TxCache: Transactional Cache using Byte-Addressable Non-Volatile Memories in SSDs

Youyou Lu †, Jiwu Shu† §, Peng Zhu † ‡
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
†State Key Laboratory of Computer Architecture, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China
luyy09@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn, zhupeng1011@gmail.com

***Abstract*—Transaction is a common technique to ensure system consistency but incurs high overhead. Recent flash memory techniques enable efficient embedded transaction support inside solid state drives (SSDs). In this paper, we propose a new embedded transaction mechanism, TxCache, for SSDs with non-volatile disk cache. TxCache revises cache management of disk cache to support transactions using two techniques. First, it persists new-version data in non-volatile disk cache in a shadow way while protecting old-version data from being overwritten. Second, it uses pointers and flags leveraging the byte-addressability to cluster pages of each transaction and manage transaction status. The non-volatility and byte-addressability properties make TxCache an efficient transaction design. Experiments using file system and database workloads show performance improvement up to 46.0% and lifetime extension up to 33.8% compared to a recent transactional SSD design.**

I. Introduction

Data consistency is a basic element in building storage systems, including file systems and database systems. Transaction is widely used to provide data consistency. A series of operations are grouped into transactions. Writes in each transaction are atomically and durably written to persistent storage. However, software transactions explicitly write and persist data to new places followed by either copying back these new data [1] or updating the indices [2]. This amplifies the writes and incurs high overhead.

Non-volatile memories (NVMs), including flash memory and byte-addressable memories, are changing the storage architecture. Transaction designs are undergoing dramatic change in NVM-based storage systems. Embedded transaction, which supports transactions in storage devices rather than in software, is a promising way for three reasons. First, flash memory needs to be erased before overwritten, so pages in flash-based SSDs are written to free pages in an out-of-place update way. This property naturally keeps both new and old versions, which halves write size compared to data logging-based designs [3], [4], [5]. Second, internal parallelism in NVM-based SSDs can be exploited to provide high internal bandwidth, which could be much higher than the bandwidth of device interface (e.g., SATA, PCIe). Data moving can be moved inside storage devices to lighten the burden of device interface bandwidth [6]. Third, write requests may be reordered inside storage devices, and this is a threat to correctness of transaction protocols [7]. Embedded transaction, which is closer to storage media, eliminates the reordering and ensures the correctness.

Although embedded transactions in flash-based SSDs gain benefits in performance and SSD lifetime, embedded transaction designs inside SSDs are still inefficient. The inefficiency comes from two aspects. First, transactions need to keep old-version data complete before successfully updating the new-version data, and this is called *versioning*. The new-version data need persistence before the old-version data are reclaimed. The persistence increases latency in transaction commits. Second, transactions need to cluster pages for each transaction, so that it can find all pages for each transaction for recovery after failures. This is called *clustering*. While pages are distributed to different locations due to internal parallelism of SSDs, clustering pages of each transaction either hurts internal parallelism [4] or causes high overhead [3].

Meanwhile, disk cache in flash-based SSDs is becoming non-volatile. Flash-based SSDs use a flash translation layer (FTL) mapping table to keep mapping metadata from logical page number to physical flash page number. This mapping table requires durability; otherwise, the updates are lost. For the durability of FTL mapping table, disk cache is non-volatile by using either battery backed DRAM or emerging byte-addressable non-volatile memories. Fortunately, the non-volatility and byte-addressability of disk cache can be leveraged to efficiently support transactions inside SSDs. Our goal in this paper is to design an efficient embedded transaction protocol for flash-based SSDs with non-volatile disk cache.

Observations and Key Ideas: In this paper, we propose an embedded transaction design, TxCache, for SSDs with non-volatile disk cache. TxCache exploits the non-volatility of disk cache for better transaction performance, and clusters pages of each transaction using pointers or flags to minimize transaction overhead. Concretely, TxCache is optimized based on the following two observations. First, pages can be persistently stored in non-volatile disk cache. With the non-volatility, new-version pages can be persistently stored in disk cache without forced write-back. While the last committed (old-version) pages are protected by revising cache replacement algorithm of disk cache, both old and new versions are kept safe efficiently. Second, non-volatile memories are byte addressable and bit alterable. Instead of clustering all pages of each transaction, TxCache clusters page metadata for all pages in each transaction. TxCache uses pointers to link all these pages into a list. Also, TxCache uses flags to indicate transaction status of each transaction. With byte-addressability,

§Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).
‡Peng Zhu joins this work as a research assistant at Tsinghua University, and he is also a graduate student at Hunan University.

* This work is supported by the National Natural Science Foundation of China (Grant No. 61327902), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), State Key Laboratory of Computer Architecture, Shanghai Key Laboratory of Scalable Computing and Systems, and Tsinghua University Initiative Scientific Research Program.

978-1-4799-5576-3/14/$31.00

these pointers and flags are updated with little overhead for lightweight clustering.

Our contributions are summarized as follows:

1) We propose an embedded transaction design, TxCache, to support transactions inside SSDs by leveraging non-volatile disk cache.
2) We keep new-version data leveraging the non-volatility of disk cache and protect old-version data safe by revising cache algorithms, so as to efficiently support versioning.
3) We cluster page metadata instead of page data and update transaction states using flags and pointers in disk cache. The byte-addressability of disk cache leads to lightweight clustering.
4) We evaluate TxCache using both file system and database workloads. Experiments show significant performance and lifetime improvement in TxCache.

II. Background

A. Non-volatile Memories

Flash memory is a new kind of storage media which stores bits using semiconductor techniques rather than magnetic ones. Flash memory has lower access latency compared to HDDs. Typical read and write latency of flash memory is 25us and 200us [8]. Flash memory accesses are also different from HDD accesses. Flash pages cannot be overwritten, in other words, a page needs an erase operation before being written [3], [5], [8]. In addition, flash memory has endurance problem. Each flash memory cell can only be programmed and erased in limited cycles. Different from HDDs, flash-based SSDs have limited lifetime, and the lifetime is sensitive to the amount of written data [3], [8], [9].

Recently, byte-addressable non-volatile memories are undergoing fast development, such as Phase Change Memory (PCM), Spin-Transfer Torque RAM (STT-RAM), Resistive RAM (RRAM). These emerging non-volatile memories are byte addressable and provide performance comparable to DRAM, which make them alternatives to DRAM. Besides, these NVMs have orders of magnitude higher endurance than flash memory. Due to the non-volatility and DRAM comparable performance, NVMs are good candidates for disk cache.

B. Transaction Recovery

In database management systems (DBMSs), transactions have four properties: Atomicity (A), Consistency (C), Isolation (I) and Durability (D). Transaction management has two parts: concurrency control and transaction recovery. Concurrency control controls the execution sequence of concurrently executed transactions, while transaction recovery is used to recover persistent data to a consistent state. In storage systems, we focus on transaction recovery, which atomically and durably updates pages in a transaction.

There are two common techniques for transaction recovery: write-ahead logging (WAL) [1] and shadow paging [2]. WAL allocates a log area in disks and writes new-version data to the log area. Only after the new-version data are persistently written to the log and the transaction commits, the new-version data can be written back to their home locations. Shadow paging takes another approach. It writes new-version data to new locations. Instead of copying the new-version data back to their home locations, shadow paging updates the metadata pointers to point to the new-version data. This atomically changes the data version.

With the advent of new memory technologies, shadow paging is becoming more popular. There are two reasons. First, shadow paging only writes data once, and this is friendly to flash memory, which has endurance problem. Second, high random read performance in new memories mitigates the performance penalty of pointer updates in shadow paging.

C. Related Work

Recent research has investigated into embedded transaction designs in flash-based SSDs to leverage the no-overwrite property of flash memory. TxFlash [3] proposes to cluster pages of each transaction by linking them in a cyclic list. The link pointers are stored in page metadata of each page. The completeness of the cyclic list is used to identify the status of each transaction. FusionIO's Atomic Write [4] uses a log-structured FTL and clusters the mapping metadata of pages in each transaction. The log-structured mapping table is used to identify transaction status. LightTx [5] employs zones to track transaction pages of different states, so as to reduce the transaction tracking cost while providing high flexibility for better performance. However, these embedded designs ignore the non-volatility of non-volatile disk cache, which otherwise could make transaction designs more efficient.

Some work has started to use multi-level non-volatile cache for better transaction designs. UBJ [10] is designed for computer systems with non-volatile main memory. UBJ journals data in main memory and writes back only after transaction commits. Kiln [11] is designed for CPUs with non-volatile last-level cache (LLC). Kiln supports transaction consistency for computer systems with non-volatile main memory and LLC. However, both of them are not feasible for embedded transaction inside SSDs for two reasons. First, disk cache size is not as large as main memory, and it might not be able to hold large transactions. Second, cache algorithms in disk cache are different from CPU cache algorithms. Therefore, TxCache is designed for SSDs with non-volatile disk cache to support embedded transactions inside SSDs.

III. Design

In this section, we describe the TxCache design, which leverages non-volatile disk cache to efficiently support transactions inside SSDs. TxCache has two design goals to address the efficiency problem in transaction support:

1) *versioning* to eliminate duplicate flash memory writes by leveraging the non-volatility of disk cache.
2) *clustering* to efficiently track transaction status using the byte-addressability of non-volatile cache.

A. Overview

To support transactions inside SSDs, both device interface and FTL are extended in TxCache SSD, which is shown in Figure 1. In TxCache, disk cache is non-volatile using either emerging non-volatile memory (e.g., PCM, RRAM, STT-RAM) or battery-backed DRAM. Since the disk cache is non-volatile and byte-addressable, TxCache keeps transactional

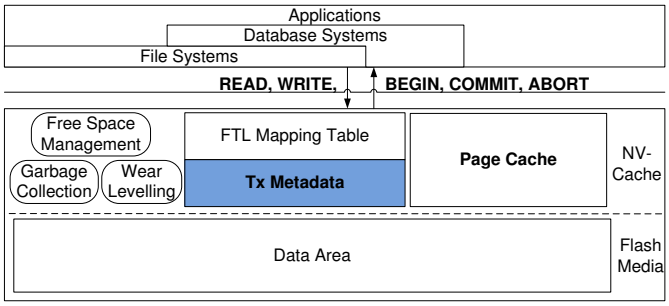


Fig. 1. TxCache SSD Framework

metadata in disk cache and extends FTL with both versioning and clustering functions. The following part of this section introduces the interface, versioning and clustering of TxCache before discussing crash recovery.

TABLE I. DEVICE INTERFACE

Operation	Description
READ(*TxID, LBA, len...*)	read data from LBA (logical block address) for the transaction TxID
WRITE(*TxID, LBA, len...*)	write data to the transaction TxID
BEGIN(*TxID*)	check the availability of the identifier TxID and start the transaction
COMMIT(*TxID*)	commit the transaction TxID
ABORT(*TxID*)	abort the transaction TxID

Table I shows the extended device interface for TxCache SSDs. Compared to traditional device interface, each read or write operation is associated with a transaction identifier (TxID). TxID indicates which transaction the read or write operation belongs to. In addition, a set of transaction control commands (i.e., BEGIN, COMMIT and ABORT) are introduced to control status of each transaction.

With transaction control commands, transaction semantics are passed from software to devices. The BEGIN command checks the availability of a transaction identifier, TxID. If the TxID is available, the command returns success; Otherwise, it chooses an available TxID and returns. The command then initiates the transaction metadata data structure in disk cache, including the TxID and the TxStatus (transaction status). The COMMIT command commits a transaction and updates the FTL mapping table to make the pages in the committed transactions accessible. The ABORT command aborts a transaction and frees the memory space that has been allocated to this transaction.

Storage devices usually have one or multiple queues for command operations, e.g., Native Command Queuing (NCQ) in SATA devices. Commands are reordered inside devices for optimization. In TxCache, commands are reordered only between transaction control commands. To provide correct transaction semantics, reads or writes are not allowed to be reordered across transaction control commands.

B. Versioning

In transactions, versioning is used to keep both the old and new versions complete and durable. The old-version data are protected, because they are used for roll-back in case of transaction aborts. Thus, the old-version data cannot be overwritten or freed before the transaction is committed and the new-version data are written completely. In TxCache, new-version data are buffered in the disk cache before written back to flash memory. Since the new-version data in disk cache survive system failures, the basic idea is to keep the latest version in disk cache and protect the old version in flash memory. Two restrictions are needed to protect both the old and new versions from being destroyed.

The first restriction is that the latest committed version in disk cache can not be overwritten. Instead of overwriting pages in disk cache, TxCache allocates free pages for page updates. For page updates in a running transaction, mapping entries in the FTL mapping table are not updated before the transaction is committed. Only after the transaction is committed, the mapping entries of pages in the transaction are updated, and these pages are the latest version. And then, the previous versions are freed. In this way, TxCache keeps the latest committed version safe in disk cache.

The second restriction is that pages in uncommitted transactions cannot be written back to flash memory. In TxCache, cache replacement is revised, and only pages of committed transactions are candidates for replacement. For large transactions whose size is larger than the cache size, TxCache allocates a space in flash memory to store these writes. The space is called *extended cache* in this paper. Pages in current running transactions are logged to the extended cache before transactions are committed. After the transaction is committed, pages in both disk cache and extended cache are written back to their home locations in flash memory. In this way, TxCache protects old-version data from overwritten even when transaction size is larger than disk cache size.

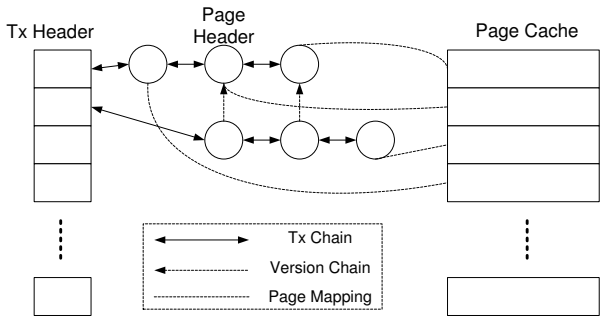


Fig. 2. Transaction Metadata in TxCache

Figure 2 illustrates the organization of page cache and transaction metadata in TxCache. Since disk cache is non-volatile, pages are buffered in page cache persistently without being written back. Exceptions are for transactions whose sizes are larger than cache size. While not all pages can be buffered in disk cache, some are written back to extend cache in flash memory. In order not to destroy old-version in flash memory (for the second restriction aforementioned), these evicted pages in uncommitted transactions are logged to extend cache, a dedicated log area in flash memory.

In TxCache, each page in page cache or log area has a page metadata (Page Header) data structure. Page Mapping lines in Figure 2 illustrate the pointers. Page Headers for pages, as well as Tx Header, in each transaction are linked into a list (i.e., Tx Chain). Tx Header records metadata (e.g., transaction status) that describes a transaction. Different versions of each page may be accessed by different transactions. To protect latest commit version (for the first restriction aforementioned), each

Page Header points to the last committed version, shown as Version Chain in Figure 2. Only a pointed Page Header can free its pointed Page Header in the Version Chain. The Version Chain makes sure the latest committed page version is safe. Tx Headers and Page Headers, as well as Tx Chain, Version Chain and Page Mapping, form transaction metadata of TxCache. Transaction metadata are stored persistently in disk cache and are used to manage transactions.

C. Clustering

While versioning keeps both old and new versions complete and durable, clustering tracks the status of each transaction as well as its pages. Once system fails, clustering is used to identify all pages for each transaction, so that all these pages can be rolled forward or backward respectively for committed and uncommitted transactions. In legacy transaction protocols, clustering incurs high cost. Pages are physically continuous in the log area for each transaction in WAL, and they needs to be copied to their home locations. TxFlash uses pointers to link all pages in a circle list, and the pointer maintenance incurs high overhead on garbage collection. In TxCache, page metadata instead pages are clustered, and these page metadata are persistently updated using flags and pointers in disk cache due to the byte-addressability. This leads to low-overhead clustering in TxCache.

As shown in Figure 2, page metadata for pages of each transaction are linked into a list in transaction metadata. Transaction status changes are kept using three flags (FLAG_C, FLAG_M and FLAG_WB) in Tx Header. Commit status flag FLAG_C is used to indicate the commit status. It is set to 0 when the transaction starts, and is set to 1 once the transaction commits. The commit status flag is used to check the commit status of each transaction. Mapping update status flag FLAG_M is used to indicate whether the FTL mapping table has been updated. It is initialized to 0, and is set to 1 after the mapping table update completes. Since the mapping table is updated only after transaction commits, the flag is used to check the status and roll-forward the incomplete FTL mapping table update. Checkpoint (or writeback) flag FLAG_WB is used to indicate the checkpoint status of each transaction, i.e., whether pages of this transaction have been copied back (i.e., checkpointed) to their home locations. Only after all pages have been written back (FLAG_WB is set), the transaction header is reused. In other words, a transaction (as well as the identifier TxID) is free only after all its pages are written back.

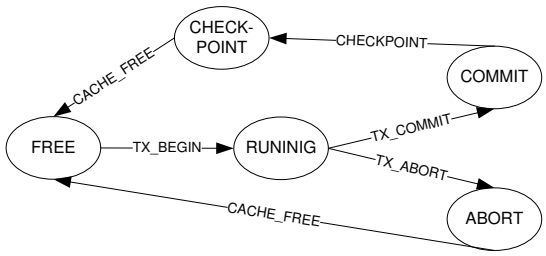


Fig. 3. Transaction State Transition

In all, transaction status of each transaction is tracked by updating the flags in its transaction header. Figure 3 shows the state transition of a transaction header. Initially, a transaction is free. It turns to RUNNING when the transaction begins, and the FLAG_F (transaction free flag) is unset. If transaction commits, FLAG_C is set, and it turns to COMMIT state; if transaction aborts, FLAG_F is set, and the transaction turns to ABORT state. For abort transactions, cache pages are freed. For committed transactions, the mapping entries are updated to the FTL mapping table followed by setting FLAG_M. After that, cache pages can be evicted to flash memory, and FLAG_WB is set when all cache pages are written back. After all cache pages are written back, the transaction turns to CHECKPOINT state. Then, cache pages can be freed followed by resetting FLAG_F.

D. Crash Recovery

When system fails after unexpected software bugs or power failures, data on storage devices need to be recovered to a consistent state. In TxCache, incomplete transactions are kept in disk cache. Therefore, TxCache only needs to check the status of transactions in disk cache, so that it can drop all incomplete transaction and roll forward complete ones. Recovery steps of TxCache are as follows:

1) *Transaction Status Identification*: The first phase is to identify status of each transaction. This is done by checking the flag bits stored in transaction header. For all non-empty transactions, the commit flag is checked to differentiate committed transactions from uncommitted ones.
2) *Redo and Undo*: The second phase is to roll forward committed transactions and roll back uncommitted ones. For the uncommitted transactions, the recovery process frees all pages in memory and sets these transactions free. For the committed transactions, the recovery process further checks the mapping update flag. If mapping update has not completed, it iterates each page header and updates the FTL mapping table.

IV. Evaluation

In this section, we compare TxCache with previous designs, including WAL [1] and TxFlash [3], to evaluate both performance and endurance effects of TxCache. We also evaluate the impact from different log sizes. Finally, we estimate the recovery time.

A. Experimental Setup

We evaluate TxCache using Microsoft Research's trace-driven SSD simulator [8], which is based on DiskSim [12]. We extend this simulator with transaction interface (as shown in Table I) and transaction protocols, including WAL [1], TxFlash [3] and TxCache. We call it TxSSD simulator. TxSSD simulator is configured using flash memory parameters as listed in Table II. In addition, we set both read and write latency of non-volatile disk cache to 0.15us. The default size of non-volatile disk cache is set to 32MB.

In the evaluation, we use both file system and database workloads. For file system workloads, we revise the jdb (journal block device) modules in ext3 file system to collect the transaction traces. We use fileserver, varmail and webproxy workloads from filebench benchmark [13]. The three workloads respectively emulate the file server, mail server and

TABLE II. TXSSD PARAMETERS

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

web proxy server workloads. We run each workload for ten minutes and collect the traces. For database workloads, we instrument the PostgreSQL [14] source code and record the XLog operations for transaction traces. We run DBT-2 [15] to collect TPC-C workload traces on PostgreSQL, in which the number of client connects is 7 and the number of warehouses is 28. In total, we collect 1,328,700 requests for TPC-C workload.

B. Transaction Throughput

We first measure the transaction throughput to evaluate the performance gains in TxCahce. We compare TxCache with traditional WAL, TxFlash (an shadow paging variant optimized for flash memory). We also compare TxCache with NVCache, which uses non-volatile disk cache as persistent storage but has no transaction support, to measure the transaction overhead in TxCache.

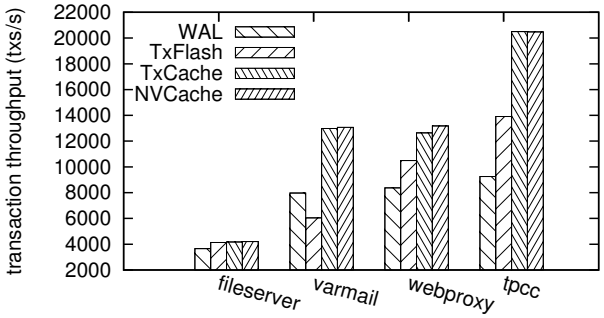


Fig. 4. Transaction Throughput Evaluation of Different Protocols

Figure 4 shows the transaction throughput of WAL, TxFlash, TxCache and NVCache. Compared to WAL, TxCache shows a throughput increase of 14.3%, 63.0%, 51.0% and 121.4% respectively for fileserver, varmail, webproxy and TPC-C workloads. This is because WAL writes twice to the flash memory and almost doubles the write traffic. Compared to TxFlash, TxCache shows a throughput increase of 0.9%, 115.1%, 20.5% and 47.4% respectively for the four workloads. In file server workload, due to the large sizes of transactions, pages are frequently written to extended cache, which is located in flash memory. The performance benefit is not as significant as others. But for workloads like varmail or TPC-C, which have better locality, TxCache benefits more from the page merging in disk cache.

Figure 4 also shows the transaction throughput of NVCache, which persistently stores page but does not support transactions. By comparing TxCache and NVCache, we conclude TxCache incurs little overhead in transaction support. The throughput penalty is within 4.1% and has an average of 1.4%.

C. Endurance

We measure total write size of flash writes to evaluate the endurance impact on SSDs.

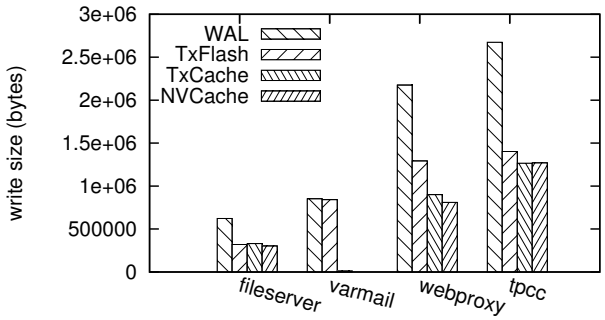


Fig. 5. Endurance Evaluation of Different Protocols

Figure 5 shows the flash memory write size in WAL, TxFlash, TxCache and NVCache. Compared with WAL, TxCache reduces write size by 46.9%, 98.8%, 58.6% and 52.6% respectively for fileserver, varmail, webproxy and TPC-C workloads; Compared with TxFlash, TxCache reduces write size by -3.5%, 98.8%, 29.4% and 9.7%, respectively. There are two observations. One is the inefficiency in write size reduction for fileserver workload. This is because file server workload has large transaction sizes, and extend cache is extensively used to hold large transactions. The other is the dramatic decrease in varmail workload. This is because varmail workload has good locality, and large portion of pages can be merged before being written back to flash memory.

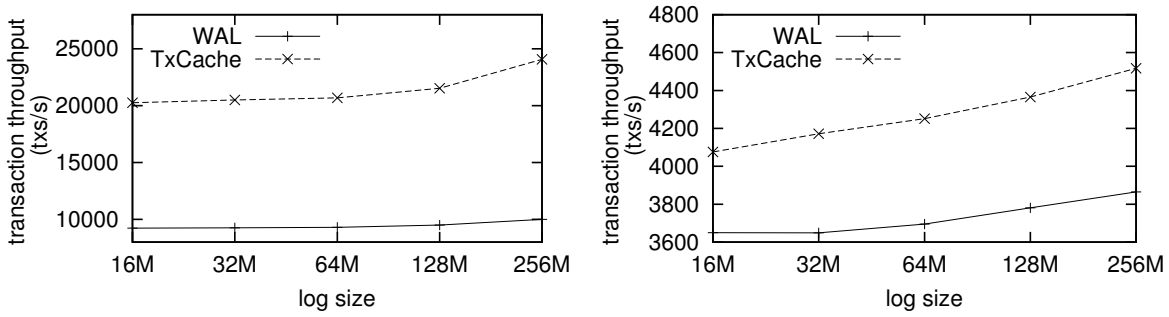
Figure 5 also shows flash memory write size in NVCache. Compared with NVCache, TxCache has comparable write size. This shows that TxCache has small endurance overhead in transaction support.

D. Impact of Log Size

To evaluate the throughput and endurance impact of log size, we vary the log size from 16MB to 256MB to measure the performance of WAL and TxCache. WAL allocates the log space in flash memory, so the log size is the flash memory log size. TxCache uses non-volatile disk cache as the log, so the log size is the size of non-volatile disk cache. Since TxFlash does not use logs, it is not evaluated.

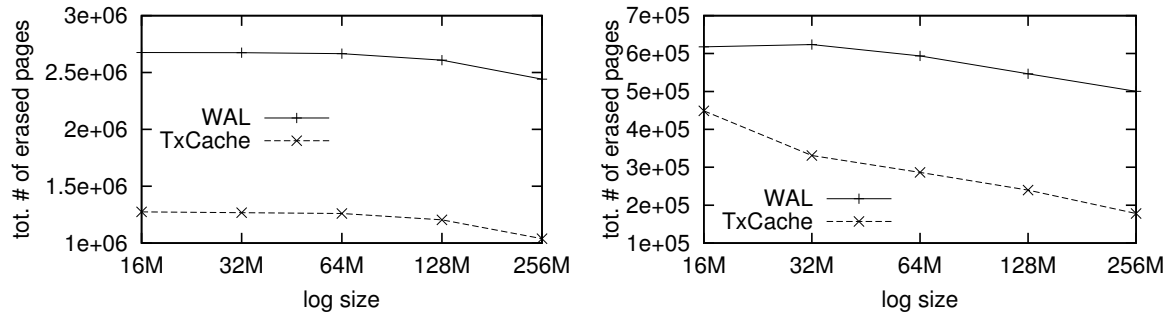
Figure 6 shows transaction throughput in WAL and TxCache under different log sizes. From the figure, we have two observations. The first observation is that both WAL and TxCache benefit more from larger log space. This is because more pages can be merged before being written back to their home locations when log space is large. The second observation is that TxCache has a larger increase when log size increases. This is because the old versions except the latest committed one in non-volatile disk cache can be overwritten in TxCache. In contrast, all updates are logged in WAL, and all versions are kept until garbage collection. When the workload has good locality, obsolete memory space can be reused. Therefore, TxCache benefits more from larger log space.

Figure 7 shows the write size in WAL and TxCache under different log sizes. Both WAL and TxCache writes less when



(a) Throughput vs. Log Size (TPC-C) (b) Throughput vs. Log Size (fileserver)

Fig. 6. Impact of Log Size on Performance (measured in transaction throughput)



(a) Endurance vs. Log Size (TPC-C) (b) Endurance vs. Log Size (fileserver)

Fig. 7. Impact of Log Size on Endurance (measured in total number of erased pages)

log size increases, and TxCache deceases more than WAL when log size is larger. These further explain the reasons of throughput improvement in Figure 6.

E. Recovery Time

TABLE III. RECOVERY TIME

Protocols	WAL	TxFlash	TxCache
Recovery Time	64 ms	6,957 ms	0.019 ms

Table III shows the recovery time for WAL, TxFlash and TxCache. Log size is set to 32MB in this evaluation. Results show TxCache has negligible recovery time compared to WAL and TxFlash. During recovery, TxCache needs to scan the transaction headers in the non-volatile disk cache. For a maximum concurrency of 128 transactions, it takes 19 microseconds. In contrast, WAL needs to scan the log space in flash memory, and TxFlash needs to scan the whole drive for FTL mapping table reconstruction and transaction status identification. WAL and TxFlash respectively spend 64 and 6,957 milliseconds. TxCache achieves nearly instant recovery and is much faster than WAL and TxFlash.

V. CONCLUSION

In flash-based SSDs, embedded transaction leveraging no-overwrite property is an promising way to provide system consistency. While non-volatile memories are being used as disk cache, TxCache further exploits the non-volatility and byte-addressability of non-volatile disk cache to make transaction support more efficient. Versioning is efficiently supported by persistently storing new-version data in non-volatile cache without being written back. Clustering is achieved using pointers and flags, and this is lightweight for byte-addressable memory accesses. Evaluations using both file system and database workloads show significant improvement in transaction throughput and SSD lifetime.

REFERENCES

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

[2] J. Gray *et al.*, "The recovery manager of the System R database manager," *ACM Computing Surveys*, 1981.

[3] V. Prabhakaran *et al.*, "Transactional flash," in *OSDI'08*, 2008.

[4] X. Ouyang *et al.*, "Beyond block I/O: Rethinking traditional storage primitives," in *HPCA'11*, 2011.

[5] Y. Lu *et al.*, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *ICCD'13*, 2013.

[6] J. Coburn *et al.*, "From aries to mars: Transaction support for next-generation, solid-state drives," in *SOSP'13*, 2013.

[7] V. Chidambaram *et al.*, "Optimistic crash consistency," in *SOSP'13*, 2013.

[8] N. Agrawal *et al.*, "Design tradeoffs for SSD performance," in *USENIX ATC'08*, 2008.

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

[10] E. Lee *et al.*, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *FAST'13*, 2013.

[11] J. Zhao *et al.*, "Kiln: closing the performance gap between systems with and without persistence support," in *MICRO'13*, 2013.

[12] G. R. Ganger *et al.*, "The DiskSim simulation environment version 2.0 reference manual," 1999.

[13] "Filebench benchmark," http://sourceforge.net/apps/mediawiki/filebench.

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

[15] "DBT2 test suite," http://sourceforge.net/apps/mediawiki/osdldbt.