Transactional NVM Cache with High Performance and Crash Consistency

Qingsong Wei
Data Storage Institute, A*STAR
WEI_Qingsong@dsi.a-star.edu.sg

Chundong Wang
Data Storage Institute, A*STAR
wangc@dsi.a-star.edu.sg

Cheng Chen
Data Storage Institute, A*STAR
National University of Singapore
CHEN_Cheng@dsi.a-star.edu.sg

Yechao Yang
Data Storage Institute, A*STAR
yangyc@dsi.a-star.edu.sg

Jun Yang
Data Storage Institute, A*STAR
yangju@dsi.a-star.edu.sg

Mingdi Xue
Data Storage Institute, A*STAR
XUE_Mingdi@dsi.a-star.edu.sg

ABSTRACT

The byte-addressable non-volatile memory (NVM) is new promising storage medium. Compared to NAND flash memory, the next-generation NVM not only preserves the durability of stored data but has much shorter access latencies. An architect can utilize the fast and persistent NVM as an external disk cache. Regarding the system's crash consistency, a prevalent journaling file system needs to run atop an NVM disk cache. However, the performance is severely impaired by redundant efforts in achieving crash consistency in both file system and disk cache. Therefore, we propose a new mechanism called **t**ransact**i**onal **N**VM disk **ca**che (Tinca). In brief, Tinca jointly guarantees consistency of file system and disk cache and removes the performance penalty of file system journaling with a lightweight transaction scheme. Evaluations confirm that Tinca significantly outperforms state-of-the-art design by up to 2.5× in local and cluster tests without causing any inconsistency issue.

CCS CONCEPTS

• **Information systems** → **Storage class memory**; • **Software and its engineering** → **Consistency**; **File systems management**; **Operating systems**;

KEYWORDS

Non-volatile Memory, Transactional NVM Cache, Data Consistency

ACM Reference format:
Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi Xue. 2017. Transactional NVM Cache with High Performance and Crash Consistency. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
DOI: 10.1145/3126908.3126940

1 INTRODUCTION

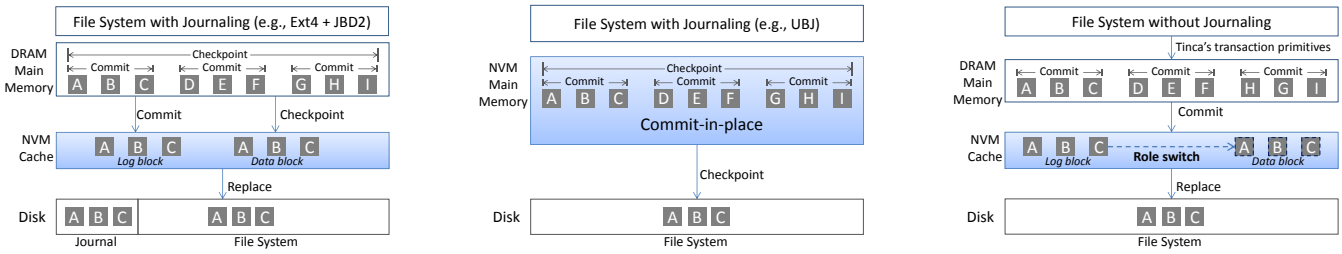
High-performance computing (HPC) and Big Data applications demand massive I/O throughputs from storage systems. Recently the next-generation non-volatile memory (NVM) technologies, including spin-transfer torque RAM (STT-RAM), phase change memory (PCM), resistive RAM (ReRAM), and 3D XPoint, provide new promising storage media and attract strong interests from both academia and industry [1–10]. These emerging NVM technologies differ from NAND flash memory with shorter access latency and byte-addressability, so they can be put onto DIMM beside DRAM for CPU's direct load and store [11–14]. Therefore, writing data persistently into NVM can be through memory copy followed by CPU cache line flush (e.g., `clflush`) and memory fence (e.g., `sfence`). An architect can leverage NVM's DRAM-like access speed and disk-like durability for high performance and crash consistency that are of paramount importance for a computer system. In this paper, we consider incorporating NVM-based external disk cache to a ferromagnetic hard disk drive (HDD) or a flash-based solid state drive (SSD). Using a faster storage medium as disk cache has become essential and effective to improve the system's overall performance, especially within distributed storage platforms for HPC and Big Data [6, 15–22]. State-of-the-art cache management mechanisms, like Flashcache [23], bcache [24], and FlashTier [17], manage a flash-based SSD cache for HDDs.

On the other hand, many modern file systems, such as XFS [25], NTFS [26], and Ext3/4 [27, 28], rely on the *journaling* mechanism for crash consistency. To modify data of a file in situ is unfavored by file system, as an unexpected crash that happens in the middle of in-place updating may cause data inconsistency. With journaling, modified data are coalesced in a unit of transaction and first *committed* to a journal. Later they are written to desired locations in file system via *checkpointing* [29–32]. Yet a committed data block in the journal and its corresponding checkpointed data block in the file system have the same contents, but write operations are done twice. That's the problem of *double writes* of journaling.

When NVM cache is interposed between a journaling file system and a disk, as shown in Figure 1(a), double writes of journaling result in double memory copies and double executions of cache line flush and memory fence to NVM. Double writes hence enforce significant write amplifications to NVM cache. Cached data blocks are eventually flushed to disk, so double writes also entail amplified disk I/Os. Consequently, the performance gain by employing fast NVM cache is compromised. Worse, considering the limited write endurance of some NVM technologies [1, 5, 8, 33–35], double writes adversely affect the lifetime of NVM cache.

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126940



(a) Conventional Architecture (b) Architecture of UBJ (c) Architecture of Tinca

Figure 1: An Overview of Architectures

Meanwhile, NVM disk cache itself must be crash consistent [17, 23]. A disk cache needs to maintain *cache metadata* for address mapping, caching states, space management, etc. FlashTier and bcache use a log (journal) to record modified cache metadata while Flashcache synchronously updates cache metadata once a data block is written into cache. This is another source of write amplifications onto cache device because cache metadata are organized in block. Nonetheless, journaling file system and disk cache do not interact as one works separately from the other one to attend its own scope of crash consistency. The joint write amplifications of redundant efforts for crash consistency in journaling file system and disk cache result in severe performance penalties. If such redundant efforts can be removed, the performance shall be promoted.

Using NVM to redesign journaling has been investigated. Lee et. al. [31] proposed UBJ that unions journal device with page cache of NVM-based main memory, as shown in Figure 1(b). UBJ commits-in-place in NVM by freezing data and later checkpoints them to file system (disk) to free NVM space. However, when updating data of a frozen block, UBJ cannot overwrite it but must do a memory copy (`memcpy`) for out-of-place updating, which is on the critical path for an application to write data. Furthermore, UBJ follows the checkpoint of conventional journaling as its checkpoint unit is one committed transaction with many blocks. As a result, using checkpoint to free NVM space takes longer time when multiple blocks have to be written to disk. Also, access speeds of PCM and ReRAM thwart utilizing NVM as main memory.

With such observations, we have designed a novel mechanism called **t**ransact**i**onal **N**VM disk **ca**che (Tinca). The architecture of Tinca is illustrated in Figure 1(c). In a nutshell, Tinca is a cache manager that includes a lightweight transactional support for upper-layer file system. Its main ideas are summarized as follows.

- *Tinca is a self-contained NVM caching mechanism with transactional supports for crash consistency of both file system and NVM cache.* It provides transactional primitives to a file system for use so that the file system can be relieved from implementation and runtime overheads for consistency.
- *Leveraging the non-volatility of NVM cache, Tinca avoids writing the same data twice for the aim of crash consistency. It defines how to commit a transaction without double writes.* Unlike journaling file system that relies on committing and checkpointing, Tinca does *role switch* in a transaction to make a data block switch between roles of being committed and checkpointed.
- *Inspired by the byte-addressability of NVM, Tinca employs a fine-grained method to manage cache metadata and regulate committing transactions.* Tinca organizes cache metadata to suit Tinca's transaction management so that they can be atomically updated in a transaction.

Table 1: Typical DRAM and NVM Technologies [7, 10, 39]

Parameter	DRAM	STT-RAM	ReRAM	PCM
Density	1×	1×	2× to 4×	2× to 4×
Read Latency	60ns	100ns	200ns to 300ns	200ns to 300ns
Write Speed	~1GB/s	~1GB/s	~140MB/s	~100MB/s
Write Endurance	10^{16}	10^{16}	10^{6}	$10^{6} - 10^{8}$

A prototype has been built with NVDIMM. Extensive experiments show that Ext4 with Tinca delivers the throughput up to 2.5× as much as that of original Ext4 with a conventional caching mechanism in local and cluster tests with identical crash consistency achieved.

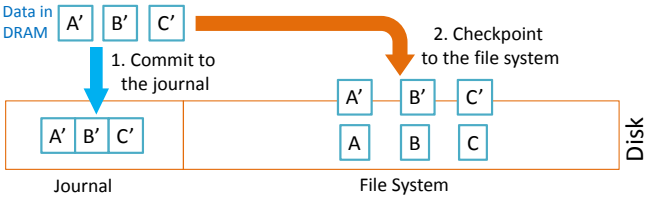
The rest of this paper is organized as follows. Section 2 present the background of NVM, caching mechanisms and crash consistency of file system. Section 3 shows the motivation of the paper. Section 4 details the design of Tinca. Section 5 shows the performance evaluation in local and cluster platforms. Section 6 discusses related work. Section 7 concludes the paper.

2 BACKGROUND

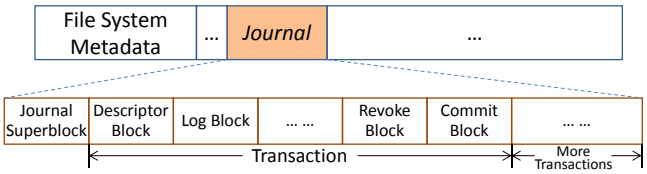
2.1 Non-volatile Memory

In this paper, we consider a scenario which uses NVM sitting on memory bus beside DRAM as an external disk cache. So CPU can directly load and store data with NVM [2, 11–13]. Characteristics of STT-RAM, ReRAM, and PCM are shown in Table 1. STT-RAM has a comparable read speed to SRAM and a high integration density compared to DRAM [36]. PCM has a larger density than DRAM but suffers from limited write endurance [1, 5–8]. ReRAM has some similar characteristics to PCM [7, 37]. Recently Intel and Micron have unveiled 3D XPoint that comes with a slower access speed but a much greater density than DRAM [38]. This paper focuses on byte-addressable NVM, excluding NAND flash memory.

It is non-trivial to consistently write data into NVM. Modern CPUs natively support an *atomic write* of 8B to memory. Using instructions like `cmpxchg16b` with the LOCK prefix can atomically write data of 16B [7]. Writing data that is greater than 16B cannot be atomically done in one instruction as of today. Moreover, writing multiple CPU cache lines to NVM may be done in a different order from the one defined by a software program [3, 10, 12, 40, 41]. A crash that happens amid reordered writes can cause inconsistency problems. For example, writing data of a file must be completed



(a) Double Writes Caused by Journaling



(b) JBD2's Journal Layout within Ext4 File system

Figure 2: Double Writes of Journaling and Journal Layout

before updating the file's modification time (`i_mtime`). If `i_mtime` is updated before writing file data but a crash occurs, the file will go inconsistent. One method to preserve a desired writing order is using regular store instructions (e.g., `mov`) followed by a combination of cache line flush (e.g., `clflush`) and memory fence (e.g., `sfence`). `clflush` explicitly invalidates and flushes a CPU cache line to NVM. `sfence` is a barrier by which store operations after `sfence` cannot proceed unless those before it have been done. Hence a series of {`sfence`, `clflush`, `sfence`} write multiple cache lines to NVM in order with considerable performance penalties. New cache line flush instructions (`clflushopt` and `clwb`) have been proposed to substitute `clflush` but still bring in overheads.

2.2 Caching Mechanisms

As data sets are becoming huge, storage I/O performance is essential for Big Data and HPC applications to quickly access the large volumes of data [16, 20–22, 42]. Distributed file systems, such as Lustre [43], Hadoop Distributed File System (HDFS) [44], GlusterFS [45], BeeGFS [46], and CephFS [47], are being developed to provide aggregate bandwidth for Big Data and HPC. Continuous growth in CPU multi-core and many-core demands fast data movement from storage to memory. However, storage performance does not keep pace with the increase of computing speed.

To fill the gap between DRAM and HDD in a cost-efficient way, caching schemes using flash-based SSD have been widely studied. SSD cache mechanisms need to cover crash consistency, address mapping, replacement and free space management. Since SSD is block device, the overhead of consistency and address mapping is significant due to write amplification. Flashcache, bcache, and FlashTier are typical SSD cache mechanisms. This paper presents transactional NVM cache with low overhead to reduce the gap between computing and storage.

2.3 Crash Consistency and Journaling

There exist different levels of file system consistency. For example, at a low consistency level, only file system metadata, like `inodes` for files, are consistent but file data are not considered. At a higher level, a file system must make file data durable before its associated `inode` is persisted. Although applied widely, this consistency level still cannot recover file data in case of crash [7, 48]. An even higher level, i.e., *data consistency*, ensures that both file system metadata and data are consistent and recoverable. In this paper we target the data consistency that is demanded by many applications [10, 48]. Unless stated otherwise, 'data' or 'data block' mentioned in following sections refers to both file system metadata and data.

Journaling is a preferred consistency technique. A journaling file system has a file system area for data storage and a journal to temporarily log modified data. Take redo journaling for an example. In Figure 2(a), Data A, B and C are to be updated. An in-place overwrite with updated A', B' and C' may leave data to be erratic if a crash happens in writing the data. The redo journaling first *commits* A', B' and C' into journal as the log data, and later *checkpoints* them to file system. In case of a crash amid the writing of A', B' and C', data can be recovered from logged copies in the journal.

The widely-used Ext4 leverages Journal Block Device (JBD2) for journaling [27, 28]. The structure of JBD2's on-disk journal is shown in Figure 2(b). It contains a journal superblock and multiple transactions. The superblock summarizes the journal, such as the block size as well as start and end addresses of free space. In a transaction, one or plural descriptor blocks mark the beginning of a transaction and describe data blocks committed in this transaction, like their block numbers in file system. Committed data blocks form 'log blocks' that follow descriptor blocks as the main body of a transaction. A revoke block tracks truncated blocks. At last a commit block ends the transaction. Once commit block is written, committing a transaction completes. Superblock, descriptor block, commit block and revoke block are journaling metadata for transaction management on disk. In our Tinca, transaction management is byte-addressable, which will be discussed in Section 4.

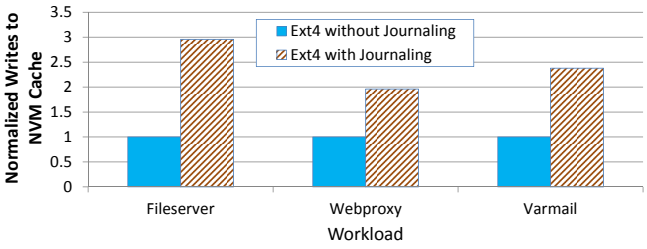
3 MOTIVATION

3.1 Overhead of File System Consistency

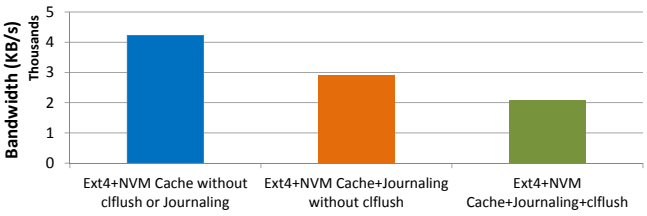
Performance Penalty Double writes of journaling entail severe write amplifications. When NVM is interposed as a cache to disk, it has to absorb such double writes before passing them down. Intuitively double writes result in double memory copies and double executions of `clflush`, the first time for the logged block in the journal and the second time for the checkpointed one in file system. Whereas, the cost of `clflush` and `sfence` is high [12, 40].

We have done two experiments to observe the impact of double writes as well as that of `clflush` and `sfence`. We first ran three workloads with Filebench [49] by mounting Ext4 on an NVM cache with flash-based SSD. Figure 3(a) depicts the write traffics to NVM cache for different workloads with Ext4 mounted in the `journal` mode (both file system metadata and data logged to the journal). It is evident that Ext4 with journaling causes about 195%~290% writes compared to Ext4 without journaling. Such double writes not only impair the performance but also affect the lifetime of NVM cache regarding the write endurance of some NVM. Therefore, *an elimination of double writes benefits both performance and reliability.*

In the second experiment we performed the same amount of random writes issued by Fio [50] with different configurations. The leftmost bar in Figure 3(b) shows the bandwidth by running Ext4 without journaling or `clflush` on an NVM cache. When journaling



(a) Write I/O Traffic to NVM Cache



(b) The impact of double writes with `clflush` and `sfence`

Figure 3: The Impact of Double Writes of Journaling

is imposed for crash consistency, the bandwidth drops by 31.5%, as indicated by the middle bar. With `clflush` and `sfence`, the performance further deteriorates by 28.3% to be the rightmost bar. Thus, *the performance gain earned by employing a fast NVM disk cache is severely compromised due to double writes of journaling as well as double executions of* `clflush` *and* `sfence`.

Implementation Efforts It is non-trivial to design and implement a file system with crash consistency. An analysis of journal file systems [51] revealed that Ext3 and ReiserFS, which were relatively mature at that time, had contained considerable vulnerabilities in their source codes. Worse, as journaling file system evolutes, journaling's enhanced transactions "add many bugs due to their large code bases" though they improve crash consistency [52]. Other consistency techniques, such as copy-on-write (COW), soft updates, and backpointer, are not easy to be implemented also. *If the underlying device can provide transactional supports, the design and implementation of a file system will become much relaxed.*

3.2 Overhead of Cache Metadata Consistency

The consistency of cache metadata must be preserved. Flashcache is a widely-used caching mechanism [17, 53]. First, Flashcache guarantees the consistency of cache metadata via a synchronous update style. It writes cache metadata from DRAM to cache device every time file system writes a block. Such a straightforward approach is effective but heavyweight. Second, Flashcache organizes cache metadata in a block format. In line with its synchronous update, writing a data block into cache always entails writing a cache metadata block into cache device. The write amplification is catastrophic. We used Flashcache to mange an NVM cache on top of an SSD and ran Fio with random writes. As told by Figure 4, if updating metadata is fully waived, the throughput can be improved by 45.2% on Ext4 with journaling. On Ext4 without journaling (double writes), the throughput can be further improved by 65.5%.

To sum up, write amplifications of NVM cache come from two sources: cached data blocks due to journaling file system and cache metadata due to disk cache's consistency preservation.

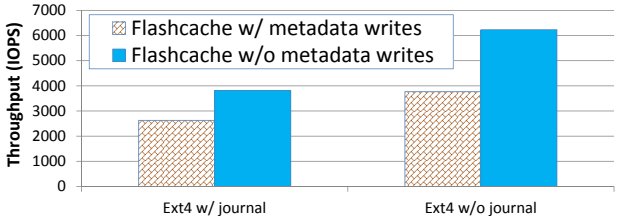


Figure 4: The Impact of Updating Cache Metadata

4 TINCA

Aforementioned observations demand an NVM caching algorithm that (1) provides transactional supports so that a file system is relieved from burdens of preserving crash consistency, (2) comprises a systematic mechanism that avoids double writes but still achieves crash consistency of both file system and disk cache, and (3) manages NVM cache metadata in a fine-grained way to facilitate the consistency mechanism and reduce performance penalties. These motivate us to develop Tinca.

4.1 Overview

Tinca stands for **t**ransact**i**onal **N**VM disk **ca**che. It is a self-contained design with both NVM caching and transactional support. Tinca's main features are summarized as follows.

Transactional Primitives Tinca defines transactional primitives that a file system can use for crash consistency. The functionalities of Tinca's main primitives are summarized as follows.

- `tinca_init_txn`(*txn*) initiates and starts a *running transaction* indexed as *txn* that is resided in DRAM and takes in data blocks to be committed.
- `tinca_commit`(*txn*, blk_0, blk_1, ...) converts a *running transaction* to *committing transaction* and commits blocks of transaction *txn* into NVM disk cache.
- `tinca_abort`(*txn*) aborts transaction *txn*. All blocks that have been written into the NVM cache will be revoked to their last consistent states.

When a file system seeks for the transactional support of Tinca, it needs to decide what data blocks to be committed in a transaction and execute corresponding primitives.

Committing Transactions without Double Writes Tinca defines transaction and commit protocol to back its transactional primitives. In particular, Tinca's commit protocol regulates how a transaction proceeds in NVM cache without writing the same data twice. The elimination of double writes is impossible for a standalone disk because modified data must be rendered durable to the disk twice for committing and checkpointing, respectively. However, NVM cache is an intermediate persistent medium on top of disk and absorbs both data blocks that are committed and checkpointed. Given a block committed, Tinca can convert it to be the corresponding block that shall be checkpointed. The checkpointing is hence waived. Tinca does such a conversion by a competent scheme called *role switch* that coordinates writing a data block and changing the block's cache entry. As a result, Tinca achieves data consistency without double writes of journaling.

Fine-grained Cache and Transaction Management Tinca manages cache entries for byte-addressable NVM in a fine-grained

way rather than a conventional block format. A cache entry is specially designed so that it can be modified using available architectural support for atomic memory write. Tinca also leverages a fine-grained structure in NVM to regulate committing transactions.

These features enable Tinca to achieve crash consistency for both file system and NVM cache. The avoidance of double writes and minimization of cache metadata updates also help Tinca to achieve high performance.

4.2 NVM Layout & Cache Metadata

Before a discussion on how to commit a transaction containing multiple blocks, we first introduce Tinca's NVM space layout and cache entry, as shown in Figure 5. The NVM space is partitioned for both purposes of transactional support and caching. The first area is a *ring buffer* that is used to regulate committing a transaction, which will be discussed in Section 4.4. The second NVM area is used to maintain cache entries. The third, also the largest part of NVM area, is used for caching data blocks delivered by file system. This area is managed in a unit of 4KB block by default.

Tinca defines *cache entries* that are not only used for address mapping of caching but also for committing transactions. Figure 5 illustrates a cache entry that has 16 bytes for a cached data block. The leftmost byte currently has two flag bits. One is the **R** bit that stands for the block's *role*, which will be described in Section 4.3. The other flag bit is the **M** bit to label whether the cached data block has been modified ('1') or clean ('0').

The on-disk block number of a data block occupies 7 bytes in a cache entry. In the remainder 8 bytes of an entry, 4 bytes stand for the NVM block number that the data block was *previously* mapped to and the other 4 bytes are for the NVM block number the data block is *currently* mapped to. Tinca keeps two NVM block numbers for a data block in its cache entry to support rolling back the data to previous version in case of crash.

Conventional caching mechanisms organize cache entries in blocks. Tinca, on the other hand, manages them in a NVM-favored fine-grained way. In particular, as the size of a cache entry is 16B, the CPU can atomically modify an entry using a `cmpxchg16b` with the LOCK prefix. To facilitate `clflush` and `sfence`, cache entries are kept contiguous and cache line aligned. For instance, with 64B per CPU cache line, four NVM cache entries are stored and loaded together for CPU's reference. In addition, given a 16B-sized cache entry, the space needed to store all cache entries in the NVM is not significant. For example, an 8GB NVM cache requires $\frac{8GB}{4KB} \times 16B =$ 32MB for cache entries, which is just 0.4% of NVM capacity.

4.3 Role Switch on Writing a Block

Committing a transaction is composed of writing all data blocks in the transaction one by one. Before introducing how to commit a transaction, we present writing a block with *role switch* and *COW block write* for crash consistency in this section.

Role Switch Data blocks are written to NVM cache with block numbers of the file system area. In order to achieve crash consistency without writing such blocks twice, Tinca introduces a concept of *block role* for every cached block. A block has either a role of *log block* or *buffer block*, which is set as follows.

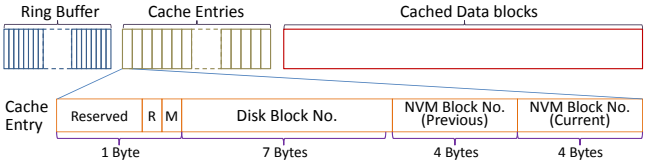


Figure 5: Tinca's NVM Cache Layout and Cache Entry

- A block in the ongoing committing transaction is labeled the role of *log block* temporally.
- On the completion of committing the entire transaction, every block in the transaction will be converted to *buffer block*.

Only blocks with a role of *buffer block* are allowed to be flushed to disk for cache replacement. The reason why a block with *log block* cannot be replaced will be explained in Section 4.6.

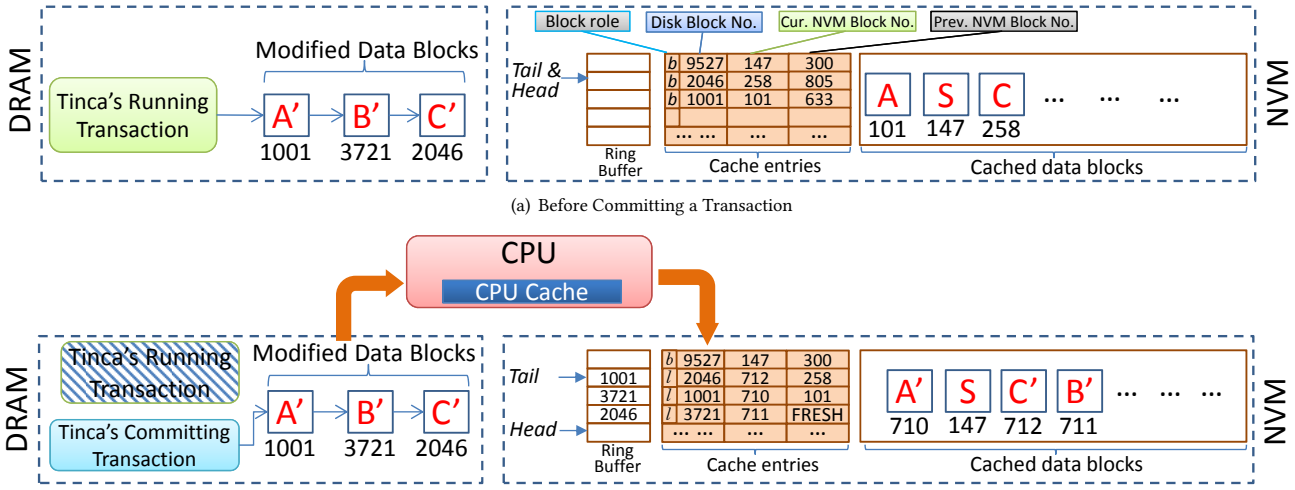
COW Block Write Writing a data block entails creating or updating a cache entry of NVM cache. If the block has been cached (write hit), Tinca writes the updated version in a COW way. First, Tinca writes the updated version in a newly-allocated NVM block. Second, Tinca records the locations of old version and updated version in the cache entry via a 16B atomic write followed by `clflush` and `sfence`. COW block write enables the recovery of a cached block with its previous version in case of crash. Keeping two versions for a write hit only lasts for committing a transaction that runs in a short period. The spatial overhead of maintaining two versions is also not significant. Relevant experiment will be presented in Section 5.4.3.

Writing a block may incur a cache miss. There is no previous version for a write miss. Tinca just creates a new cache entry where the previous NVM block number is set to be a special FRESH tag.

4.4 Committing a Transaction of Tinca

Tinca allows file system to coalesce multiple data blocks in a transaction. It maintains running transactions and committing transactions at runtime. A call of `tinca_commit` makes a running transaction be committing transaction. Tinca's transaction is lightweight compared to that of conventional journaling. One reason is the avoidance of double writes by role switch. The other reason is that Tinca leverages the byte-addressable NVM to commit a transaction. As mentioned, a conventional journaling file system uses metadata blocks to coordinate a transaction, like descriptor blocks to describe and start the transaction and a commit block to end the transaction. In contrast, Tinca uses a fine-grained ring buffer to do so. As shown in Figure 5, the ring buffer can be viewed as a contiguous array in which each element has 8B to store an on-disk block number for a data block. The ring buffer is used in a round-robin way with two pointers called *Head* and *Tail*. Before committing a transaction, both *Head* and *Tail* point to the same position in the ring buffer.

Figure 6 illustrates an example for the commit protocol of Tinca. As shown in Figure 6(a), before committing, both *Head* and *Tail* point to the same position of ring buffer. File system has initiated a running transaction by calling `tinca_init_txn` that links three data blocks in DRAM. `tinca_commit` triggers committing the transaction. Committing a transaction consists of writing every



(a) Before Committing a Transaction

(b) Committing a Transaction: 1. Running transaction converted to committing transaction; 2. Each block in the transaction committed (a. Write a data block with `clflush` to NVM; b. Update/create a cache entry; c. Write block number to the location *Head* points at; d. Move *Head* pointer by one); 3. Do role switches for all involved blocks; 4. Move *Tail* to be the same as *Head*

Figure 6: An Illustration of Committing a Transaction of Tinca

data block and moving *Head* and *Tail* pointers. A brief procedure of committing a transaction is summarized as follows.

(1) **Writing data block and cache entry** Tinca writes a data block, for example, 1001 with data *A'* in Figure 6(b), into NVM cache and initializes or updates the block's cache entry. Writing a data block for cache hit is done in the COW way, as indicated by cache entries in Figure 6(b).

(2) **Recording block number into ring buffer** After the cache entry is persisted, Tinca writes the block number 1001 into the position indicated by *Head* pointer via an 8B atomic memory write. `clflush` and `sfence` are executed to persist 1001 in the ring buffer.

(3) **Moving *Head* pointer** Tinca moves *Head* forward by one. In Figure 6(b), *Head* has been moved for three blocks.

(4) **Role switches** Tinca repeats (1)–(3) until no block is left in the transaction. Then, Tinca does a role switch for every block committed in the transaction. The role switch enables Tinca to avoid double writes. In Figure 6(b) the role switches are going to be performed.

(5) **Moving *Tail* pointer** At last Tinca sets *Tail* to be identical as *Head*. Moving and setting *Head* and *Tail* pointers are done via an 8B atomic write followed by `clflush` and `sfence` to make them durable in NVM. The atomic modification of *Tail* ends committing the transaction and a completion will be returned to the file system.

4.5 Crash Recovery of Tinca

Tinca persistently stores *Head* and *Tail* pointers in NVM and moves them to record block numbers committed in a transaction. *Head* and *Tail* pointers and the '**R**' (role) bits in cache entries play important roles in crash recovery for Tinca. On reboot after a crash, Tinca compares *Head* to *Tail*. If they are identical, there are two possibilities of crash occurrence. First, the crash has happened before a transaction. Second, the crash has happened when Tinca recorded the first committed block number but before moving *Head* pointer; however, whether the first block number has been persisted into ring buffer is uncertain. For both cases, a scan of all cache entries is necessitated. If no *log block* is found, all blocks are consistent and nothing needs to be done. If a cache entry with *log block* role is found, this block has to be revoked (undone) with the previous block number found in the cache entry and the cache entry itself needs to be modified also; if the previous NVM block is FRESH, both data and cache entry should be deleted from NVM cache.

If *Tail* is not the same as *Head*, the crash occurred during committing the transaction. All Tinca needs to do is scan the ring buffer from *Tail* pointer to revoke blocks that have been written in the transaction until *Head* pointer is reached.

4.6 Cache Replacement

Tinca caches for both write and read requests. It applies the *write-back* policy by default and maintains auxiliary structures in DRAM to facilitate caching. A hash table and a linked list are employed for two purposes: (1) accelerating the search with a disk block number for a cache entry and (2) assisting Tinca to select and replace the least-recently-used (LRU) data block to disk. Another structure is the free block monitor that traces NVM blocks that are not being used. These structures are not needed to be persistently stored in NVM as they can be reconstructed on the startup of system.

At runtime Tinca asks the free block monitor to find out a vacant block to satisfy an incoming allocation request. If no free block can be found, a victim block will be picked out for replacement.

Tinca's victim selection for replacement is defined as follows.

(1) The basic rule is the LRU policy: a cached data block that has the LRU data shall be evicted to make space.

(2) An additional rule is imposed in the interest of Tinca's transaction coordination: data blocks involved in the committing transaction are not allowed to be swapped out.

(a) A block that has the *log block* role accommodates the data being committed. It has not been stationary yet. Such a block is prevented from being replaced. If there

Table 2: Benchmarks Used to Evaluate Tinca and Classic

Benchmark			Read/Write Ratio	Request Size	Overall Dataset Size	Running Time	Description
Local Test	Fio		3/7, 5/5, 7/3	4KB	20GB	20 minutes	Varied ratios of mixed random write and read
	TPC-C		Typical TPC-C	Typical TPC-C	32GB	20 minutes	OLTP Workload issued by HammerDB to MySQL
Cluster Test	TeraGen		All Writes	100B per row	100GB	Until all data written	A generator that creates input data for TeraSort
	Filebench	Fileserver	1/2	16KB	51.2 GB	30 minutes	File server operating on a large number of files
		Webproxy	5/1	16KB	32GB	30 minutes	Web proxy server in the Internet
		Varmail	1/1	16KB	32GB	30 minutes	Email server operating on a large number of emails

are two copies of the data block involved in the committing transaction, neither one is allowed for replacement.

(b) On the completion of transiting from *log block* to *buffer block*, committed data blocks will turn to be the most-recently-used (MRU) in the LRU linked list. NVM blocks that hold these data's previous versions are reclaimed for future use.

The second rule is used to ensure NVM blocks involved in committing a transaction, no matter whether they contain the updated version or the previous version, will be kept in NVM cache. The profit of this rule is twofold. First, crash consistency is not impaired. Second, the performance can be improved. A block that has the previous version of some data being committed may be the LRU victim to be replaced. Flushing this version incurs an unnecessary disk write because the newly-committed version has to be flushed again in the future. Tinca yet keeps the previosu version in NVM cache until it is nullified when committing a transaction is completed. A disk write is hence saved.

5 PROTOTYPE AND EVALUATION

5.1 Prototype, Setup & Recoverability of Tinca

Prototype We have developed a prototype for Tinca. Since large NVM chips are unavailable, we take NVDIMM as an alternative. NVDIMM is a combination of DRAM and NAND flash memory and has the same read and write latencies as DRAM. In order to reflect the effect of real NVM technologies, we configure NVDIMM to simulate PCM by adding write/read delays (180ns/50ns) as suggested by previous work [7, 10, 12]. We have connected NVDIMM and DRAM to the memory bus. In implementation, the memory space provided by NVDIMM is reserved by the `memmap` option in the GRUB menu prior to booting Linux.

The entire package of Tinca is a pluggable Linux kernel module. Inside the module Tinca coordinates transactions, caches blocks and manages NVM space. It creates or modifies a cache entry using atomic `cmpxchg16b` with the LOCK prefix followed by `clflush` and `sfence`. To the upper-layer file system Tinca exports transactional primitives that are prototyped by altering JBD2's interfaces. JBD2's descriptor block and commit block are substituted by Tinca ring buffer and *Head* and *Tail* pointers. The default size of ring buffer is set to be 1MB. In particular, we have replaced JBD2's `start_this_handle` and `jbd2_journal_commit_transaction` using Tinca's `tinca_init_txn` and `tinca_commit`, respectively, to initiate and commit a transaction of multiple data blocks with their file system addresses. The checkpointing functionality is removed from JBD2. The failure recovery of JBD2 is also renovated with Tinca's recovery strategy. In brief, Tinca recovers from NVM cache.

The competitor we use to compare against Tinca has three layers. The top layer is original journaling file system Ext4 with JBD2. The middle layer is Flashcache [53] to act as a cache manager over NVM, which manages cache metadata in a block format and synchronously update them. The low layer is an NVM-based block device with `clflush` and `sfence` [7, 54]. The two implementations will be referenced as **Tinca** and **Classic**, respectively.

Setup The hardware of one local machine is as follows. The CPU is Intel Xeon E5-2640 (2.50GHz, 64B per CPU cache line) and only supports `clflush`. New instructions for cache line flush, such as `clflushopt` and `clwb`, are not available on our platform. But this does not affect the rationality and efficiency of our tests.

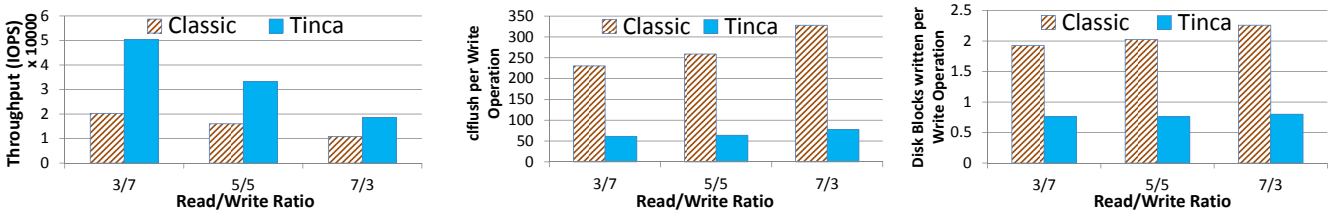
We have tested in both local machine and cluster environments. In the local machine, the DRAM main memory has 4GB while the capacity of NVM cache (NVDIMM) is 8GB. A 128GB SATA SSD is used as the underlying disk. The OS is Ubuntu Server 14.04 LTS with kernel 3.14.63 and GCC 4.9.2. The consistency level of both implementations is data consistency by which both file system metadata and data are consistent. The default cache mode is *writeback*.

As to the cluster platform, we have connected four storage nodes, each of which is with the same configuration as a local machine, via 10 Gigabit Ethernet. Table 2 lists characteristics of micro- and macro-benchmarks that have been used for a comprehensive evaluation of Tinca's performance in both local and cluster tests.

Evaluation Metrics In a test case with each benchmark we have run for three times and obtained the mean results. The performance metrics used to compare **Tinca** and **Classic** are as follows.

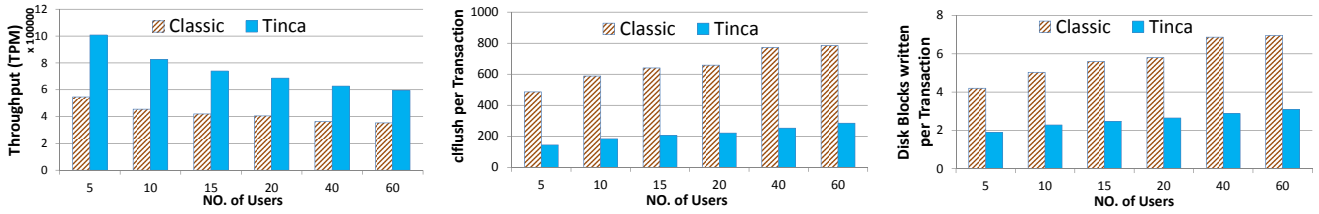
- *Throughput*: I/O operations per second (IOPS), file operations per second (OPs/s) and TPC-C transaction per minute (TPM).
- *Normalized quantity of* `clflush`: both **Tinca** and **Classic** need to write data into NVM cache via `clflush` in the NVM cache, we collected the number of `clflush` normalized against a write/file operation or TPC-C transaction.
- *Normalized quantity of Disk Writes*: both **Tinca** and **Classic** eventually replace and flush data to SSD. Thus, we recorded the number of disk blocks written per write/file operation or TPC-C transaction.

Recoverability Before testing its performance, we first validate Tinca's recoverability in a local machine. We set two scenarios of system failure: (1) unexpectedly plugging out the power cable, and (2) suddenly killing **Tinca**'s process. We randomly did either failure for multiple times. Each time **Tinca** can recover and crash consistency of the system is never impaired.



(a) Write Throughput (IOPS) (b) `clflush` per Write Operation (c) Disk Blocks per Write Operation

Figure 7: A Comparison between Classic and Tinca on Fio Micro-benchmark



(a) Throughput (TPM) (b) `clflush` per TPC-C Transaction (c) Disk Blocks per TPC-C Transaction

Figure 8: A Comparison between Classic and Tinca on TPC-C Workload

5.2 Testing on Local Machine

5.2.1 *Fio.*

Fio is a prevalent micro-benchmark [50]. We used Fio to generate a file of 20GB and varied the read/write ratio to be 3/7, 5/5, and 7/3, as listed in Table 2. We ran Fio for 20 minutes with **Classic** and **Tinca**, respectively. The write throughputs in IOPS are presented in Figure 7(a). The write IOPS would decrease for **Classic** and **Tinca** as the write percentage rises from 30% to 70% in all operations. However, **Tinca** consistently outperforms **Classic** with much higher throughputs. In particular, **Tinca**'s write IOPS is 2.5×, 2.1×, and 1.7× that of **Classic** on three read/write ratios, respectively.

The performance gap between **Classic** and **Tinca** are caused by their two major differences: 1) **Classic** has to ask JBD2 to write the same data twice for crash consistency but **Tinca** writes data once, and 2) **Classic** employs Flashcache that synchronously updates cache entries in a block format while **Tinca** does so in a fine-grained manner. The numbers of `clflush` and disk writes per write operation are presented in Figure 7(b) and 7(c), respectively. As for NVM cache, **Tinca** reduced up to 73.4%, 75.4%, and 76.3% cache lines flushes than **Classic** on three write/read ratios, respectively. At the disk level, **Tinca** wrote 60.6%, 62.6%, and 64.6% fewer disk blocks per write operation, respectively, compared against **Classic**. The reductions of cache line flushes and disk writes enable **Tinca** to achieve much higher performance than **Classic**.

5.2.2 *TPC-C.*

TPC-C is a representative online transaction processing (OLTP) workload [55]. We run MySQL [56] on top of **Classic** and **Tinca**, respectively. HammerDB [57] is used to generate TPC-C workloads. We set up 350 warehouses of about 32GB size. HammerDB reports the throughput with transaction per minute (TPM) at runtime. Note that here the *transaction* here refers to TPC-C transaction. We varied the number of users to be 5, 10, 15, 20, 40, and 60, so as to impose more and more burdens on MySQL server. Two points can be obtained from Figure 8(a). First, **Tinca** dramatically and consistently outperforms **Classic** with much higher throughputs. In particular, the mean TPM for **Tinca** is 1.8× and 1.7×, respectively, as much as that for **Classic** with 5 and 60 users. Second, from 5 users to 60 users, both **Tinca** and **Classic** drop with decreasing throughputs; however, **Tinca** decreases less than **Classic**. For example, between 5 users and 60 users, **Tinca** yields 35.3% lower throughput while for **Classic** the drop is 41.0%.

The number of `clflush` in NVM cache and the number of disk blocks written to SSD per TPC-C transaction are shown in Figure 8(b) and 8(c), respectively. As **Tinca** avoids double writes and manipulates cache entries in a fine-grained way, it costs much fewer NVM writes with much fewer `clflush` than **Classic** to handle a TPC-C transaction. As shown in Figure 8(b), the numbers of `clflush` per TPC-C transaction for **Tinca** are just 29.8% and 36.2% of that for **Classic** with 5 users and 60 users, respectively.

The elimination of double writes helps **Tinca** to conduct much fewer disk I/Os than **Classic**, as told by Figure 8(c). With 5 users, **Classic** needs to write about 4.2 blocks to SSD per TPC-C transaction while **Tinca** does so with 1.9 blocks. With 60 users, **Classic** writes 7.0 blocks while **Tinca** writes 3.0 blocks. Therefore, the quantity of disk I/Os are significantly reduced. The observations on reductions of NVM and SSD write I/Os confirm that the elimination of double writes and fine-grained cache management help Tinca greatly promote performance of a database server.

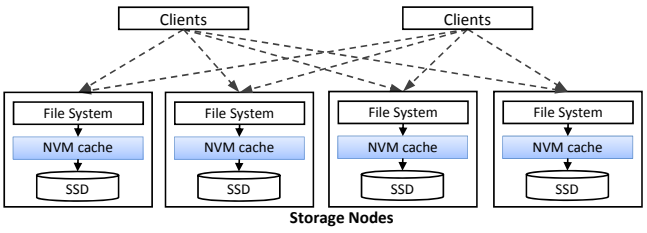
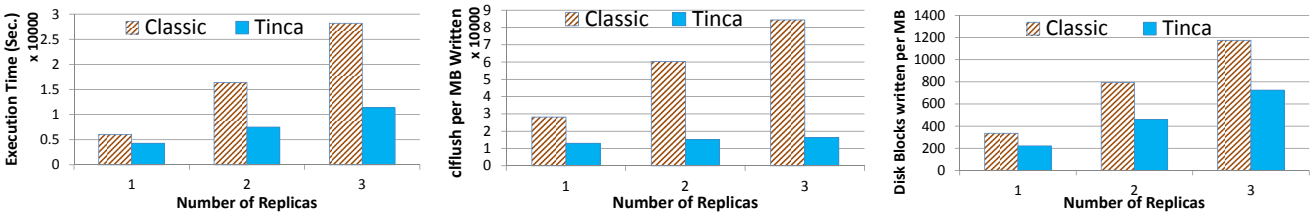
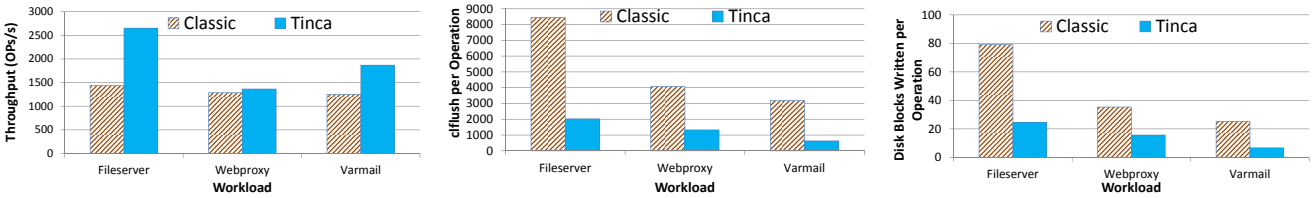


Figure 9: An Illustration of Storage Cluster with NVM Cache



(a) Execution Time (Second) (b) `clflush` per MB Written (c) Disk Blocks per MB Written

Figure 10: A Comparison between Classic and Tinca on TeraGen within HDFS



(a) Throughput (OPs/s) (b) `clflush` per File Operation (c) Disk Blocks per File Operation

Figure 11: A Comparison between Classic and Tinca on Filebench Workloads in GlusterFS

5.3 Testing in Cluster Environments

HPC and Big Data applications may require massive throughputs from distributed file systems that employ file system and caching mechanism as local storage manager for data nodes. We integrated **Tinca** and **Classic** into local storage managers of two prevalent distributed file systems, i.e., HDFS and GlusterFS. Figure 9 illustrates the architecture of clustered storage with NVM cache. We used micro- and macro-benchmarks to evaluate the performances of **Tinca** and **Classic** in the clustered storage platforms.

5.3.1 HDFS with TeraGen.

We set up four data nodes and one name node with HDFS and varied the number of replicas to be 1, 2, and 3. In HDFS we ran the micro-benchmark named TeraGen [58] that generates data sets that would be used as input for TeraSort. We let TeraGen generate 100GB data set to HDFS cluster. The results in execution time reported by TeraGen for **Classic** and **Tinca** are shown in Figure 10(a), from which two observations can be obtained. First, in the cluster environment, **Tinca** still significantly outperforms **Classic**. For example, with 3 replicas, **Tinca** cost 59.7% less time than **Classic** with TeraGen. Second, as the number of replicas increases, the performance gap between **Classic** and **Tinca** goes wider, since **Tinca** spent 29.0% and 54.1% less time than **Tinca** with 1 and 2 replicas, respectively. With more replicas, HDFS has to write more data into NVM cache and disk. The more data written, the more evident effect of avoiding double writes by **Tinca**.

We have also measured the numbers of `clflush` and disk blocks written per MB generated by TeraGen. The results are presented in Figure 10(b) and 10(c), respectively. In both diagrams, **Tinca** yielded much fewer `clflush` and disk writes compared to **Classic**. For example, with 3 replicas, **Tinca** flushed 80.7% fewer cache lines to NVM than **Classic**, and the former wrote 38.3% fewer blocks to SSD than the latter. These results confirm that **Tinca** can greatly boost the performance of a clustered storage platform with the reduction of double writes and metadata updates.

5.3.2 GlusterFS with Filebench.

We further ran the macro-benchmark Filebench [49] on GlusterFS. We setup GlusterFS on the same hardware platform of 4 nodes as described in Section 5.3.1. For our evaluation, we have selected three typical workloads. They are Fileserver, Webproxy, and Varmail. Their descriptions are shown in Table 2. In general, Webproxy is a read-intensive workload while the remainders are write-intensive.

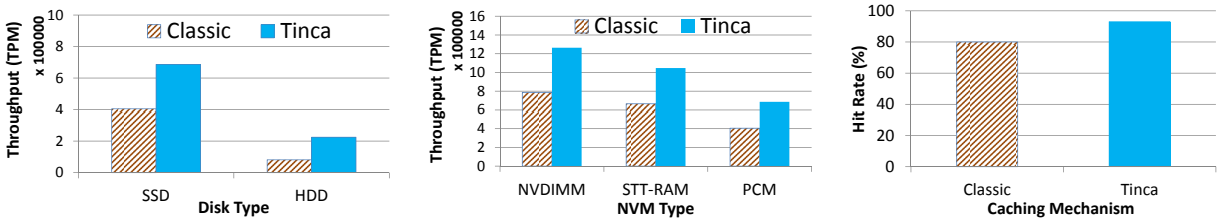
Filebench reports file operations per second (OPs/s) as throughput. We fixed the number of replicas in GlusterFS as 2. Figure 11(a) shows the OPs/s of **Classic** and **Tinca**. It is evident that the throughput of **Tinca** is 1.8× and 1.5× as much as that of **Classic** on Filserver and Varmail, respectively. As shown in Table 2, write operations contribute half of all operations in these two workloads. Without writing the same data for two times, **Tinca** is able to significantly promote performance. Even for the read-intensive Webproxy workload, **Tinca** yields 20.1% higher throughput than **Classic**.

Figure 11(b) and 11(c) respectively present the quantity of `clflush` per file operation and the number of disk blocks written to SSD per file operation for **Tinca** and **Classic**. Obviously reflected from the two diagrams, **Tinca** dramatically reduces writes to both NVM and disk. This observation explains why it can significantly outperform **Classic** and in turn confirms the effectiveness of Tinca's strategy.

5.4 Discussion

5.4.1 Impact of Disk Storage and NVM Cache Media.

By running TPC-C workload with 20 users we have traced the impact of storage and NVM media. We first kept the NVM as simulated PCM and varied the underlying disk to be HDD and SSD. Figure 12(a) shows throughputs of **Classic** and **Tinca** in TPM on SSD and HDD, respectively. It is undoubted that both **Classic** and **Tinca** would decrease in performance with a slower storage device. The throughput of **Classic** on HDD drops by about 5× compared to that on SSD while the throughput of **Tinca** drops by about 3×.



(a) The Impact of Disk (b) The Impact of NVM (c) Cache Write Hit Rate of **Classic** and **Tinca**

Figure 12: The Impact of Disk and NVM types and the Cache Write Hit Rate with TPC-C Workload

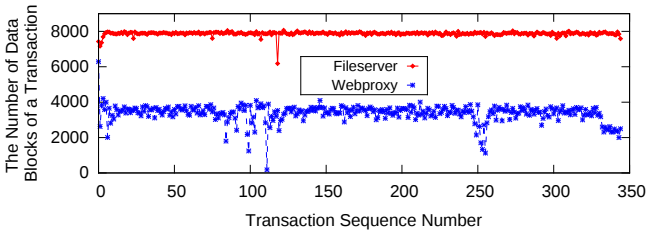


Figure 13: The Number of Data Blocks per Transaction with Two Typical Workloads

Hence, **Tinca** is more efficient with the slower HDD as the performance gap between **Tinca** and **Classic** increases from 1.7× to 2.8× as with SSD and HDD, respectively. The widening gap is accredited to **Tinca**'s avoidance of double writes, leveraging which **Tinca** is able to reduce disk writes. As a result, for an HDD with a longer write latency, the reduction of disk writes brings in more substantial performance gains.

We also tested with different NVM technologies. The default NVM technology we simulate is PCM. We added write/read delays (50ns/50ns) onto the original NVDIMM to emulate STT-RAM [7]. We still ran TPC-C with 20 users using SSD as the underlying disk under three NVM technologies. The throughputs are presented in Figure 12(b). With faster NVDIMM and STT-RAM, the throughputs of **Tinca** and **Classic** both increase. The impact of double writes on faster disk cache is relaxed a bit as the performance gap between **Tinca** and **Classic** drops slightly from 1.7× to 1.6× from PCM to NVDIMM and STT-RAM.

5.4.2 *Cache Write Hit Rate.*

Because of space limitation, we only present the cache write hit rates of **Classic** and **Tinca** along with running TPC-C with 20 users on simulated PCM and SSD, as shown in Figure 12(c). The hit rate of **Classic** is 80% while it is 93% for **Tinca**. As **Tinca** does not write the same data twice to NVM cache, the cache space is more efficiently utilized. The increment of hit rate is hence achieved.

5.4.3 *Spatial Overhead of COW Block Write.*

Tinca uses COW to update cached block if a write hit happens, so one data block may take up two NVM blocks in committing a transaction. We have measured such spatial cost for COW block write. We monitored the number of blocks committed in one transaction when running Fileserver and Webproxy, respectively. As told by Figure 13, Fileserver generally writes 2× blocks than Webproxy. However, although Fileserver incurs larger amount of writes, the number of data blocks per transaction is about 8,000. Given such a number, in the worst case (100% write hit) where every block needs two NVM blocks in a transaction, extra spatial cost is 8,000×4KB≈32MB. It is just 0.4% space of an 8GB NVM cache.

5.4.4 *A Comparison between Tinca and UBJ.*

Let us do a comparison between Tinca and UBJ [31]. Three major differences exist between them. First, Tinca and UBJ have different architectures. Tinca is designed as a persistent cache between main memory (DRAM) and disk while UBJ is designed for whole NVM-based main memory atop disk. Journaling of UBJ is in file system layer while Tinca offloads the journaling functionality from file system to disk cache. Second, Tinca and UBJ have different impacts on updating data. UBJ first commits-in-place a data block in NVM by freezing the block, and then checkpoints it to disk. On updating a frozen data block in buffer cache, UBJ cannot overwrite the original copy. It must do a memory copy (`memcpy`) on the block and update out of place. This additional act of `memcpy` occurs on the critical path of writing data by applications. Tinca updates data in place in the DRAM buffer cache, which is more efficient. Third, Tinca and UBJ rely on different transaction protocols. As mentioned, UBJ has commit-in-place and checkpoint. The unit of commit and checkpoint for UBJ is a transaction that may include thousands of blocks. Checkpointing data to disk may be frequently triggered for UBJ to free NVM space. Worse, checkpointing in UBJ takes longer time for multiple blocks in a large transaction to be written. By contrast, Tinca only conducts commit without checkpoint and writing data to disk is done by Tinca for cache replacement.

6 RELATED WORK

Developing devices with transactional supports have been investigated, especially with flash-based SSDs [32, 59–62]. TxFlash [59] leverages the copy-on-write nature of NAND flash memory and exports transactional interface. X-FTL [61] pays attention to database-aware transactional support and LightTx [60] tries to reduce the transaction tracking cost while providing flexibility for better performance. TxCache [32] employs an internal disk cache in an SSD to provide embedded transactional interfaces for file systems to utilize. However, all these algorithms are for flash-based SSDs that must be customized and have not considered an utilization of NVM disk cache for more efficient transactional supports. Tinca differs from them in that Tinca is a modular *software* package while previous designs depend on customized hardware/firmware supports in an SSD or even needs to modify SATA interface. In addition, Tinca is not limited to SSD as it suits both HDD and SSD with an NVM disk

cache. Therefore, Tinca is more portable and efficient since it can be easily positioned in real environments.

Intel provides applications with user-space NVM Library (NVML) like `libpmemobj` for transactional supports [63]. Tinca differs from NVML in two major aspects. First, Tinca is an integrated design of persistent cache with transaction support. Second, Tinca has a key difference from NVML in designing transaction support: NVML (`libmemobj`) employs an undo log for transaction support. It writes an object to the undo log before updating the object and incurs double writes. COW used by Tinca does not cause double writes.

Using a flash-based SSD cache to promote performance of HDDs has been studied [16, 20–22]. Li et. al. [20] incorporates adjustable data compression and deduplication into SSD cache. Arteaga et. al. [22] employs an on-demand caching solution to meet demands of virtual machines. Byte-addressable NVM has been used to accelerate legacy storage systems as well [2, 6, 64, 65]. Fan et. al. [64] have developed a new management policy on using the NVM as a cache to flash-based SSDs. Kim et. al. [6] have studied profit and feasibility on using the PCM to make a standalone PCM-based SSD or equip PCM device as an external disk cache.

Managing NVM with a persistent in-memory file systems has been explored also [3, 7, 10, 11, 13, 66]. Narayanan et. al. [66] proposed to keep the whole system status with NVM when power failure occurs. Zhao et. al. [67] proposed Kiln that leverages the last level non-volatile cache to support in-place update for NVM-based persistent memory. Liu et. al. [13] designed a unified architecture of working memory and persistent store with NVM. Shivaram et. al. [68] and Yang et. al. [12] revisited basic tree structures in NVM. All these indicate the promising potential of NVM to reshape computer systems in the future.

7 CONCLUSION

We propose Tinca to make a transactional NVM disk cache that leverages byte-addressable NVM for high performance and crash consistency. We first investigate performance penalties of write amplifications in NVM cache caused by journaling file system and cache metadata consistency. We hence use a method of role switch to avoid double writes in consistently persisting data into NVM cache. We design fine-grained cache metadata that Tinca utilizes to write a data block to NVM in a COW way. Based on the role switch and COW block write, we define a commit protocol for Tinca to coalesce and write multiple data blocks in one transaction. Tinca is able to provide transactional primitives for file system to use. We have built a prototype for Tinca and done extensive experiments in local and cluster platforms. Experiments show that, with identical data consistency, Ext4 with Tinca can yield up to 2.5× performance compared to original Ext4 with state-of-the-art caching mechanism.

ACKNOWLEDGMENTS

We are grateful to our shepherd, Professor Weikuan Yu, and anonymous reviewers for their helpful comments.

REFERENCES

[1] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 57:1–57:12, New York, NY, USA, 2009. ACM.

[2] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[4] Myoungsoo Jung, Ellis H. Wilson, III, Wonil Choi, John Shalf, Hasan Metin Aktulga, Chao Yang, Erik Saule, Umit V. Catalyurek, and Mahmut Kandemir. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 75:1–75:11, New York, NY, USA, 2013. ACM.

[5] Doe Hyun Yoon, Jichuan Chang, Robert S. Schreiber, and Norman P. Jouppi. Practical nonvolatile multilevel-cell phase change memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 21:1–21:12, New York, NY, USA, 2013. ACM.

[6] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, Santa Clara, CA, 2014. USENIX.

[7] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[8] Xinning Wang, Bin Wang, Zhuo Liu, and Weikuan Yu. Preserving row buffer locality for PCM wear-leveling under massive parallelism. In *Proceedings of the 2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '15, pages 198–207, Washington, DC, USA, 2015. IEEE Computer Society.

[9] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[10] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP '09*, pages 133–146, New York, NY, USA, 2009. ACM.

[12] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

[13] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified working memory and persistent store architecture. *SIGARCH Comput. Archit. News*, 42(1):455–470, February 2014.

[14] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on NVM. In *Mass Storage Systems and Technologies, 2016 32nd Symposium on*, MSST '16, pages 1–12. IEEE, May 2016.

[15] Timothy Bisson, Scott A. Brandt, and Darrell D. E. Long. NVCache: Increasing the effectiveness of disk spin-down algorithms with caching. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, MASCOTS '06, pages 422–432, Washington, DC, USA, 2006. IEEE.

[16] Qing Yang and Jin Ren. I-CASH: Intelligently coupled array of SSD and HDD. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 278–289, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[18] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 945–956, Washington, DC, USA, 2012. IEEE Computer Society.

[19] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with Bonfire. In *Proceedings of the*

11th USENIX Conference on File and Storage Technologies, FAST'13, pages 59–72, Berkeley, CA, USA, 2013. USENIX Association.

[20] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 501–512, Berkeley, CA, USA, 2014. USENIX.

[21] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Enabling cost-effective flash based caching with an array of commodity SSDs. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 63–74, New York, NY, USA, 2015. ACM.

[22] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand flash cache management for cloud computing. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 355–369, Berkeley, CA, USA, 2016. USENIX Association.

[23] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash caching on the storage client. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 127–138, Berkeley, CA, USA, 2013. USENIX Association.

[24] Kent Overstreet. Linux bcache. http://bcache.evilpiepirate.org/.

[25] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, ATC '96, pages 1–14, Berkeley, CA, USA, 1996. USENIX Association.

[26] David A Solomon. Inside Windows NT (Microsoft programming series), 1998.

[27] Stephen Tweedie. Ext3, journaling filesystem. In *Proceedings of the Linux Symposium*, pages 24–29, 2000.

[28] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.

[29] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 105–120, Berkeley, CA, USA, 2005. USENIX Association.

[30] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[31] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 73–80, Berkeley, CA, USA, 2013. USENIX Association.

[32] Youyou Lu, Jiwu Shu, and Peng Zhu. TxCache: Transactional cache using byte-addressable non-volatile memories in SSDs. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2014 IEEE*, pages 1–6, Aug 2014.

[33] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[34] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[35] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.

[36] Zhenyu Sun, Xiuyuan Bi, Hai (Helen) Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 329–338, New York, NY, USA, 2011. ACM.

[37] Cong Xu, Dimin Niu, Naveen Muralimanohar, Norman P. Jouppi, and Yuan Xie. Understanding the trade-offs in multi-level cell ReRAM memory design. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 108:1–108:6, New York, NY, USA, 2013. ACM.

[38] Micron and Intel. 3D XPoint technology. http://www.micron.com/about/innovations/3d-xpoint-technology.

[39] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[40] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming. Technical Report HPL-2012-236, HP Laboratories, December 2012.

[41] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, and Mingdi Xue. How to be consistent with persistent memory? an evaluation approach. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 186–194, Aug 2015.

[42] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 174–181, New York, NY, USA, 2015. ACM.

[43] Oracle Corporation. Lustre. http://lustre.org/, December 2016.

[44] The Apache Software Foundation (ASF). Hadoop distributed file system (HDFS). https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, March 2017.

[45] Red Hat Inc. GlusterFS. https://www.gluster.org/, April 2017.

[46] Fraunhofer Center. BeeGFS. http://www.beegfs.com, December 2016.

[47] Ceph Community. Ceph file system. http://docs.ceph.com/docs/master/cephfs/, January 2017.

[48] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 101–116, Berkeley, CA, USA, 2012. USENIX Association.

[49] Spencer Shepler, Eric Kustarz, and Andrew Wilson. Filebench. http://filebench.sourceforge.net.

[50] Flexible IO (Fio) Tester. Fio. http://freecode.com/projects/fio, March 2015.

[51] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 802–811, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 31–44, Berkeley, CA, USA, 2013. USENIX Association.

[53] Mohan Srinivasan and Paul Saab. Flashcache: A write back block cache for linux. https://github.com/facebook/flashcache/, September 2015.

[54] Feng Chen, Michael P Mesnier, and Scott Hahn. A protected block device for persistent memory. In *Mass Storage Systems and Technologies, 2014 30th Symposium on*, MSST '14, pages 1–12. IEEE, June 2014.

[55] Transaction Processing Performance Council. TPC-C benchmark. http://www.tpc.org/tpcc/, February 2010.

[56] MySQL. MySQL: The world's most popular open source database. https://www.mysql.com/, October 2015.

[57] HammerDB. Hammerdb project. http://www.hammerdb.com/, July 2015.

[58] The Apache Software Foundation (ASF). Teragen and terasort. https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html, March 2017.

[59] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[60] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 115–122, Oct 2013.

[61] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for sqlite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 97–108, New York, NY, USA, 2013. ACM.

[62] Wei Shi, Dongsheng Wang, Zhanye Wang, and Dapeng Ju. Möbius: A high performance transactional SSD with rich primitives. In *Mass Storage Systems and Technologies, 2014 30th Symposium on*, MSST '14, pages 1–11. IEEE, 2014.

[63] Intel Corporation. Intel NVM library. http://pmem.io/nvml/libpmem/, July 2017.

[64] Ziqi Fan, David H.C. Du, and D. Voigt. H-ARC: A non-volatile memory based cache policy for solid state drives. In *Mass Storage Systems and Technologies, 2014 30th Symposium on*, MSST '14, pages 13–23, June 2014.

[65] Qingsong Wei, Jianxi Chen, and Cheng Chen. Accelerating file system metadata access with byte-addressable non-volatile memory. *ACM Transactions on Storage*, 11(3):12:1–12:28, July 2015.

[66] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[67] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[68] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–15, Berkeley, CA, USA, 2011. USENIX.