows the transactional flash (TxFlash) protocol, which is a variant of the shadow paging protocol and is optimized for flash memory. For simplicity, we call this variant as *in-flash shadow paging*. Due to the no-overwrite property of flash memory, updated pages in a transaction are written to free pages (shown as the dark boxes). After this, the mapping entries in the FTL mapping table are updated to point to the new version (from the dotted lines to the solid lines). Transactional writes are updated only once in a shadow updating way, without being written back to their home locations. To cluster pages for each transaction, TxFlash introduces two cyclic commit protocols, SCC and BPCC [5], to link all pages of

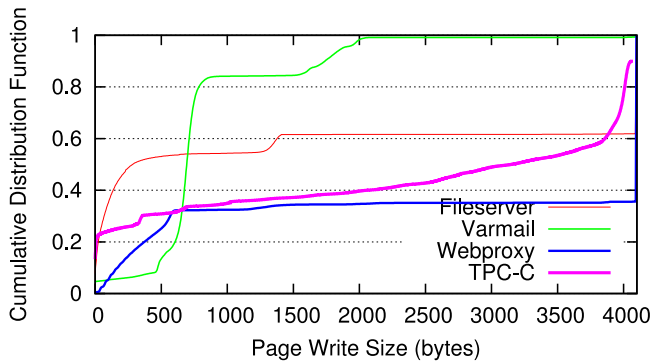


Fig. 2. Cumulative distribution function (CDF): the cumulative frequency of page write sizes.

a transaction in a cyclic list using pointers in the page metadata of each flash page (shown as the solid lines that link all dark boxes into a cycle in Fig. 1b). The two cyclic commit protocols are used for transaction status identification without the use of commit records.

In this paper, we observe that transaction updates are usually small, which will be discussed in Section 2.3. Based on this observation, we propose the *differential logging* technique to log the differential parts of these updates (e.g., page A, B and D) in a WAL-like way as shown in Fig. 1c. We also leverage the no-overwrite property of flash memory for full page updates, e.g., page C. As such, we propose DiffTx, which combines differential logging (for partial page updates) with in-flash shadow paging (for full page updates), to reduce write size and improve performance.

2.3 I/O Pattern of Transactional Writes

Transactions are often used to provide consistency of file system metadata or database operations. These operations usually update multiple pages but update only a small part in each page. For instance, a file create operation creates its own inode, its parent's inode and the directory entry etc. The size of an inode is 256 bytes in ext3, and the size of a directory entry depends on the length of the file name and is usually several bytes. A database operation updates multiple record in different pages. In most cases, the size of a record is only several bytes, which is far less than a page size (e.g., 4 KB).

To study the I/O pattern, we collect I/O traces from both file system (fileserver, varmail, webproxy) and database (TPC-C) workloads, and analysis the write size pattern of all write operations. (See Section 4.1 for detailed setup of trace collections). Fig. 2 plots the cumulative distribution function (CDF) of the update size in each page. In file system workloads, we collect the actual write sizes only for metadata updates while updating data pages fully. From the figure, we have two observations. First, the majority of page updates are either smaller than 1 KB or close to 4 KB. The number of pages that have write size from 1 KB to near 4 KB is comparatively small. Second, a large portion of page writes are small updates. With the two observations, DiffTx could be efficient by combining differential logging for partial page updates with shadow paging for full page updates.

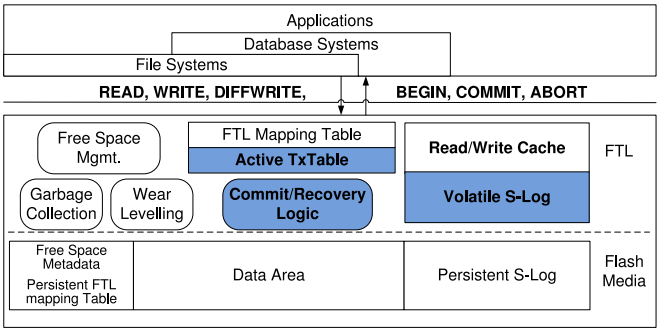


Fig. 3. The framework of transactional SSDs for DiffTx.

TABLE 1
Device Interface in DiffTx

Operation	Description
READ(*LBA, len...*)	read data from LBA (logical block address)
WRITE(*TxID, LBA, len...*)	write data to the transaction TxID
DIFFWRITE(*TxID, LBA, start, len, diff_data*)	write the differential data
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 Design

In this section, we first present the overview of a transactional SSD design, including the interface and FTL extensions for DiffTx. We then describe the DiffTx commit protocol. DiffTx reduces write size by incorporating shadow paging (for full page updates) with write-ahead logging (for partial page updates) for versioning and clustering. And it removes write ordering with delayed completeness check of transactional writes. Finally, we discuss the recovery process after system failures.

3.1 Overview

DiffTx is an embedded commit protocol inside an SSD for transaction recovery. To support this kind of in-device transaction recovery, we extend both device interface and FTL of an SSD. Fig. 3 shows the framework of a transactional SSD (TxSSD) using DiffTx protocol. The DiffTx TxSSD extends the interface with new transactional commands, including BEGIN, COMMIT and ABORT. With these transactional commands, software systems, including user applications, database management systems and operating systems, can pass transactional statuses to storage devices. In addition to these three commands, a DIFFWRITE command is added to pass the differential parts of partial page updates from the software to the device. Descriptions of these commands are listed in Table 1.

The FTL is also extended with new components (shown as colored diagrams in Fig. 3) to process transaction recovery. In addition to conventional components (e.g., FTL mapping table, garbage collection, wear leveling) in a FTL, there are three new components: the Active TxTable, the Commit/Recovery Logic, and the Volatile S-Log. *The Active TxTable* tracks the mapping metadata of updated pages for each unfinished (active) transaction. *The Commit/Recovery Logic* processes the commit or abort operations during normal runnings and performs recovery after system failures. *The Volatile S-Log* is the summary log used for DiffTx to store the mapping metadata of full page updates and the differential data from partial page updates. It serves as the logging area for DiffTx, and it has a persistent copy in flash memory, which we refer to as *the Persistent S-Log*.

3.1.1 Components

Active TxTable. Active TxTable is used to keep the mapping metadata of transactions that have not been committed. Only after the transaction commits, the mapping metadata are updated to the FTL mapping table in order to be made visible to other transactions. The Active TxTable keeps the mapping metadata of all updated pages for each active transaction. The mapping metadata are shown as the META-PU (metadata for page update) in Fig. 4. The META-PU is also appended to the Volatile S-Log for differential logging. For a full page update, the IS-DIFF is unset, and the LPN, VER, BASE-PPN are used. LPN, VER and PPN respectively represent the logical page number, the version number and the physical page number. For a partial page update, the IS-DIFF is set, and the OFF, LEN and DATA-LOC are also used in addition to the LPN, VER and BASE-PPN. The OFF and LEN represent the offset and length of the differential data in the base page, and the DATA-LOC represents the data offset in the log record. The physical page number of a base page may get changed during garbage collection. DiffTx chooses to merge the base page with its differential data during garbage collection, to avoid the update to the BASE-PPN.

To support differential logging in DiffTx, each mapping entry in the FTL mapping table is extended with an obsolete bit flag. The flag is set when the latest data are updated in the S-Log, and thus the read operation needs to read the latest version from the S-Log.

S-Log. S-Log keeps the summary of a transaction, including both the mapping metadata of full page updates and the metadata and data of partial page updates. As shown in Fig. 4, each log record in S-Log has two parts: the S-Log Record Header and the S-Log Record Body. The S-Log Record Header keeps the descriptive metadata of the whole record, such as the transaction identifier (TxID) and the number of updated pages of this transaction (TxLEN), the number of log pages in this record (REC-LEN), in the record head. It also keeps the mapping metadata, META-PU (as discussed in the Active TxTable part). The S-Log Record Body sequentially records the differential data of all the partial page updates in the transaction.

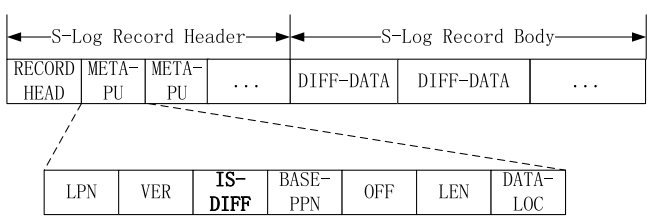


Fig. 4. The S-Log layout.

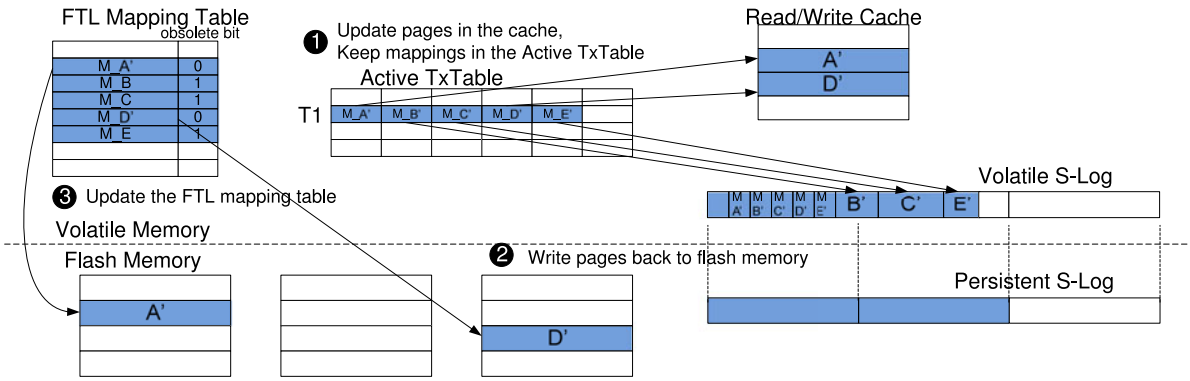


Fig. 5. Update Flow in DiffTx: (1) Transaction writes are first buffered in disk cache, and metadata are temporally kept the Active TxTable. Full page updates are buffered in Read/Write Cache, while partial page updates have their differential data buffered in the Volatile S-Log. (2) After transaction commits, the full pages in the Read/Write Cache or the S-Log are written back to flash memory for transaction durability. (3) Only after the transaction data are persistent, the FTL mapping table are updated to reveal the latest committed pages to succeeding transactions.

3.1.2 Interface and Operations

Differential data can be calculated by comparing differential page update to the previous version of the page. We use DIFFWRITE command for a complementary way. DIFFWRITE passes the differential data directly from the software without the reading of previous versions.

Operations. When a transaction begins, it issues a BEGIN command with a TxID to the SSD. The SSD checks the availability of the TxID. It then initiates an active list in the Active TxTable for tracking the updates in this transaction. For the page writes in this transaction, the data and metadata are respectively written to the Read/Write Cache and the Active TxTable. If the write operation is a full page write using a WRITE command, the page is updated to the Read/Write Cache, and the metadata (LPN, version, PPN) are updated to the Active TxTable. If the write is an partial page write using a DIFFWRITE command, the differential data is updated to the volatile S-Log, and the metadata (LPN, version, PPN, offset, len) is updated to the Active TxTable. When a transaction ends, it either issues a COMMIT or ABORT command. For a COMMIT command, full pages in the Read/Write Cache and differential data in the S-Log are written to the persistent storage. After this, the FTL mapping table is updated using the metadata of this transaction in the Active TxTable.[1] For an ABORT command, the data and metadata from the Read/Write Cache, the S-Log and the Active TxTable of the transaction are all discarded.

Fig. 5 shows the update flow in DiffTx. First, the mapping metadata of write requests, including both full (using WRITE commands) and partial (using DIFFWRITE commands) page updates, are updated in the Active TxTable. Full page updates are written to the Read/Write Cache in the device. For partial page updates, the original write requests are inserted into a temporal request queue, and new log write requests are generated for the differential data. Differential data, as well as mapping metadata for both partial and full page updates, are written to a log record and written to the Volatile S-Log. Second, when a transaction commits, the volatile data pages and S-Log records are written back to flash memory. Third, these mapping metadata are updated to the FTL mapping table. Until the S-Log runs short of space, the original write requests in the temporal request queue are merged, and the home-location write requests are generated.

3.2 DiffTx Commit Protocol

A commit protocol carefully keeps and switches the data versions, so as to be able to determine the commit status of each transaction. A commit protocol has the following three functions: (1) *versioning*: to keep both old and new versions, (2) *clustering*: to cluster all updated pages for each transaction, and (3) *commit identification*: to determine the status of a transaction during recovery.

3.2.1 Differential Logging

Versioning. Transaction atomicity requires that all or none pages of a transaction are updated, i.e., pages are all switched to the new version or all kept in the old version. Both new and old versions need to be kept in case of falling back when system fails. In shadow paging (e.g., TxFlash [5]), page updates are written to new locations followed by updating the mapping entries. In write-ahead logging (e.g., journaling in ext3 [19]), page updates are written to the log area followed by being written back to their home location.

DiffTx combines the two protocols. Full page updates are updated in the shadow paging way by leveraging the no-overwrite property of flash memory. Because of the no-overwrite property of flash memory, page updates are redirected to free pages. By keeping these mappings in the Active TxTable rather than in the FTL mapping table, both versions are accessible. Partial page updates are updated in the write-ahead logging way. The dirty parts in each partial page update are written to the S-Log, as shown in Fig. 4. The new version can be read by merging the differential data in the log with the old version. In this way, DiffTx combines the shadow paging with the write-ahead logging to provide versioning.

Clustering. A commit protocol needs to be able to find all pages of each transaction for undo or redo operations, and this requires page clustering of each transaction. WAL

1. Note that ordering here is different from commit ordering in transactions. Commit ordering in transactions requires commit record written after persistence of all log pages. Commit ordering lies between two persistence operations and thus prevents succeeding flash write operations. In contrast, ordering here does not prevents persistence of succeeding writes, as the FTL updates are memory operations.

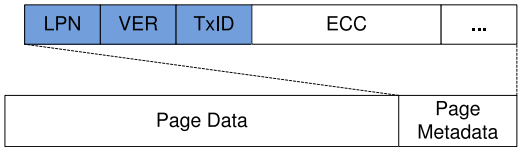


Fig. 6. The page metadata layout.

clusters pages by sequentially appending them in the log. However, full page updates in flash memory are written to different parallel units due to internal parallelism of SSDs. TxFlash uses pointers to link all pages in each transaction to be a cycle. But the erase operation breaks the cycle, which incurs high overhead to maintain the cyclic property as discussed in TxFlash (SCC/BPCC) protocol [5].

DiffTx clusters all pages by putting the metadata of full page updates together with the differential data in the S-Log. As shown in Fig. 4, mapping metadata of all updated pages are sequentially appended in the S-Log Record Header. Therefore, all pages of a transaction can be found by reading the mapping metadata of the log record.

Commit identification. During recovery after system crashes, the commit protocol checks the completeness of all updates for each transaction, so as to identify the commit status of the transaction. In WAL, the completeness check is to check the availability of the commit record. Because the commit record is issued only after all pages have been written back to the storage persistently, this write ordering guarantees that all updates have been persisted if the commit record is persistent. To maintain the ordering, I/O operations halt until the persistence of all updates in the log, leading to high performance penalty [5], [7], [10], [11], [12].

DiffTx removes this ordering during normal execution and checks the completeness of both the log record and the full page updates during recovery. Only when both of them are complete, the transaction is determined to be committed. The completeness check of the log record is performed by checking the availability of its log pages (i.e., flash pages that store log records). A log record may have one or more log pages, and these pages are appended sequentially in the S-Log. Since page updates can only be written to free pages, pages with non-zero values in the log are valid log pages. A log record is complete if the number of valid pages matches the REC-LEN value (i.e., the number of log pages in current log record) stored in the S-Log Record Header.

The completeness check of log records is based on two observations. First, atomicity of a flash page can be checked using the ECC stored in its page metadata (as shown in Fig. 6). Because writes can only be performed in free pages, log pages contain either all zeros or new-version data, but not obsolete data versions. Therefore, the new-version data are complete, if the page has non-zero values and the page passes ECC checking. Second, once the log record is completely written, the transaction metadata it contains are complete. This is because the S-Log log record is persisted after transaction commits. At the time when transaction commits, the transaction metadata has already been appended completely in the log record. Therefore, the completeness of log record writes ensures the completeness of their transaction metadata. With the two observations, we can confirm the transaction metadata completeness in each log record by checking the completeness of each log page using ECC and the completeness of the log record using REC-LEN value in its log header.

The completeness check of the full pages is performed by iterating the META-PU of full page updates in the log record and checking the availability of each of them. The completeness of each page is checked using ECC in the page metadata. In the page metadata, we add new fields (LPN, VER, TxID) to identify the transaction belongings of each page. By checking both the availability and the transaction belongings, the completeness check of full page updates in each transaction is done. As such, the completeness of all updated pages in each transaction is checked to identify the commit status.

While the completeness check of both the log record and the full page updates can remove the write ordering before commit operation, it requires random reads of full page updates and hurts the recovery performance. In order to mitigate the recovery performance penalty in DiffTx, a log record can carry a flag to indicate the completeness of previous transactions during normal execution. The completeness flag is the transaction ID that is stored in the S-Log Record Header. For instance, Tx_m has written back its log record and full pages when Tx_n writes its log record. Tx_n keeps the ID m (i.e., the metadata of completeness of Tx_m) as the completeness flag in the S-Log Record Header of Tx_n's log record. Therefore, Tx_m is complete if its completeness flag in Tx_n is found, and its completeness check of full page updates does not need to be performed.

If either the log record or the full pages are not complete, the transaction is not-committed. During normal execution, the aborted transaction simply discard the log record. Because of the absence of the log record, the transaction is identified to be not-committed during recovery. If some full pages have been persisted in an aborted transaction, these pages are not indexed in the FTL mapping table, so that they are invalid and are erased by later garbage collection process.

3.2.2 Checkpoint and Merge

A read operation needs to read and merge both the base page and the differential data. In addition to read operations, a checkpoint operation merges the differential data with their base pages and writes them back to flash memory. A checkpoint operation is performed when the S-Log runs out of space. One page may have different parts or versions in the S-Log. To avoid multiple flash memory accesses for merging one page, the S-Log is mapped to DRAM memory as the Volatile S-Log. All differential data are accessed in DRAM memory. Since DRAM memory accesses are much faster than flash memory accesses, this does not slow read operations.

All differential data in the S-Log need to be read and merged with the base page, if there is no full page update of this logical page since last checkpoint. Otherwise, the base page is updated to the latest full page update, and only the differential data that are written later than the full page update are read while the previous versions are discarded.

3.2.3 Other Issues

Garbage collection in DiffTx has two restrictions. One is that garbage collection is performed for S-Log, i.e., pages in the S-Log are not erased before they are checkpointed. The

other is that only pages indexed in neither the FTL mapping table nor the Active TxTable are eligible to be erased. With this restriction, both new and old versions are kept. Wear leveling is performed as normal. The S-Log area is allocated from different physical locations, and the old physical pages are rated for wear leveling.

FTL mapping table needs frequent persistence for durability in case of system failures. In DiffTx, the mapping metadata have been kept in the log record for transactional writes. Since the log record is written back to flash memory on transaction commits, the mapping metadata are persistent for committed transactions. For non-transactional writes, DiffTx also provides durability for their mapping metadata by appending the mapping metadata to the S-Log.

3.3 Recovery

After system crashes, volatile data in the FTL, including those in the FTL mapping table, the Active TxTable, the Read/Write Cache and the Volatile S-Log, get lost.[2] Data and metadata have persistent copies in flash memory, because volatile data and metadata of a transaction are persisted on the commit operation (as discussed in Section 3.1.2). Therefore, recovery process is performed to identify the committed transaction using persistent data in flash memory, and then to redo committed transactions and undo not-committed transactions. In DiffTx, recovery steps are as follows:

1) First, the Persistent S-Log is scanned to check the completeness of each log record. Since the log records are appended in the S-Log, only the last one can be incomplete. If it is incomplete, the transaction is identified as not-committed.
2) Second, for transactions that do not have a completeness flag in the S-Log, the completeness of its full page updates needs to be checked. If not all of these full pages are updated, the transaction is incomplete and is identified as not-committed. Transactions that pass the check of above two steps are committed transactions.
3) Finally, the recovery process redoes the committed transactions and discards the not-committed ones. For committed transactions, the differential data are merged with their base pages, and all mappings of their page updates are updated to the FTL mapping table. For not-committed transactions, the log records, including the differential data, are discarded. Full pages that have been written in not-committed transactions are untouched and will be erased by later garbage collection process.

In not-committed transactions, since the log records are appended in the S-Log, these log records are discarded without affecting other transactions. Full pages that have been written are left untouched, and they do not affect other pages, page versions or other transactions. This is because they are treated as invalid pages in SSDs, as they are not indexed in either the FTL mapping table or the Active TxTable. The mappings are kept in the Active TxTable instead of in the FTL mapping table before the writeback of the log record. Since the Active TxTable is volatile and gets lost after system failures, these mappings are lost. Therefore, the full page updates are identified as invalid pages and will be collected as garbage.

With above steps, transactions are identified as either committed or not-committed. And the committed transactions are redone while the not-committed ones are discarded to recovery the SSD to a consistent state.

2. Device memory is volatile except in high-end SSDs, which use capacitors or batteries.

4 Evaluation

We evaluate DiffTx against previous transaction protocols, WAL, TIPL and TxFlash (SCC/BPCC), aiming to answer the following questions:

1) How does DiffTx perform compared to previous approaches in terms of performance (Section 4.2) and endurance (Section 4.3)?
2) What is the overhead of checkpoint (Section 4.4) and recovery (Section 4.5)?
3) What are the impacts from changes in transaction abort ratio (Section 4.6) and the log size (Section 4.7)?

In this section, we first describe the experimental setup before answering the above three questions.

4.1 Experimental Setup

We evaluate DiffTx against previous transaction protocols: WAL [3], TIPL [16] and TxFlash [5]. WAL is a traditional protocol which is widely used in DBMSs and file systems. TIPL uses redundant logging, in-page logging (IPL) and system logging. TIPL allocates one log sector for each page, and all differential updates are appended in the log sector before being merged to the page. This is known as in-page logging. TIPL also appends the differential updates to the system log to support transactions. TxFlash is a variant of shadow paging that is optimized for flash memory to leverage the no-overwrite property of SSDs.

We use both file system and database workloads for evaluation. We revise ext3 file system and PostgreSQL database management system to collect transaction traces. We then replay these traces on our transactional SSD (TxSSD) simulator, which extends a trace-driven SSD simulator [1] with transactional protocols based on DiskSim [20].

TxSSD simulator. TxSSD extends the interface and FTL components in SSD simulator to support embedded transaction protocols. To support DiffTx, TxSSD is extended with the interface as shown in Table 1 and the components as shown in Fig. 3. In the evaluation, TxSSD simulator is configured using the parameters listed in Table 2 following Samsung K9F8G08UXM NAND flash datasheet [21]. Compared to conventional SSDs, the extended components in DiffTx SSD consumes extra memory space. The Active TxTable consumes about 0.3MB. Memory consumption of the Volatile S-Log is fixed, and the default S-Log size in DiffTx is set to 32 MB. In DiffTx, pages that have write sizes smaller than 512 B are set to partial page updates, while the others are full page updates.

WAL, TIPL and TxFlash protocols are also implemented in TxSSD. They share the transactional interface with DiffTx

TABLE 2
Parameters of TxSSD Simulator

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 percent
Page read latency	0.025 ms
Page write latency	0.200 ms
Block erase latency	1.5 ms

and have slightly differences in FTL component extensions. In WAL, the log area is also mapped to the volatile memory. Records are only persisted on commit. WAL simulates two kinds of journaling: data journaling and metadata journaling. To simulate data journaling, WAL logs both data and metadata of file systems in the log area. This is called WAL(D) in this paper. To simulate metadata journaling, WAL logs only metadata of file systems in the log area. This is called WAL (M) in this paper. The default log size is also set to 32 MB. In TIPL evaluation, we allocate one 512 B log sector (i.e., in-page log) for each 4 KB flash page. Every eight flash pages share one flash page to store the in-page log. If one of the log sector is full, the whole page that contains the eight log sectors is written to flash memory. TIPL also allocates 32 MB for the system log by default. Once the system log runs short of space, a checkpoint is performed to persist all dirty pages and log sectors. In TxFlash evaluation, we implement both SCC and BPCC protocols. Page metadata of each page is extended with pointers to support the two cyclic commit protocols.

Transaction traces. In both file systems and DBMSs, group commit technique is widely used to reduce persistence overhead by grouping multiple transactions [19], [22], [23]. Disk I/O traces are more accurate than transaction I/Os. Thus, we collect disk I/Os at block driver level rather than transaction I/Os at memory level. For file system traces, we revise the journaling module (JBD) in ext3 to collect the I/O traces on JBD commit. For database traces, we collect I/Os in the PostgreSQL using IO-Trace tool [17], [24], which records the differential parts of each I/O operation.

File system traces (fileserver, varmail, webproxy) are collected by running filebench [25] on ext3 file system in Kernel 3.10.11. The JBD commit time interval is set to 1 second, and we collect 20-minute trace for each workload. Database traces (TPC-C) are collected using Hammerora 2.11 [26] on PostgreSQL 9.3.1 [27] using default configurations. The trace sizes are listed in Table 3, and the I/O patterns are shown in Fig. 2.

4.2 Performance

We measure transaction throughputs and commit latencies of different workloads using WAL, TIPL, TxFlash[3] and DiffTx protocols to evaluate the protocol performance. In WAL evaluation, we evaluate both metadata and data journaling for file system workloads. Metadata journaling only logs the metadata while writing data to their home locations, e.g., ordered mode in ext3. Data journaling logs both metadata and data, e.g., data mode in ext3. These two types of journaling are referred to WAL(M) and WAL(D) in the following part. Because database management systems do not have metadata journaling, WAL(D) and WAL(M) are not differentiated for TPC-C workload and are referred to WAL in this paper.

3. When abort ratio is zero, both SCC and BPCC protocols in TxFlash have similar performance. Thus, we do not differentiate them when abort ratio is zero.

TABLE 3
Evaluation Workloads

Workloads	tot. # of pages	tot. # of Txs	avg. pages per Tx
Fileserver	319,859	52	6151.1
Varmail	1,051,628	60,227	17.5
Webproxy	1,295,414	1,045	1239.6
TPC-C	1,403,278	78,129	18.0

Transaction throughput. Fig. 7 shows the throughput of each workload using WAL, TIPL, TxFlash and DiffTx transaction protocols. On average, DiffTx outperforms WAL(D), WAL(M), TIPL and TxFlash by 42.0, 32.8, 31.5 and 25.9 percent, respectively. The benefit mainly comes from the reduced write traffic. WAL(M) has better performance than WAL(D), because it avoids data updates to the log area. TxFlash avoids both data and metadata updates to the log area, and has even better performance than WAL(M). TIPL has better performance than WAL(D), as it reduces write traffic by only logging the differential data. But TIPL still need to write two logs, the in-page log and the system log. Comparatively, DiffTx writes only one log for the partial page updates, and writes full page updates in a show paging way. DiffTx further reduces write traffic and achieves the best performance of all evaluated protocols.

In the evaluation, workload varmail has slight different performance in WAL compared to other workloads. In varmail, WAL has similar write traffic with TxFlash. This is because writes have good locality in varmail, and are coalesced well when written back from the log area to the data area. The reason why TxFlash has poorer performance than WAL is that WAL writes in a more sequential way than TxFlash. WAL allocates a continuous space, which can distribute writes to different units and better exploit the internal parallelism of SSDs. In contrast, TxFlash tries to allocate physical pages for each logical page in the same channel, and channel workloads are not always balanced.

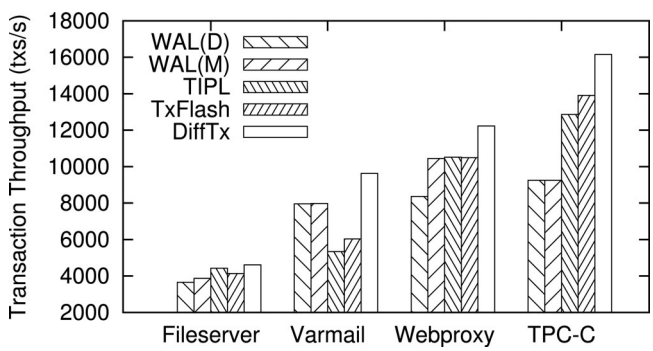


Fig. 7. Transaction throughput of different protocols.

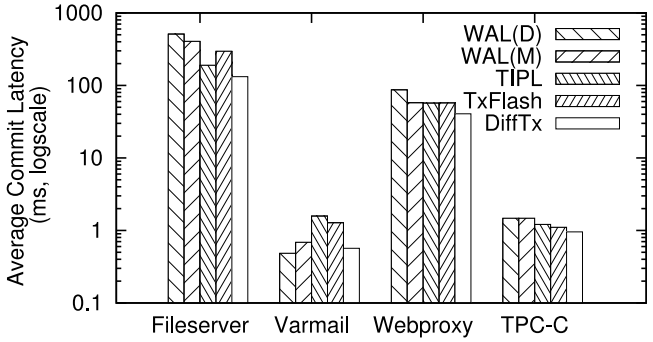


Fig. 8. Commit latency of different protocols.

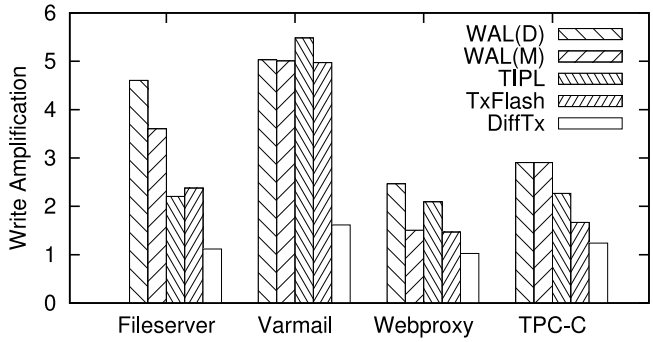


Fig. 10. Write traffic of different protocols.

In general, transaction performance is improved as write traffic is reduced. DiffTx achieves the best performance of all evaluated protocols.

Commit latency. Fig. 8 shows averaged commit latencies (in log scale) of the evaluated workloads in different protocols. DiffTx has the least commit latencies in all evaluated protocols. It dramatically reduces commit latencies for workloads with large transaction size, such as fileserver and webproxy. It reduces the commit latencies by 74.2, 67.6, 30.7 and 55.6 percent in fileserver workload and by 53.1, 29.3, 28.7 and 29.1 percent in webproxy workload compared to WAL(D), WAL(M), TIPL and TxFlash, respectively. The benefit in DiffTx comes from two aspects: the removed write ordering on commit and the reduced write traffic. DiffTx removes the commit record and thus does not need to halt I/Os during transaction commits. Also, as fewer pages are required to be persisted to flash memory, the commit latency is reduced. In all, DiffTx achieves lower commit latency than other protocols.

Fig. 9 shows the frequency distribution of commit latencies of TPC-C workload in different protocols, to evaluate the latency consistency. The other workloads show similar results and are omitted due to space limitation. From the figure, we can observe that DiffTx shows small variance in commit latency. Most transactions have commit latencies smaller than 2 milliseconds. In comparision, the other procotols have transactions in which commit latencies are between 2 to 4 milliseconds. Therefore, we conclude DiffTx has consistent commit latencies.

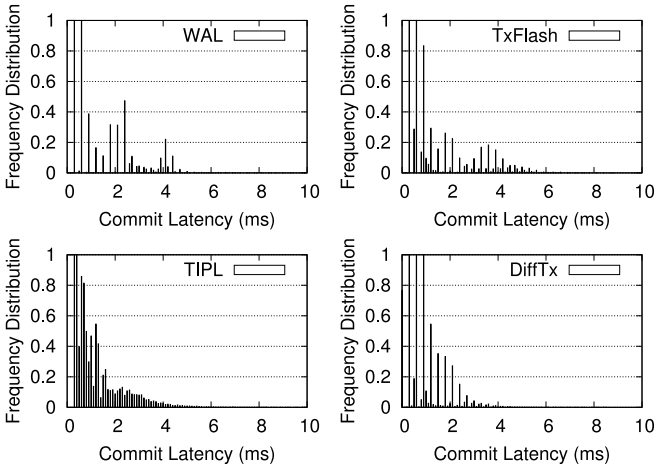


Fig. 9. Variance of commit latencies (TPC-C).

4.3 Endurance

We measure the write amplification to evaluate the protocol impact on flash endurance. Write amplification [2] is calculated by dividing the flash memory write size with the workload write size, which is the sum of page size in full page updates and differential data size in partial page updates.

Fig. 10 shows the write amplification of different protocols. Generally, WAL(M) and TIPL write less than WAL(D), TxFlash writes less than WAL(M), and DiffTx writes the least. WAL(D) writes both data and metadata twice, respectively in the log area and the data area. WAL(M) writes only the metadata twice. TxFlash reduces the writes to only once, by leveraging the no-overwrite property of flash memory. While the three protocols write in page units, TIPL reduces write traffic in byte units, by logging only the differential parts of each page. But TIPL write three times, respectively in the in-page log, the system log and the data pages. In contrast, DiffTx logs only the differential data so as to reduce the write traffic. The merge operation could also mitigate the write amplification of the log writes. Thus, DiffTx can achieve low write amplification. All evaluated workloads in Difftx have write amplification of a little over one. In other words, the write data to flash memory has nearly the same size as the application write. DiffTx limits write amplification to a low degree.

Workload varmail shows different results for WAL(D) and TIPL. Though TIPL writes only differential data, it has higher write amplification than WAL(D), which writes data in full pages twice. This is because most pages in varmail have update size larger than 512 B, which is the size of the log sector in TIPL. The in-page logging technique, which allocates one dedicated log sector for one page, is ineffective for this kind of workloads. In contrast, DiffTx uses only one log, S-Log, to accommodate differential data from different pages, and is more effective in write traffic reduction.

4.4 Checkpoint Overhead

We count the checkpoint frequency and measure the total checkpoint time to evaluate the checkpoint overhead of DiffTx against WAL(D), WAL(M) and TIPL. All of them use default log size (32 MB). TxFlash does not have checkpoint operations and is not evaluated.

Table 4 shows the measured checkpoint frequency and checkpoint time as well as the calculated average checkpoint time. DiffTx has a much lower checkpoint frequency than both WAL(D) and WAL(M), and thus has much lower checkpoint time in total. Because only differential data are

TABLE 4
Checkpoint Overhead

Workloads	avg. ckpt time (ms)				# of ckpts				tot. ckpt time (ms)			
	WAL(D)	WAL(M)	TIPL	DiffTx	WAL(D)	WAL(M)	TIPL	DiffTx	WAL(D)	WAL(M)	TIPL	DiffTx
Fileserver	1,564	1,398	280	0	24	39	13	0	33,563	61,017	3,639	0
Varmail	13	20	2	3	102	102	44	17	1,302	2,045	78	43
Webproxy	1,143	122	2.3	17	58	157	100	3	7,060	179,505	230	52
TPC-C	1,513	1,513	1	518	171	171	107	10	258,646	258,646	141	5,181

updated to the log, the log in DiffTx is used up slower than that in WAL(D) or WAL(M). DiffTx has lower total checkpoint time but large average checkpoint time than TIPL. Thus, in term of total checkpoint time, DiffTx and TIPL win in different workloads:

1) In fileserver workload, there is no checkpoint in DiffTx. This is because fileserver has either full page updates, which do not need logging, or partial page updates with extreme small writes. The sizes of these partial page updates are tens of bytes, as shown in Fig. 2, leading to higher efficiency in differential logging. Therefore, DiffTx gains more in workloads like fileserver.
2) In TPC-C workload, TIPL has smaller total checkpoint time than DiffTx. This is because some pages in TIPL have been merged and persisted due to the used up of the in-page log, which has a log size of 512 B. The in-page log is consumed more quickly in TPC-C, and this causes frequent merge operations in in-page logging of TIPL. The overhead has been amortized in the in-page logging. In contrast, DiffTx performs all merge operations on checkpoint. Also, DiffTx needs to read pages to be merged with the differential data, and this leads to slight higher checkpoint time. Thus, TIPL has lighter checkpoint time in workloads like TPC-C.

Even though the checkpoint overhead in some workloads can be amortized to the in-page logging in TIPL, DiffTx has better performance and endurance in whole, as discussed in Sections 4.2 and 4.3. And generally, DiffTx has lower checkpoint overhead than WAL(D), WAL(M) and TIPL.

4.5 Recovery Time

We also measure the recovery time of the evaluated protocols. The recovery time is the log or device scan time to recover the consistent state after system failures.

Fig. 11 shows the recovery time of WAL, TxFlash and DiffTx. Since both WAL(D) and WAL(M) scan only the log to recover the consistent state, the maximum recovery time depends on the log size. For the default log size of 32 MB in evaluation, WAL(D) and WAL(M) (referred to WAL in Fig. 11) have the recovery time of 64 ms, which is the least of the three. TxFlash has to scan the whole device to find all pointers in each page for its recovery protocols. For the default SSD size of 32 GB, it takes 6,957 ms, which is the longest of the three. DiffTx needs to read both the log and the base pages, which are needed to be merged with the differential data in the log to recover the latest page versions. DiffTx has the modest recovery time, 633 ms. It takes much shorter time than TxFlash, because there is no need to scan the whole SSD. It takes longer time than WAL, because it needs to read base pages for differential data in the log. TIPL needs to read the log, but does not need to perform merge operations. The merge operations are performed by the in-page logging when a page is read. So, TIPL takes shorter recovery time, 114ms. In all, recovery time in DiffTx is slight larger than that in WAL and TIPL, but is much smaller than that in TxFlash. In the four protocols, recovery time in WAL, TIPL and DiffTx depends on the amount of valid log in the log area, and recovery time in TxFlash depends on the size of an SSD. When the SSD capacity grows, recovery time in TxFlash is expect to increase linearly, but not in DiffTx. As such, recovery time in DiffTx is acceptable.

4.6 Impact of the Abort Ratio

To evaluate the abort ratio impact in different protocols, we measure both the effective throughput (IOPS for committed transactions) and the write traffic (the number of pages that are written) of each workload under different abort ratios.

Fig. 12a shows the effective throughput of TPC-C workload under different abort ratios. All evaluated protocols, including WAL, TIPL, TxFlash (SCC/BPCC) and DiffTx, show a decrease in effective throughput with the increase of abort ratio. Among these protocols, SCC shows a sharper decrease from 13,910 IOPS on abort ratio 0 percent to 1,579 IOPS on abort ratio 50 percent. This is because SCC forces the aborted pages to be erased before a new version is written. The forced erase operations exaggerate the garbage collection cost. While all the protocols show decreased performance as the abort ratio increases, DiffTx has the best performance and shows an average decrease in performance.

Fig. 12b shows the write size of TPC-C workload under different abort ratios to further explain the abort ratio impact on performance. SCC shows a dramatic increase of write size when abort ratio increases, e.g., write size on abort ratio 50 percent is 3.6 times that on abort ratio 0

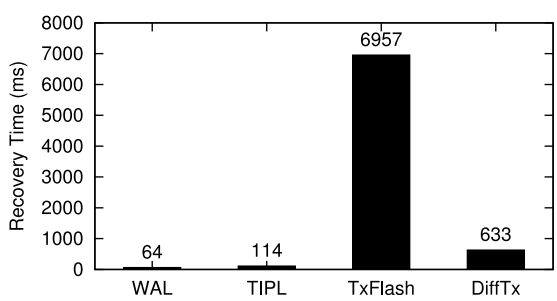
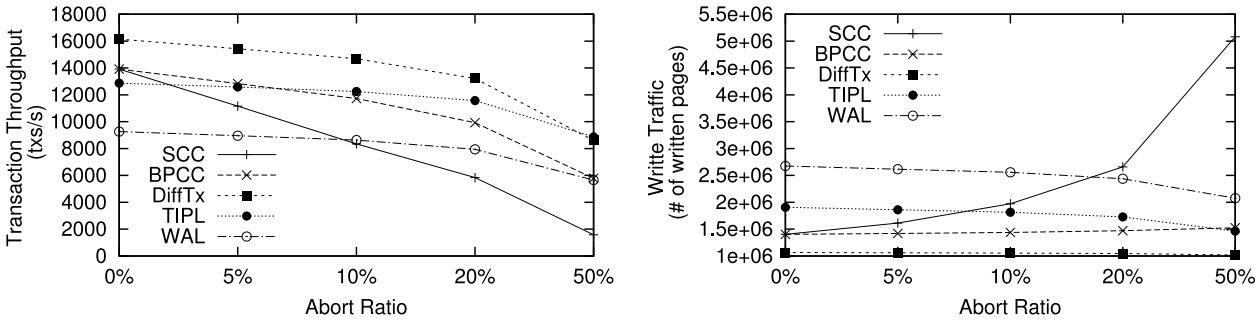


Fig. 11. Recovery time of different protocols.



(a) Throughput under Different Abort Ratios (b) Write Traffic under Different Abort Ratios

Fig. 12. Impact of the abort ratio on transaction throughput and write traffic (for workload TPC-C).

percent. This is because of the aforementioned forced erase restriction. Other protocols do not show a large difference in write size when abort ratio increases. Among all these protocols, DiffTx shows the smallest write traffic.

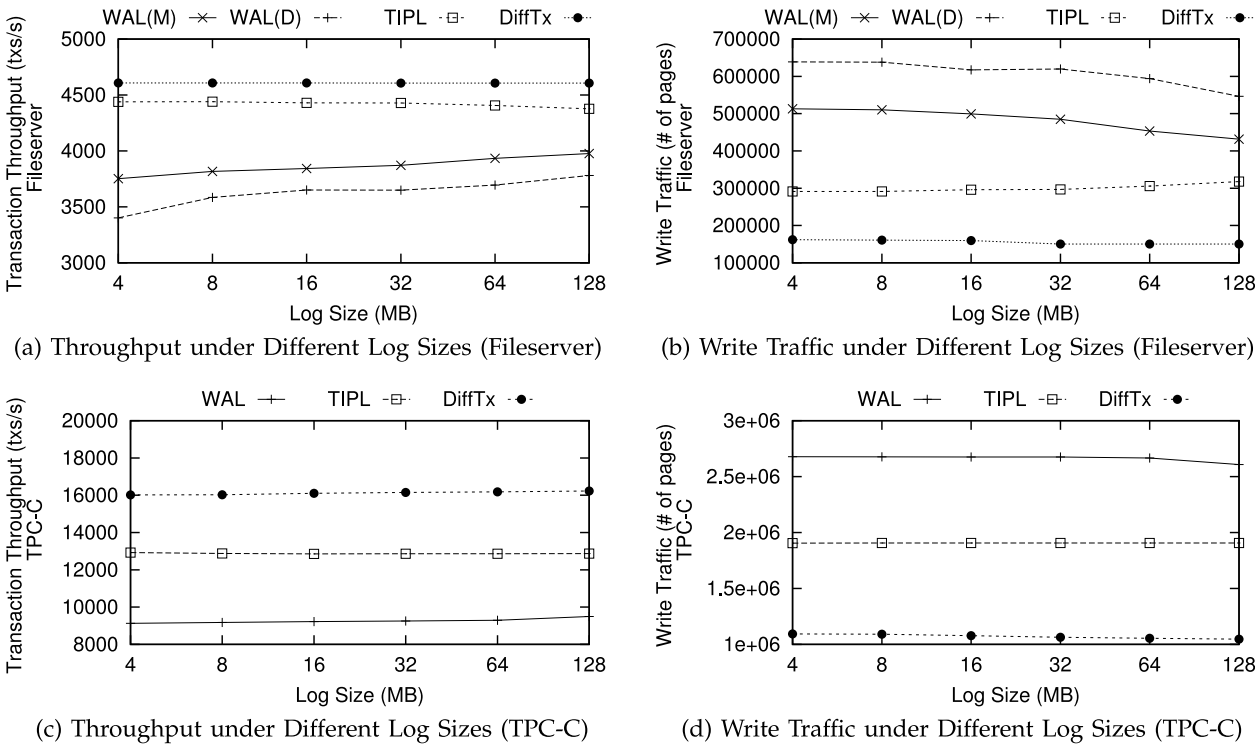
4.7 Impact of the Log Size

We measure the transaction throughput and the write traffic of fileserver and TPC-C workloads with different log sizes to evaluate the log size impact of the protocols. Varmail and webproxy workloads show similar results and thus are omitted in this paper.

Fig. 13 plots transaction throughput and write traffic of fileserver and TPC-C with different log sizes. Fig. 13a shows transaction throughput of fileserver workload with log size ranging from 4 to 128 MB. From the figure, we can observe that DiffTx shows a stable performance under various log sizes while WAL is more sensitive to the change of log size. WAL shows an approximately 10 percent increase in transaction throughput as log size is increased from 4 to 128 MB, while DiffTx shows no significant increase. Fig. 13b shows write traffic of fileserver workload with different log sizes, which explains the differences in transaction throughput shown in Fig. 13a. The reason is that more pages can be merged in WAL when log size becomes larger. The increased possibility of merge operations leads to decreased write size and thus increases transaction throughput. Comparatively, DiffTx achieves better performance with rather low log size, and the performance is stable. Figs. 13c and 13d respectively show transaction throughput and write traffic of TPC-C workload under different log sizes. TPC-C workload has similar results with filerserver workload. In TPC-C workload, DiffTx also achieves better and more stable performance than WAL and TIPL.

5 Related Work

Transaction support in flash-based SSDs. Research is active on leveraging the no-overwrite property of flash memory to support system consistency. Atomic-write [6] leverages the



(a) Throughput under Different Log Sizes (Fileserver) (b) Write Traffic under Different Log Sizes (Fileserver)

(c) Throughput under Different Log Sizes (TPC-C) (d) Write Traffic under Different Log Sizes (TPC-C)

Fig. 13. Impact of the log size on transaction throughput and write traffic (for workloads Fileserver and TPC-C).

no-overwrite property of flash memory for versioning and uses a log-structured FTL for clustering. It sequentially appends mapping entries of each transaction to the FTL mapping log. Atomic FTL [28] sequentially appends pages in log blocks for both versioning and clustering. Transactional Flash File System [29] provides transaction support for file systems in micro-controllers of NOR-based flash SSDs. OFSS [2] leverages the out-of-place update for versioning and employs an updating window to assist the page clustering of each transaction. But these studies focus on the consistency of file systems, which have no aborts.

Commit protocols in flash-based SSDs have also been designed to support general transactions which have aborts. TxFlash [5] leverages the out-of-place update for versioning and links all pages of each transaction into a cyclic list for page clustering. SCC and BPCC protocols are designed to differentiate the aborted pages from the erased pages by putting more constraints to the garbage collection (GC) process. To reduce the overhead caused by GC process in TxFlash, Flag Commit [30] proposes to use SLC NAND flash memory, which supports multiple in-place updates, to reset the pointers in the cyclic list. LightTx [7], [9] divides flash blocks with different transaction states into different zones so as to reduce the page clustering cost, even if there are aborts. But all of them use full page updates for transactions. In contrast, DiffTx supports general transactions with higher efficiency by using differential logging.

Shortcut-JFS [31] proposes to update large writes in the shadow updating way while leaving small updates for normal file system journaling. Shortcut-JFS is closely related to DiffTx but with two main differences. First, Shortcut-JFS is designed for byte-addressable non-volatile memory, which naturally supports fine-grained small writes. DiffTx has to compact the small writes into pages. Second, Shortcut-JFS updates large writes in the log area, and this causes fragmentation after log truncation. DiffTx only stores the mapping metadata in the log and thus ease the log truncation operation.

Recent research also studies the transaction support with hardware from different views. UBJ [32] proposes to use byte-addressable NVM in main memory to union the transaction versioning with the cache buffer. Literature [33] discusses the write ordering relaxing for host-side flash cache in consideration of both consistency and performance. Optimistic File System [11] relaxes write ordering on transaction commit and keeps transaction correctness, while requests are reordered in disk buffer. MARS [34] implements transactions and copies data inside SSDs to save device bandwidth, because the internal bandwidth of an SSD is larger than the device bandwidth.

Differential page writes. Differential updates have been proposed in flash-based SSDs to improve endurance. Delta-FTL [18] proposes to write only the delta part of each page by reading last version and comparing the two versions. Page-Differential Logging [15] and FTL^2 [17] share the similar idea. But this technique does not work either for WAL or shadow paging. In WAL, pages are first appended to the log area, and the two versions in the same location do not necessarily have the lineage relation. Shadow paging requires writes updated in pages and does not support differential writes.

In-page logging [14] also writes only the differences of each page. Each flash block has an allocated log space for storing the differences of its pages. Transactional IPL [16] proposes to support transactions based on IPL. Since a transaction may have pages accessed in different flash blocks, Transaction IPL allocates a system-wide log to support transaction in addition to the in-page log. The in-page log consumes more space, and also incurs high writeback frequency when a log sector becomes full. In contrast, DiffTx differential partial page updates from full page updates, and uses only one log for differential logging, result in lighter write traffic.

ReconFS [13] logs the differential parts of metadata pages and provides only metadata (namespace) consistency. Comparatively, DiffTx aims for data consistency by combining the differential logging with the in-flash shadow paging.

6 Conclusion

Embedded transaction support in SSDs is promising due to the no-overwrite property of flash memory, which favors the shadow paging way with low write amplification. However, writes in a transaction usually update only a small part of each page, which are known as partial page updates. The write-ahead logging way can be more effective to reduce write amplification by differentially logging these writes. In this paper, we propose an embedded transaction protocol, DiffTx. DiffTx combines write-ahead logging (for partial page updates) with shadow paging (for full page updates), aiming at low write amplification. In addition, DiffTx exploits the internal parallelism of an SSD and reduces the transaction overhead at the same time. DiffTx clusters transaction pages by logging the mapping metadata of full page updates with the differential data of partial page updates. It also removes the write ordering on commit by delaying the completeness check of transaction writes leveraging the clean-state update property. Both the two techniques enable better performance and lower transaction overhead. Evaluations using both file system and database workloads show remarkable performance improvement and SSD lifetime extension with significant write amplification reduction.

Acknowledgments

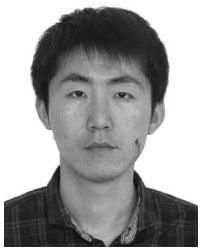
The authors would like to thank Wei Wang and Yunyun Jiang for discussions and feedbacks. This work was supported by the National Natural Science Foundation of China (Grant No. 61433008, 61327902), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), Samsung Electronics Co., Ltd., Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, and Tsinghua University Initiative Scientific Research Program. Jiwu Shu is the corresponding author.

References

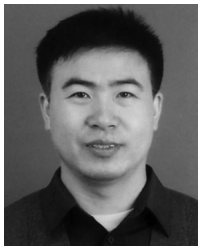
[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.

[2] 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.

[3] 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, pp. 94–162, 1992.
[4] 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. Surveys*, vol. 13, pp. 223–242, 1981.
[5] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," in *Proc. 8th USENIX Conf. Oper. Syst. Design Implementation*, 2008, pp. 147–160.
[6] 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. Arch.*, 2011, pp. 301–311.
[7] 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. IEEE 31st Int. Conf. Comput. Design*, 2013, pp. 115–122.
[8] 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.
[9] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "High-performance and lightweight transaction support in flash-based SSDs," *IEEE Trans. Comput.*, 2015, to be published http://dx.doi.org/10.1109/TC.2015.2389828.
[10] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, p. 9.
[11] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 228–243.
[12] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proc. IEEE 32nd Int. Conf. Comput. Design*, 2014, pp. 216–223.
[13] 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.
[14] S.-W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 55–66.
[15] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song, "Page-differential logging: An efficient and dbms-independent approach for storing data into flash memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 363–374.
[16] S.-W. Lee and B. Moon, "Transactional in-page logging for multiversion read consistency and recovery," in *Proc. 27th IEEE Int. Conf. Data Eng.*, 2011, pp. 876–887.
[17] T. Wang, D. Liu, Y. Wang, and Z. Shao, "FTL^2: A hybrid flash translation layer with logging for write reduction in flash memory," in *Proc. 14th ACM SIGPLAN/SIGBED Conf. Lang., Compilers Tools Embedded Syst.*, 2013, pp. 91–100.
[18] G. Wu and X. He, "Delta-FTL: Improving SSD lifetime via exploiting content locality," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 253–266.
[19] S. C. Tweedie, "Ext3, journaling filesystem," in *Proc. Ottawa Linux Symp.*, 2000, pp. 24–29.
[20] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The DiskSim simulation environment version 4.0 reference manual," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-08-101, 2008.
[21] (2012). Samsung K9F8G08UXM flash memory datasheet. [Online]. Available: http://www.datasheetarchive.com/-K9F8G08U0M-datasheet.html
[22] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group commit timers and high volume transaction systems," Hewlett-Packard Lab., Palo Alto, CA, USA, Tech. Rep. 88.1, 1989.
[23] S. C. Tweedie, "Journaling the Linux ext2fs filesystem," in *Proc. 4th Annu. Linux Expo*, 1998, pp. 1–8.
[24] (2004). Disk I/O tracing for Linux 2.6 kernels. [Online]. Available: http://www.ysaito.com/linux-iotrace
[25] (2012). Filebench benchmark. [Online]. Available: http://sourceforge.net/apps/mediawiki/filebench
[26] (2013). HammerDB. [Online]. Available: http://hammerora.sourceforge.net/
[27] (2012). PostgreSQL. [Online]. Available: http://www.postgresql.org/
[28] 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.
[29] E. Gal and S. Toledo, "A transactional flash file system for microcontrollers." in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 89–104.
[30] 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. 2012.
[31] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn, "Shortcut-JFS: A write efficient journaling file system for phase change memory," in *Proc. 28th IEEE Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–6.
[32] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 73–80.
[33] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 45–58.
[34] 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.



Youyou Lu received the BS degree from Nanjing University in 2009 and the PhD degree from Tsinghua University in 2015, both in computer science. He is currently a postdoctoral research fellow in the Department of Computer Science and Technology, Tsinghua University. His current research interests include nonvolatile memories and file systems. He is a 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, and is currently a professor in the Department of Computer Science and Technology, Tsinghua University. His current research interests include storage security and reliability, nonvolatile memory-based storage systems, and parallel and distributed computing. He is a member of the IEEE.

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

Peng Zhu received the BS degree in computer science from the Hunan Institute of Technology in 2012. He is currently working toward the master's degree in Hunan University. He was a research assistant in the Department of Computer Science and Technology, Tsinghua University from 2013 to 2014. His research interests include flash-based storage systems.