Log-Structured Non-Volatile Main Memory

Qingda Hu, ***Tsinghua University;*** **Jinglei Ren and Anirudh Badam,** ***Microsoft Research;***
Jiwu Shu, ***Tsinghua University;*** **Thomas Moscibroda,** ***Microsoft Research***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Log-Structured Non-Volatile Main Memory

Qingda Hu* Jinglei Ren Anirudh Badam Jiwu Shu* Thomas Moscibroda
*Tsinghua University Microsoft Research

Abstract

Emerging non-volatile main memory (NVMM) unlocks the performance potential of applications by storing persistent data in the *main memory*. Such applications require a *lightweight* persistent transactional memory (PTM) system, instead of a *heavyweight* filesystem or database, to have fast access to data. In a PTM system, the memory usage, both capacity and bandwidth, plays a key role in dictating performance and efficiency. Existing memory management mechanisms for PTMs generate high memory fragmentation, high write traffic and a large number of persist barriers, since data is first written to a log and then to the main data store.

In this paper, we present a *log-structured* NVMM system that not only maintains NVMM in a compact manner but also reduces the write traffic and the number of persist barriers needed for executing transactions. All data allocations and modifications are appended to the log which becomes the location of the data. Further, we address a unique challenge of log-structured memory management by designing a *tree*-based address translation mechanism where *access granularities* are flexible and different from *allocation granularities*. Our results show that the new system enjoys up to 89.9% higher transaction throughput and up to 82.8% lower write traffic than a traditional PTM system.

1 Introduction

Emerging byte-addressable non-volatile main memory (NVMM), e.g., 3D XPoint [23], PCM [43, 27], STT-RAM [3, 25] and ReRAM [2], enables persistent data to be stored in *main memory*. This leads to an architecture where applications directly access persistent data via CPU load/store instructions [50, 10, 49, 37, 44, 18, 41, 55]. Such an architecture lowers latency not only due to the significantly higher performance of NVMM compared to SSDs, but also due to the fact that the system software is removed from the critical path of persistent-data accesses [36, 13, 9, 55]. Applications that use NVMM typically employ a lightweight persistent transactional memory (PTM) system [50, 10, 22, 56, 34, 9, 18, 24], instead of a traditional file system or database, to have fast access to NVMM data.

Memory usage, both capacity and bandwidth, is crucial for the performance and efficiency of PTM systems. DRAM-style memory management used by existing PTM systems to manage NVMM leads to a high amount of fragmentation that can cause wastage of over 50% space [46]. Moreover, existing transactional mechanisms used by PTM systems lead to excessive write traffic as they require all new data to be written twice – once to the log, and once to the main data region, referred to as *home space* of data. The redundant writes not only increase memory bandwidth usage but also wear out the NVMM device faster. Further, these writes need to be persisted using expensive barriers in a synchronous manner which increase the latency of transactions.

In this paper, we present a new *log-structured* memory management model for NVMM systems. This model eliminates *dichotomy* of NVMM data in the home space and a separate log area. We unify the home space and the log area by organizing the whole NVMM solely in the form of logs, which also act as the home space. Our design effectively reduces fragmentation, incorporates wear-leveling, and optimizes for the write traffic and persist barriers. Fragmentation is minimal because memory allocation becomes an immediate append to the end of a log, and freed up areas can be moved and consolidated [45, 46] to further reduce fragmentation. Besides, NVMM bandwidth consumption, write wear and the number of persist barriers are reduced because there is no need to write data *separately* to both the traditional home space and the log.

Applications using our system view NVMM in the same way as the traditional systems, but a runtime address mapping mechanism is employed to translate application addresses to log offsets. We refer to the applications' view of NVMM as the *virtual* home space. Such address mappings are fully cached in DRAM, and can be consistently restored from the log after a crash.

Another *key contribution* of this work is the design and implementation of a practical tree data structure for the home to log address mapping in our system. While log-structured approaches have been explored in different domains, such as filesystems [45, 52], databases [48, 46, 4] and object stores [31, 46], log-structured NVMM faces a *unique challenge* of address mapping overhead. Unlike existing log-structured systems, we need to present a flat address space where allocation granularities are *not* the same as access granularities.

A data structure that can support creation of mappings at access time as opposed to allocation time is required.

*Work done while Q. Hu was an intern at Microsoft Research. Affiliated with Dept. of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST).

This is because memory stores can target *arbitrary* addresses and lengths that may not be indicated at allocation time. We show in this paper that a *tree structure* is well suited for such a requirement. Meanwhile, NVMM is orders of magnitude faster than SSDs, so that address mapping performance could become a bottleneck if not designed well. For SSDs, data access latencies dwarf address translation overhead, but that is not the case with NVMM. Hence, we revisit the address mapping issue of log-structured designs for NVMM systems.

A naive tree data structure requires $O(\log n)$ operations per memory access which can be prohibitive when n is large. Moreover, trees require expensive balancing operations to achieve such time complexity. We design key optimizations to a tree structure for log-structure NVMM to reduce address translation overhead: **(1)** *Two-layer mapping*. The whole home space is first divided into static fixed-length *partitions* so that data can be routed to such a partition (or more partitions) in $O(1)$ time. In this way, the average number of nodes in a partition-local tree is much smaller than a huge tree covering the whole address space. **(2)** *Skip-list trees*. We use the skip list [42] for second-layer trees. The main benefit is that they probabilistically balance at insert time to avoid rebalancing operations, which are costly and largely impair parallelism. **(3)** *Group update*. If consecutive writes target contiguous addresses, we merge them and update the tree only once. **(4)** *Tree node cache*. We observe that memory accesses have locality so that caching recently visited tree nodes can avoid many full tree lookup paths starting from the root node.

We also present mechanisms to control the overhead of log cleaning needed for compaction, and speed up the recovery process. NVMM logs can be processed in parallel on recovery, which helps rebuild address mappings for 10 GB NVMM in 3.0 seconds.

Overall, we make the following contributions:

- A new log-structured design to eliminate the dichotomy between the data and the transactional log for PTMs. We identify the crucial difference between existing log-structured systems and the kind needed for NVMMs where access granularities are not identical to allocation granularities.
- A novel tree-based address mapping mechanism that meets the above requirement. To the best of our knowledge, we are the first to demonstrate the practicality of employing such a well optimized tree structure in a log-structured NVMM system.
- An implementation of the above ideas by modifying TinySTM [16]. Under various workloads, log-structured NVMM achieves 55.3% more throughput and 72.2% less write wear than a traditional PTM on average, when the usage of NVMM is over 90% and the log cleaning overhead takes place.

2 Background and Motivation

Current PTMs typically derive their memory management design from that for DRAM. Data is referenced using load and store instructions on native virtual memory offsets, and memory allocations are managed by an allocator such as Hoard [6] adopted by Mnemosyne [50], and jemalloc [15] adopted by Intel's NVML [22] and Oracle's NVM Direct [39]. However, the following problems arise in such systems.

Fragmentation of NVMM space. There are two sources of fragmentation in a traditional memory allocator [17, 15]. First is *internal fragmentation*. Take Intel's NVML [22] for example. It aligns any NVMM allocation size to 64 B. If 65 B of NVMM is requested, NVML shall effectively allocate 128 B, including 63 B internal fragmentation. Second is *external fragmentation*. Suppose a 64-B block is freed but has surrounding blocks in use, then it cannot serve any request beyond 64 B. External fragmentation is severe if allocation sizes vary [38]. Experiments [46] have demonstrated that fragmentation can take over 50% of all memory under management. This issue is more critical for NVMM because it holds data for a long term even across reboots.

Garbage collection, in a managed language runtime such as Java or C#, is capable of changing allocated addresses. It can reduce fragmentation but involves object reference analysis and process pauses [20]. Since NVMM is slower and larger than DRAM, the cost of object reference analysis and pauses will be prohibitive.

In contrast, a log-structured approach easily avoids internal fragmentation because new allocation is compactly appended to the log end. It absorbs external fragmentation by moving allocated data and consolidating free spaces without the need to pause the process.

Excessive NVMM write traffic and barriers. NVMM has limitations in bandwidth and endurance [28] ($10^4 - 10^9$ P/E cycles compared to DRAM's 10^{15} cycles). However, to maintain crash consistency, all NVMM writes must first be logged by PTM at a separate location. Such logging entails redundant NVMM write traffic and extra wear, compared to naive writing.

Figure 1 shows how a log-structured approach can reduce the write traffic and also the number of flushes for a representative transaction. By the pseudo function `map_address`, all addresses within the area are mapped to a new location in the log. Such mapping only involves DRAM writes which are fast and incur no wear on NVMM. This approach saves extra NVMM writes and costly CPU flushes/persist barriers.

Furthermore, the traditional PTM systems use the NVMM bus less efficiently than the log-structured approach, because updates to the home space tend to be sparse and hence have poor cacheline coverage . This

Pseudo source code:

```
void tx_update_title(employee emp, title new_title) {
  tx_begin {
    emp.title = new_title;
  } tx_end;
}
```

Transaction system behaviors: *logging PTM* vs. *log-structured NVMM*

```
in_log_title = append_to_undo_log(emp.title);
flush(in_log_title);
emp.title = new_title; // extra NVMM write
flush(emp.title); // one more flush
```

```
in_log_title = append_to_nvmm_log(new_title);
flush(in_log_title);
map_address(emp->title, in_log_title);
```

Figure 1: In a traditional PTM, objects have to be first logged and the log has to be persisted in NVM using a CPU flush before the transaction can edit the objects. Another CPU flush is needed after the edits complete. In log-structured NVMM, one flush is enough. Since the log entry becomes the new location of data, the extra write is eliminated.

leads to more bus bandwidth consumption when compared to sequentially appending them to the log.

A unique challenge in log-structured NVMM. The challenge of *tree*-based address mapping is a unique one for log-structured NVMM. It has not been seen in existing log-structured systems. Those systems manage data in a form of well-defined elements such as blocks in a filesystem [45, 52], tuples in a database [48, 4] or objects in a key-value store [31, 46], where allocation granularities are the same as access granularities. Such well defined access granularities facilitate a high performance design. For instance, an in-memory hash table can be employed to map elements to their locations in the log, which offers $O(1)$ lookup. In addition, a bloom filter can be applied to improve mapping/index performance in the case that a slow search path exists (e.g., log-structured merge trees [48]).

Unfortunately, such a convenience is missing for NVMM systems. There is no concept of data elements or IDs in bare memory. It is hard to define one in systems that employ a flat address space where accesses can be targeted at any offset with any length. Restrcting block/object-granular accesses lacks flexibility and incurs high costs [51, 16]. Simply setting a fixed and small block size (e.g., tens of bytes) is not viable either, because the metadata to maintain such blocks can be prohibitively large [30, 19]. Furthermore, NVMM is orders of magnitude faster than SSDs, so the address mapping overhead, though traditionally negligible, now stands out. Therefore, we design a more flexible but highly performant scheme, which fragments the address space on demand based on the executed store instructions rather than defining the granularity statically or at data allocation time.

3 Design

This section describes the design of log-structured NVMM (LSNVMM), a user-space library for accessing and managing NVMM.

3.1 Overview

The high-level architecture of LSNVMM is shown in Figure 2. From bottom up, LSNVMM uses DAX [32] through a filesystem that allows *direct* access to physical NVMM device via a memory map. In LSNVMM, the NVMM region is organized into logs (§3.3), and an address mapping mechanism translates virtual home-space addresses to log positions (§3.2). Applications access the NVMM region via our library that interposes all the memory accesses to the region using the address mapping mechanism.

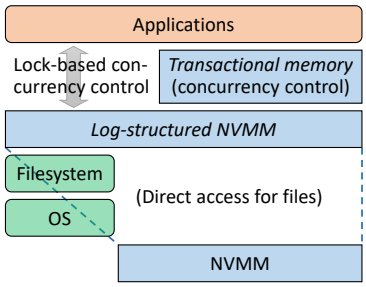


Figure 2: The architecture and system stack of log-structured NVMM.

Interface. Our library offers two main functionalities. One is memory management, with semantics similar to that of C library: `pmalloc` and `pfree` for NVMM allocation and deallocation, respectively[1]. The other functionality of our library is the transaction abstraction that provides crash-consistent data persistence. *All* NVMM data operations are performed via this abstraction, referred to as an NVMM transaction. Within an NVMM transaction, memory loads and stores are instrumented at compile time and treated differently: all stores of the transaction are persisted atomically to the log on NVMM; every load address has to be translated to a

[1]Note that our current design assumes that the persistent region is fixed to a static base address [50, 35]. Doing so enables use of native pointers that remain valid across crashes and reboots. However, special pointer types [10] can be supported easily.

proper position in the log to access the data. Concurrency control of data operations is left to an upper-layer transactional memory (TM) system. It is also possible to use explicit locks for such concurrency control.

Recovery. To achieve efficient address translation, address mappings are stored in DRAM. On a normal process shutdown, we compact the in-DRAM address mappings and other necessary metadata, and flush them to NVMM, so that they can be quickly restored when the process restarts. However, if a system crash happens, the DRAM data is lost. Therefore, we have to rebuild the in-DRAM data structures. To speed up this process, the recovery is performed using thread-level parallelism (more details in §4.5).

3.2 Address Mapping

Using our address mapping mechanism, applications interact with NVMM in much the same way as DRAM to build data structures. They need not change their memory access model that uses flexible regular virtual memory addresses and pointers. However, they have to adopt the transaction interface to make atomic changes to the data structures similar to existing PTM systems. We refer to addresses in applications' view as *home addresses*, and log positions that are hidden from applications as *log addresses*.

We use a tree structure to maintain mappings from home addresses to log addresses. Logically, one node in the tree holds a pair {home address, length} denoting an area in the home space, and the log address that the area is mapped to. The rationale for using a tree instead of a hash table is that, in flat address space based systems, allocation granularities are not identical to access granularities. For instance, an application may allocate a large structure using `pmalloc` but only read/write a small portion of that within transactions. Therefore, we need address translation support for arbitrary accesses that are not aligned with allocated objects.

The efficiency of address mapping is crucial for our system. The latency of traditional log-structured systems is dominated by the disk/SSD latency of data accesses. Also, the granularity of such data accesses is large (e.g., the block size of 512 B) and the frequency is low. However, in our case, NVMM is much faster and more frequently accessed in granularities as small as a few bytes. Hence, it warrants careful design of the address mapping. The time complexity of an operation on the tree is $O(\log n)$. We use several optimizations to reduce the practical cost of such an operation. Figure 3 depicts main data structures to support these optimizations as described below.

Two layers of mapping. The average cost of a tree operation is proportional to the tree height, so our first optimization targets at largely reducing the tree height. This

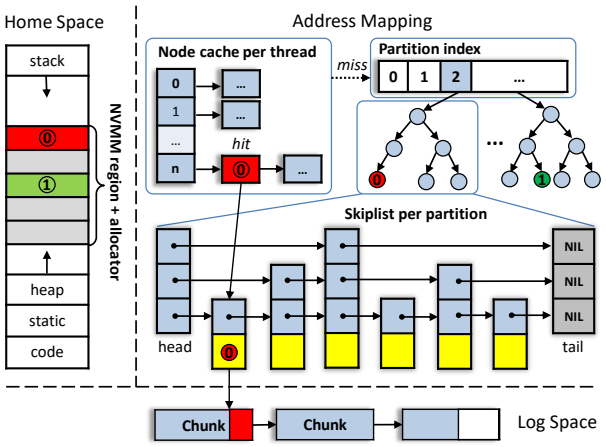


Figure 3: The spaces and address mappings in LSNVMM. Access to ⓪ is a hit in the tree node cache, and access to ① is a miss. Both examples are single addresses, but a tree node contains a range of addresses and a range lookup across nodes is supported as well.

can be realized if a huge tree is split into numerous small ones. We do so by having two layers of address mapping. In the first layer, we divide the home space into fixed-length partitions, so that a home address can be simply divided by the partition length, costing as low as one CPU cycle, to determine which partition the address locates in. In the second layer, each partition holds a small tree for further address lookup (Figure 3). Our approach can reduce the tree height by several times. With real world workloads, this optimization improves transaction throughput by 39.6% on average (§5.2).

Group update. Opportunistically merging tree nodes is another way to further reduce the number of nodes and thus the height of a tree. When two sibling nodes contain contiguous home addresses *and* map to contiguous log addresses, they can be merged. Spatially local writes within a transaction can exploit this optimization. Within each NVMM transaction, we first buffer all writes in DRAM, and combine those with contiguous home addresses on transaction commit. A group of combined writes is appended to the log and the address mapping tree is updated the minimal number of times. Overall, this optimization realizes 42.3% transaction throughput improvement according to our evaluation (§5.2).

Skip lists and locking. We choose the skip list [42], a probabilistic alternative to balanced trees, as our tree data structure (Figure 3). The main reason for our choice is that, while supporting $O(\log n)$ operations on average, the skip list does not need a complex rebalancing operation as a strictly balanced tree such as B-tree does.

Such an optimization is crucial for multi-threaded scenarios. A typical balanced tree requires a readers-writer

lock to protect concurrent operations[2]. Lock contention due to heavy reads and writes can deteriorate throughput of such systems. In contrast, by leveraging skip lists, we get rid of locking for read-only operations. Particularly, an update of the skip list involves only simple pointer manipulations on singly linked lists. Taking advantage of CPU's atomic word write (aligned 64 bits for x86), such an update is implemented in a way that is atomic to lock-free read-only operations. By avoiding such lock contention, we can see 48.9% higher transaction throughput with four threads in our experiments (§5.2).

Tree node caches. We equip each working thread with a thread-local cache that stores recently accessed home addresses and pointers to their nodes in the trees (Figure 3). When the program accesses an address, our library first searches the cache. If it is hit, the library directly gets the pointer to the tree node that contains the requested address mapping; otherwise, a full tree lookup is necessary and the resulting node is added to the cache. Such a caching mechanism is effective because of inherent temporal and spatial locality among memory accesses. As our experiments show, some memory areas are hot and frequently accessed, and memory accesses tend to cluster within 64 B areas. The hit ratio is 92.2% on average, and introduction of tree node caches leads to 30.1% increase in transaction throughput on average (§5.2).

We tweak a regular hash table design to meet a special requirement of our tree node cache. That is, once a node is cached, addresses within its mapped area tend to be a cache hit. A plain hash table does not give such a feature as cached addresses are randomly distributed. For example, a node for a 64 B area starting at 0x1000 is cached. If an access to the address 0x1008 falls into a different bucket, it would lose the chance to be checked with this node and hence be a miss. To solve the issue, we deliberately increase certain collision by using set-associativity. Based on the observation above, we try to route addresses within a 64 B scope to the same bucket so that nearby addresses can be checked with chained tree nodes that may cover them. To realize that, we pick high-order bits of an address as its hash value. Consequently, sequential addresses have a good chance of falling into one bucket.

3.3 NVMM Organization

The goal of our NVMM organization is to allow each thread to allocate NVMM with minimal overhead. Towards that end, the NVMM region is physically organized into static *chunks*, atop which we build logical logs. Multiple chunks can be linked into a list. We choose a relatively small chunk size (e.g., 32 KB), because typical NVMM writes are small; moreover, an individual chunk with a small size can be more quickly cleaned and recycled in an incremental manner.

Chunks help reduce contention among the multiple threads. We maintain a global pool of free chunks, and each thread has its own list(s) of chunks in use. A thread is allowed to buffer some free chunks when it requests one from the global pool, or after it obtains them from local log cleaning. This can avoid frequent manipulation of the global pool and its lock contention.

3.4 Log Structure

A log in the NVMM region consists of a list of chunks. Multiple logs coexist in our system. It is different from a conventional disk-based log-structure system which tends to have a single log per disk because the disk has only one disk header and sequential access is the first priority. With fast random access instead, NVMM warrants a different design, which favors thread-level parallelism by using thread-local logs. Furthermore, each thread has multiple logs to improve log cleaning efficiency, as we describe later in this section. LSNVMM employs a number of log cleaners to collect free space accumulated in chunks. The free spaces come from `pfree` operations or old data that has been updated. We use a background thread to run a cleaner.

Log entry. A log entry holds two kinds of metadata. First, a mapping for a modified or allocated memory area. When a log cleaner scans the chunk, it checks liveness of each log entry by looking up the home address from the address mapping tree. Second, a tombstone for each freed area. A tombstone is never accessed within transactions, but used on the recovery path to filter out freed areas. Atop log entries, we build transactions. A transaction consists of all log entries that it produces, by memory stores and (de)allocations.

Cleaning policy. The log cleaner moves sparse live data from several chunks to a new chunk in a compact manner, and recycles the cleaned chunks. Chunks with the amount of live data below a threshold (20% by default in our setup) are selected for cleaning.

We design three optimizations for log cleaning. **(1)** *Fast cleaning*: When all log entries in a chunk are stale, the chunk can be safely reclaimed. This can be done fast because we only need to modify a few list pointers to move the chunk to a free chunk list, without data copying. **(2)** *Separate logs*: We observe that memory stores always have better locality than memory allocations. It implies that mixing them in one log may increase the log cleaning cost and decrease the chance of fast cleaning. So we design separate logs for each thread, the *update log* serving memory stores, the *allocation log* serving memory allocations and the *deallocation log* storing only tombstones. **(3)** *Parallel cleaning*: In order to have sufficient log cleaning throughput, we

[2]There are carefully crafted lock-free balanced tree designs [8, 14] but they involve extra complexity and overhead. In contrast, our approach is simple and performs well in practice.

perform log cleaning with multiple background threads for different chunks.

4 Implementation

This section describes the implementation of LSNVMM. We start with the home space management mechanisms in §4.1, then elaborate log space management in §4.2 and address mapping between the two spaces in §4.3. Log cleaning and recovery procedures are described in §4.4 and §4.5, respectively.

4.1 Home Space Management

Memory allocation and access are two main functionalities of home space management. We draw upon existing implementation of transactional memory systems to realize such functionalities[3]. But we add persistence to transactional memory: (1) necessary allocation metadata is stored in NVMM so that the home address space can be rebuilt after a crash, and (2) committed transactions are stored in NVMM so that data updates are persistent. Next, we detail the underlying mechanisms.

Home space allocation. Considering that the 64-bit home address space is virtual and sufficiently large, fragmentation is not a severe issue there. Thus, we choose current memory allocators Hoard [6] and dlmalloc [26] to implement *home space* allocation. Hoard serves memory allocations smaller than 8 KB, while dlmalloc deals with larger ones [50].

The state of both allocators is consistently rebuilt upon crashes using metdata stored with data and therefore, no runtime effort is spent in ensuring persistence of the state. Take Hoard for example. It organizes home space into superblocks, and each superblock serves allocation requests of a certain size (e.g., a 8 KB superblock contains an array of 16 B allocations). The metadata of superblocks (location and allocation size) is stored in NVMM. With such information, we simply rely on the logs to infer allocation state after crashes. Therefore, home-space allocations do not incur any persistent operations.

Transactional memory. Applications' access to home-space data is protected by transactions. Intel STM compiler [1] is used to instrument regular C/C++ code with transaction annotations. Programmers place the keyword `__tm_atomic` and a pair of braces to specify the scope of a transaction. The compiler automatically generates calls into our transaction system when a transaction begins, issues memory loads and stores, and commits.

We employ TinySTM [16], a lightweight software transactional memory implementation, to intercept these calls and implement concurrency control of transactions. Each transaction holds a temporary private *write set* containing all written values and their addresses, which are not visible to concurrent transactions. When a transaction allocates memory, the system quickly allocates the requested size in the home space, and returns its home address. After that, all writes to the newly allocated space are buffered in the volatile write set.

Allocated memory, writes to old data are all persisted into logs when the transaction is committed. Likewise, deallocations are also logged to ensure that memory does not leak. A TinySTM transaction may receive memory writes to both volatile regions and NVMM. The LSNVMM library takes the responsibility to filter out writes to volatile memory and persist those to NVMM in a crash consistent manner when the transaction is committed. The group update optimization is performed to merge NVMM writes that have contiguous home addresses. Afterwards, a single log entry is generated for each NVMM write and flushed to logs in NVMM. Then each NVMM write obtains its log address, and the library inserts into global address mapping trees the mappings from home addresses to log addresses.

4.2 Log Space Management

From top down, the hierarchy of log storage is as follows: (1) a log is stored in a number of fixed-length *chunks*; (2) within one or more chunks, transactions that constitute the log are stored in *transaction blocks*; (3) within a transaction block, memory allocations and updates of the transaction are stored in *log entries*. We now describe these components in a bottom up order.

Log entry. Each log entry has a header and data. The header consists of (1) a 47-bit *home address* to record the start home address of the data, (2) one bit to denote whether the entry is a tombstone, and (3) a 16-bit *size* to record the data length. 47 bits are enough to hold a home address because we record the offset of the address in the NVMM region. Immediately after the header is the data whose location is its *log address*. This entry structure is used for both update and allocation logs.

Transaction block. A group of log entries belonging to a transaction make the payload of a transaction block. A preamble contains the following fields: (1) A 64-bit *version number* to record the commit time of the transaction. In our implementation, it is the monotonically increasing, globally unique timestamp generated by TinySTM for each transaction[4]. (2) A 48-bit *peer pointer* that points to another transaction block (e.g., in a different chunk as the current chunk is filled up), or in an allocation log if the current log is an update log, or vice versa. As a result, all blocks of a transaction form a

[3] LSNVMM is not bound to transactional memory. We choose the interface because it is easy to use for applications.

[4] It is an optimization to reuse the timestamp, but LSNVMM is not necessarily bound to any TM implementation. We can also simply use a global atomic counter to generate the version number.

cyclic singly-linked list. (3) A 16-bit *entry number* to record the number of log entries in the current transaction block. If the number is not enough to count all entries of a transaction, more block(s) can be linked to the current block. (4) A 32-bit *checksum* using CRC32 error-detecting code, which is calculated against the whole transaction block.

Since a logical transaction may contain multiple transaction blocks across both update and allocation logs, consistency among the blocks becomes an issue. We have to handle the issue in two cases. The first case is when a crash interrupts a transaction commit. LSNVMM can detect this case by checking the checksum of each transaction block on recovery, and discard the transaction if any of its block is invalid or lost. The second case is when a transaction block is moved to another chunk due to log cleaning. As a result, peer pointers referencing moved blocks are no longer valid. However, such inconsistency brings no problem as long as the containing transactions are safely committed, because the peer pointers are only used for detecting uncommitted transactions as in the first case. Therefore, we only need to divert log cleaning from log ends that contain uncommitted transactions.

Chunk. The payload of a chunk is a sequence of transaction blocks that make part of a log. Chunks are doubly linked by their headers. Besides, the header holds a flag to denote whether the chunk belongs to an update log or an allocation log. If a transaction block contains a log entry larger than the remaining space of a chunk, the entry can be split into more, and stored in linked peer blocks in other chunks.

4.3 Skip List

An address mapping tree is implemented as a concurrent skip list. By using insertion as an example, we show how our skip list operates in a concurrent manner. In a skip list, insertion of a node involves inserting the node to a number of levels. For each level, the insertion is identical to that of a singly linked list, which can be atomically realized by feat of atomic pointer updates. We do insertion from the bottom level up. Once the node is inserted to the bottom level, the insertion is effective. Inserting to upper levels only influences lookup performance. So, the insertion is logically atomic to concurrent reads.

While reads are lock-free, any tree structure update (e.g., insert or delete) has to hold a lock controlling the whole tree, because concurrent updates may corrupt each other. But we can still maintain high update concurrency, thanks to the large number of such trees in our design.

The tree node cache also needs a careful concurrency control. We have to check if the hit node still holds the requested home address, because it is possible the node has been removed and recycled. Accordingly, we check the home address of a node twice – before and after reading the log address of the node. If both checks match, the log address must be valid.

4.4 Log Cleaning

When memory utilization is beyond a threshold, a few background cleaner threads begin to work, in parallel with transaction threads. Cleaning steps are as follows: (1) A set of victim chunks are identified according to the policy in §3.3. For each victim chunk, a scan of all its log entries is performed to determine liveness of the data in each entry by checking its latest version in the address mapping tree, v_t. If v_t is higher than the current transaction version, the entry is discarded. (2) For a transaction block that has live entries left, the preamble is recalculated (entry number and checksum), and the entire block appended to a new chunk. (3) For the moved transaction block, a quasi TinySTM transaction is run to update global mappings with the new log addresses of the live entries. The quasi transaction is just for enforcing concurrency control. (4) After all transaction blocks are moved out of a victim chunk, the chunk is reclaimed by adding it to the global free chunk pool.

4.5 Recovery

Our recovery works in two phases to maximize thread parallelism in a manner similar to map-reduce. In the first phase, we dispatch all log chunks to the recovery threads for parallel processing. The main task of each thread is to scan the assigned chunks and group valid log entries by the partition of their home addresses. After this phase, each thread holds an array indexed by the home partition, and each element of the array has a list of log entries belonging to the partition. Note that this temporary log entry structure only contains pointers to data in NVMM and necessary metadata (version number).

In the second phase, each recovery thread takes charge of different home partitions, and the task is to replay log entries belonging to the partitions. To do so, the above lists of log entries are shuffled among threads, so that each thread holds the lists whose partitions are in the charge of the thread. Then, for each partition, the single thread in charge sorts all log entries of the partition by their home address and version number, then pick up entries with latest versions and insert their address mappings to the global address mapping tree for that partition. The approach, similar to map-reduce, avoids most thread contention.

5 Evaluation

To evaluate the performance of log-structure NVMM, we answer three questions as follows.

- *How effective are the individual optimizations we design for LSNVMM*? (§5.2)

- *How does LSNVMM perform against traditional PTM systems*? (§5.3)
- *What are the costs of log cleaning, recovery, and DRAM footprint*? (§5.4)

5.1 Experiment Setup

All the experiments are performed on a computer with 8-core Intel Xeon CPU E5-2637 v3 (3.5 GHz) and 64 GB DRAM, running 64-bit Linux kernel version 4.2.3. All results are average of five runs.

NVM simulation. As real NVMM products are not available yet, we use a simulation method akin to that in Mnemosyne [50]. We focus on effects of slow NVMM writes instead of reads, as many prior works do [35, 9, 18, 40], because the read latency of NVMM is similar to DRAM and most memory reads are effectively served by CPU caches. For a standalone NVMM write required to be immediately persisted, we introduce an extra latency. For sequential NVMM writes that are executed together, we consider both write latency and bandwidth of NVMM. The added delay is the max of the above write latency and *total write size/NVMM bandwidth*. By default, we set the write latency to 500 ns and the sustainable write bandwidth to 1 GB/s. We implement any delay by a loop reading the CPU timestamp counter (TSC) until required time has elapsed.

Benchmarks. We run five transactional benchmarks atop our systems for evaluation. The benchmarks cover both commonly used data structures and a real application: *SPS* randomly swaps elements in a large array; *RBTree*, *B+Tree* and *HashTable (HT)* perform operations on a red-black tree, a B+ tree and a hash table, respectively; *KVStore* runs a key-value store, Tokyo Cabinet [21].

For benchmarks BTree, B+Tree, HashTable and KVStore, we perform two workloads with different access patterns: the *insert* workload (Ins) inserts a number of key-value pairs, where keys are uniformly random; the *update* workload (Upd) looks up a key, and deletes it if it is found or inserts one otherwise. Keys of these pairs follow the Zipfian distribution [5, 11] so that 90% updates happen on 15% of the data. In all workloads, the value size is 128 B by default unless otherwise noted. The total number of elements/pairs in each benchmark is 10 million, resulting in 2∼4 GB of logical NVMM footprint.

5.2 Effect of Optimizations

We demonstrate the effect of *every* optimization proposed in §3.2. Comparing the library against itself provides valuable reference for other systems/implementations as such a control experiment reveals what benefit each mechanism can bring.

Evaluated systems. We add optimizations one by one to the address mapping structure, resulting in four implementations as below.

- *Base* is the baseline using a global, single skip list for whole-space address mapping.
- *2L* enhances Base with two-layer mapping. The home space is divided into 4-KB partitions, and each partition is served by a skip list for address mapping.
- *2L-GU* enhances 2L by performing group update.
- *2L-GU-C* adds thread-local tree node caches with FIFO replacement. Each cache is up to 4 M entries.

At last, we show results of LSNVMM, which is more optimized for multiple threads. 2L-GU-C uses a readers-writer lock per partition to protect a skip list from concurrency issues, while LSNVMM avoids locking for read-only operations on a skip list. So far, all optimizations are incorporated. In this experiment, we leave out log cleaning which is orthogonal to these comparisons.

Results. Figure 4 shows performance of the four implementations running the benchmarks. We make four observations. (1) 2L constantly outperforms Base for all workloads, by 39.6% on average, due to two-layer mapping. (2) 2L-GU performs 42.3% better than 2L on average, due to group update. (3) 2L-GU-C improves transaction throughputs by 30.1% on average, compared to 2L-GU, thanks to the tree node caches. (4) Overall, the above optimizations show strong performance in various benchmarks/workloads, achieving up to 268.6% (157.9% on average) performance improvement over the baseline system.

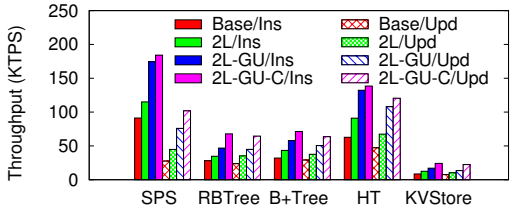


Figure 4: Transaction throughputs of the benchmarks with different optimizations, in a single thread.

Particularly, to give direct evidence of the effect of tree node caches, we plot average cache hit ratios under different benchmarks in Figure 5. A tree node cache achieves 92.2% hit ratio on average, which leads to significant performance improvement in all benchmarks.

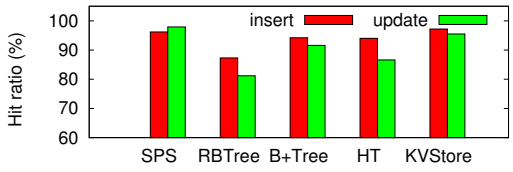


Figure 5: Access hit ratios of tree node caches under different benchmarks/workloads.

At last, our multi-thread optimization is justified by comparing 2L-GU-C and LSNVMM running the update

workload of a multi-threaded version of the data structure benchmarks, as shown in Figure 6. Removing lock overhead from read-only operations, LSNVMM achieves good scalability, and provides 48.9% higher throughput than 2L-GU-C when running four threads.

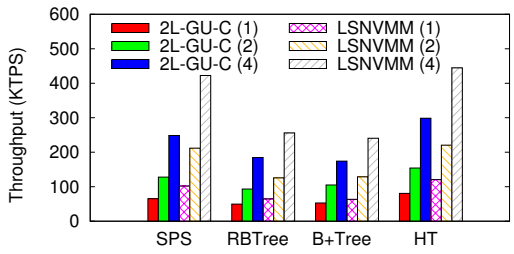


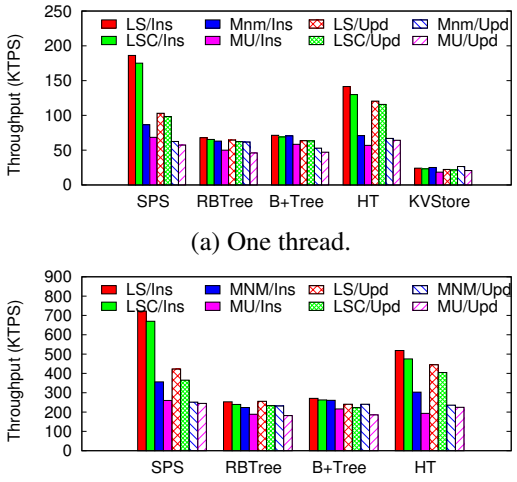
Figure 6: Multi-threaded throughputs of data structure benchmarks. "(n)" indicates the number of threads.

5.3 Comparison to Current Systems

Evaluated systems. We compare LSNVMM (LS) to redo and undo logging in the traditional memory management. In the same way as LSNVMM, both logging systems integrate with TinySTM [16]. Particularly, *Mnemosyne* (Mnm) [50] is the combination of redo logging and TinySTM with traditional memory management, and we also make *Mnmsyn-Undo* (MU) by replacing the redo logging mechanism in Mnemosyne with undo logging. Moreover, we deliberately introduce cleaning overhead to LSNVMM in *LSNVMM-Cleaning* (LSC), which triggers cleaning of chunks with over 50% stale data every around 1000 transactions.

Performance. We show performance results of the four PTM systems running the benchmarks. From Figure 7 (a), we observe that LSNVMM outperforms Mnemosyne and Mnmsyn-Undo by 37.3% and 66.1% with one thread on average, respectively. Especially for HashTable and SPS, LSNVMM achieves 89.6% and 89.9% (118.2% and 125.9%) speedup beyond Mnemosyne (Mnmsyn-Undo), respectively. These two benchmarks turn out to issue less memory loads than others. In contrast, LSNVMM does not perform well with KVStore running the update workload, mainly because it has intensive memory loads. As for scalability, Figure 7 (b) shows the performance of the PTM systems running the benchmarks in four threads. We can see that LSNVMM scales well. It performs 44.7% and 80.8% better than Mnemosyne and Mnmsyn-Undo on average, respectively. Finally, log cleaning incurs minimal overhead in this setting. Compared to LSNVMM without log cleaning, LSNVMM-Cleaning reduces the throughput of benchmarks by 4.1% and 7.8%, with one thread and four threads, respectively. More evaluation of log cleaning follows in §5.4.1.

In conclusion, LSNVMM remarkably outperforms logging PTMs even with log cleaning overhead, and shows scalability with multiple threads. LSNVMM is especially suitable for write-intensive workloads.



(a) One thread.

(b) Four threads.

Figure 7: Transaction throughputs of the benchmarks with different memory management systems.

NVMM write traffic and wear. We calculate NVMM *write traffic*, i.e., the cache line size multiplied by the total number of cache lines written back to NVMM. This metric reflects the NVMM bandwidth consumption. Among the write traffic, only modified data actually wears NVMM [57, 12], we estimate NVMM wear in terms of total *dirty bytes* ever written to NVMM. Figure 8 shows that part in breakdown of write traffic. We make two observations. (1) LSNVMM saves 82.8% and 82.0% write traffic of Mnemosyne and Mnmsyn-Undo on average, respectively. Besides the fact that redo/undo logging logically writes twice what LSNVMM does, we can clearly see the influence of cache line granularity. As home-space updates in Mnemosyne and Mnmsyn-Undo are typically sparse and fine-grained, they waste lots of NVMM traffic on flushing *entire* cache lines. (2) LSNVMM reduces dirty bytes by 80.1% and 65.1% compared to Mnemosyne and Mnmsyn-Undo on average, respectively. Thanks to the group update technique, LSNVMM merges a large number of sequential and repeated writes. On the contrary, Mnemosyne persists every write of the transaction in the log, even if it can be merged or coalesced with others.

NVMM fragmentation. In this experiment, we test different memory allocators under three typical workloads [46] that emulate variation of data value sizes. All workloads consist of two phases with different individual allocation sizes. W1 first allocates collectively 1 GB in randomly 100 - 150 bytes, and then repeats so in randomly 200 - 250 bytes. W2 is different with W1 only in that it frees 90% of the memory allocated in the first

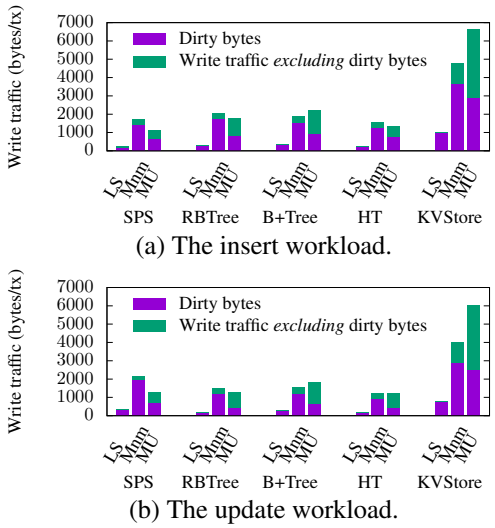


Figure 8: NVMM write traffic and wear of different memory management systems running the benchmarks (in a single thread).

phase before it goes to the second phase. W3 has the same behavior as W2 except that its individual allocation size in the first phase is random 1,000 - 2,000 bytes and in the second phase random 1,500 - 2,500 bytes.

Figure 9 depicts the results. We make two observations. (1) Typical DRAM-oriented memory allocators hardly manage memory efficiently in these workloads. Mnemosyne (Hoard) produces 25.3% memory fragmentation on average, and NVML (jemalloc) produces 35.0%. In contrast, LSNVMM keeps it as low as 4.5% by virtue of log cleaning. (2) The memory fragmentation of LSNVMM is inversely proportional to the allocation size, because each allocation has its own *metadata* cost. For example, LSNVMM incurs 7.3% fragmentation in W1 but only 0.6% in W3.

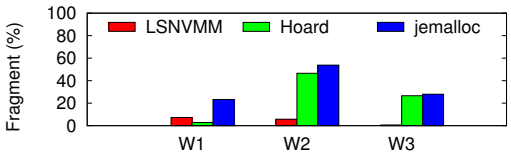


Figure 9: NVMM fragmentation ratios of LSNVMM and two other representative traditional memory allocators, Hoard [6] adopted by Mnemosyne [50], and jemalloc [15] adopted by Intel's NVML [22].

5.4 Log-Induced Costs

5.4.1 Log Cleaning

We first evaluate the effect of separate logs on fast cleaning (§3.4). Figure 10 depicts the amount of log data that is reclaimed by fast cleaning as the number of update operations increases. In the experiment, we firstly insert 10 million elements to the corresponding benchmarks. We make two observations from this figure. (1) Beyond initial 10 million updates, the fast cleaning can effectively clean around more than 200 MB memory per million updates. (2) The separate log design can clean more chunks than the baseline. Their gap is bigger in the RBTree benchmark, because it has more clustered memory stores than HashTable so that a separate update log is apt to fast cleaning.

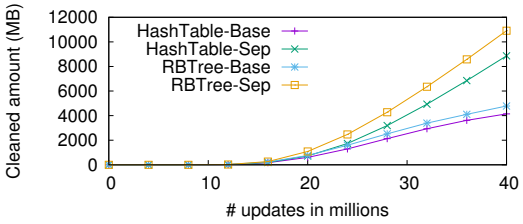


Figure 10: Fast cleaning performance with baseline ("Base") or separate logs ("Sep") with a random 1 KB update workload on HashTable or RBTree.

When all cleaning overhead walks in, Figure 11 shows the resulting performance of the benchmark as well as the throughput of the cleaner. In the experiment, we preload B+Tree to occupy a certain fraction of NVMM, and then run the update workload with four working threads and two cleaning threads. We test two cases where the value size is 128 B and 1 KB, respectively. We draw a major conclusion from this figure: LSNVMM does not lose much performance under high NVMM pressure. The performance degradation due to cleaning was 8% or less, even at 90% memory utilization.

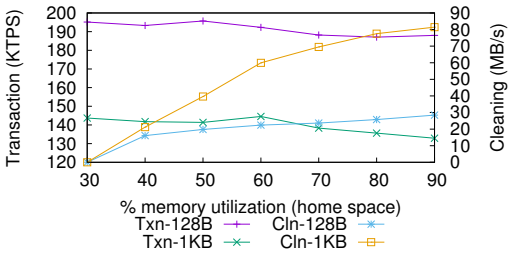


Figure 11: Transaction ("txn") and cleaning ("cln") throughputs of the B+Tree benchmark with random key distribution and different value sizes (128B vs. 1KB) as a function of memory utilization.

5.4.2 Recovery

Figure 12 shows the required time to recover from a 10 GB of logs in NVMM. We rebuild the whole LSNVMM in multiple threads. We make two observations from this figure: (1) The recovery process quickly speeds up with more threads. For 128 B values, LSNVMM needs 19.2 seconds to recover in one thread,

but only 3.0 seconds in eight threads. (2) The recovery latency is inversely proportional to the data allocation size, because the number of address mappings decreases as the allocation size increases.

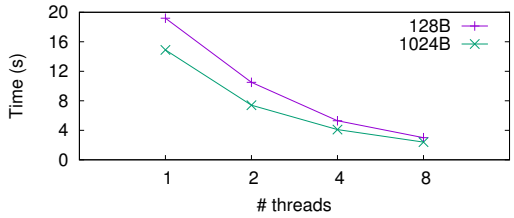


Figure 12: Recovery time of 10-GB NVMM logs, with different numbers of threads and different value sizes (128B vs. 1KB).

5.4.3 DRAM footprint

We evaluate the DRAM footprints using the real application KVStore under the insert workload with different value sizes. Figure 13 illustrates the amount of DRAM required. It is around 16.9% of NVMM when the value size is as small as 128 B, and drops quickly as the value size increases.

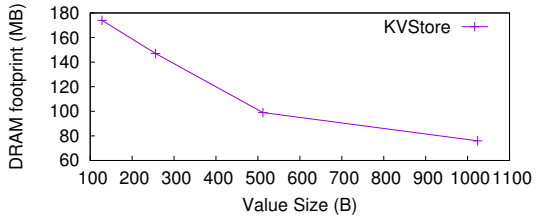


Figure 13: DRAM footprint of the address mapping structures and thread cache in KVStore for 1 GB NVMM data as a function of the value size.

6 Related Work

Persistent memory systems. They can be classified into three categories by their interfaces. One category is PTM. For example, Mnemosyne [50], SoftWrAP [18] and DudeTM [33] are redo logging based PTMs, while NV-Heaps [10], NVML [22] and DCT [24] are undo logging based ones. Our work is built on many PTM techniques, but follows a different, log-structured way to address the memory management issue.

The second category provides data structure interfaces, such as CDDS [49] and NV-Tree [53]. Their interfaces to applications are not as flexible as transactions. The third category is software transparent. WSP [37] and ThyNVM [44] are two representatives. They either have a strong assumption on hardware or involve advanced hardware features. In contrast, LSNVMM is a general solution and requires no customized hardware.

Memory allocators. Makalu [7] and nvm_malloc [47] are NVMM allocators that aim at collecting garbage in a failure-safe manner. WAlloc [54] proposes a wear-aware memory allocator to improve the wear leveling. These works address other aspects of memory management, while we focus on the memory fragmentation problem.

RAMCloud [46] shares the same goal as our work to reduce memory fragmentation. It also uses a log-structured approach. But it is a key-value store of well-defined data objects, without the need for a tree-based address mapping mechanism as in LSNVMM. LSNVMM supports general transactions for arbitrary data.

Log-structured systems. The log-structure approach was early designed in LFS [45], which buffers random writes in DRAM and makes best use of sequential I/O of hard disk drives. F2FS [29] proposes a well optimized file system on flash storage devices, which adopts separate metadata and data logs, and uses adaptive logging to avoid frequent garbage collection. It is similar to our separate log design. NOVA [52] is a file system optimized for hybrid memory systems, providing strong consistency guarantees. It maintains independent logs for each inode to improve scalability. Some databases [4] implement log-structured data management, and take advantage of NVMM to simplify traditional DBMS. Overall, the log-structured approach is widely used in those systems, but their designs hardly apply to LSNVMM whose unique challenge is tree-based address translation as discussed in §2.

7 Conclusion

The log-structured NVMM eliminates the dichotomy between data home and data logs in current logging PTMs. This solves the vital NVMM fragmentation issue, and lowers NVMM write wear and persistence overhead. To that end, we create four key optimizations to tackle the performance challenge in tree-based address mapping. Our experiments show that the log-structured NVMM can outperform Mnemosyne and Mnmsyn-Undo by 44.7% and 80.8% on average in terms of transaction throughput. Our work reveals how a software tree structure can be optimized to a level that can efficiently serve address mapping for NVMM load/store instructions.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Yu Hua, for their valuable feedback. This work was partially supported by the National Natural Science Foundation of China (Grant No. 61502266, 61433008, 61232003), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and the China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098).

References

[1] Intel C++ STM compiler prototype edition. https://software.intel.com/en-us/forums/intel-c-stm-compiler-prototype-edition, 2012.

[2] AKINAGA, H., AND SHIMA, H. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE 98*, 12 (2010).

[3] APALKOV, D., KHVALKOVSKIY, A., WATTS, S., NIKITIN, V., TANG, X., LOTTIS, D., MOON, K., LUO, X., CHEN, E., ONG, A., DRISKILL-SMITH, A., AND KROUNBI, M. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst. 9*, 2 (May 2013), 13:1–13:35.

[4] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, pp. 707–722.

[5] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12, pp. 53–64.

[6] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), ASPLOS IX, pp. 117–128.

[7] BHANDARI, K., CHAKRABARTI, D. R., AND BOEHM, H.-J. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2016), OOPSLA '16, pp. 677–694.

[8] BRAGINSKY, A., AND PETRANK, E. A lock-free B+tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2012), SPAA '12, pp. 58–67.

[9] CHATZISTERGIOU, A., CINTRA, M., AND VIGLAS, S. D. REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures. *Proc. VLDB Endow. 8*, 5 (Jan. 2015), 497–508.

[10] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, pp. 105–118.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 143–154.

[12] DU, Y., ZHOU, M., CHILDERS, B. R., MOSSÉ, D., AND MELHEM, R. Bit mapping for balanced PCM cell programming. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA '13, pp. 428–439.

[13] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 15:1–15:15.

[14] ELLEN, F., FATOUROU, P., HELGA, J., AND RUPPERT, E. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (2014), PODC '14, pp. 332–340.

[15] EVANS, J. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference* (2006).

[16] FELBER, P., FETZER, C., AND RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP '08, pp. 237–246.

[17] FREE SOFTWARE FOUNDATION, INC. *The GNU C Library*, 2.24 ed., Aug. 2016. https://www.gnu.org/software/libc/manual/.

[18] GILES, E. R., DOSHI, K., AND VARMAN, P. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–14.

[19] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (2011), USENIX ATC '11, pp. 271–284.

[20] HERTZ, M., AND BERGER, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2005), OOPSLA '05, pp. 313–326.

[21] HIRABAYASHI, M. Tokyo cabinet: a modern implementation of DBM. `http://1978th.net/tokyocabinet/`, 2010.

[22] INTEL. The NVM Library. `http://pmem.io/`, 2016.

[23] INTEL NEWSROOM. Introducing Intel Optane technology – bringing 3D XPoint memory to storage and memory products. `https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/`, July 2015.

[24] KOLLI, A., PELLEY, S., SAIDI, A., CHEN, P. M., AND WENISCH, T. F. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ASPLOS '16, pp. 399–411.

[25] KLTRSAY, E., KANDEMIR, M., SIVASUBRAMANIAM, A., AND MUTLU, O. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceeding of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software* (Apr. 2013), ISPASS '13, pp. 256–267.

[26] LEA, D. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`.

[27] LEE, B., ZHOU, P., YANG, J., ZHANG, Y., ZHAO, B., IPEK, E., MUTLU, O., AND BURGER, D. Phase-change technology and the future of main memory. *IEEE Micro 30* (Jan. 2010), 131–141.

[28] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09, pp. 2–13.

[29] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies* (2015), FAST '15, pp. 273–286.

[30] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th Conference on File and Storage Technologies* (2009), FAST '09, pp. 111–123.

[31] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11, pp. 1–13.

[32] LINUX KERNEL ORGANIZATION, INC. Direct access for files. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`, 2016.

[33] LIU, M., ZHANG, M., CHEN, K., QIAN, X., WU, Y., AND REN, J. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ASPLOS '17, pp. 329–343.

[34] LIU, R.-S., SHEN, D.-Y., YANG, C.-L., YU, S.-C., AND WANG, C.-Y. M. NVM Duet: Unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, pp. 455–470.

[35] LU, Y., SHU, J., AND SUN, L. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–13.

[36] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., RANGANATHAN, P., AND BINKERT, N. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (2013), TRIOS '13, pp. 1:1–1:17.

[37] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII, pp. 401–410.

[38] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), NSDI '13, pp. 385–398.

[39] ORACLE. NVM Direct. `https://github.com/oracle/nvm-direct`, 2016.

[40] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16, pp. 12:1–12:16.

[41] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, pp. 265–276.

[42] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM 33*, 6 (June 1990), 668–676.

[43] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev. 52*, 4 (July 2008), 465–479.

[44] REN, J., ZHAO, J., KHAN, S., CHOI, J., WU, Y., AND MUTLU, O. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, pp. 672–685. `http://persper.com/thynvm/`.

[45] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[46] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST '14, pp. 1–16.

[47] SCHWALB, D., BERNING, T., FAUST, M., DRESELER, M., AND PLATTNER, H. nvm_malloc: memory allocation for NVRAM. In *In Sixth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (in conjunction with VLDB)* (2015).

[48] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 217–228.

[49] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies* (2011), FAST '11, pp. 61–75.

[50] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI, pp. 91–104.

[51] WANG, C., CHEN, W.-Y., WU, Y., SAHA, B., AND ADL-TABATABAI, A.-R. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization* (2007), CGO '07, pp. 34–48.

[52] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (2016), FAST '16, pp. 323–338.

[53] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST '15, pp. 167–181.

[54] YU, S., XIAO, N., DENG, M., XING, Y., LIU, F., CAI, Z., AND CHEN, W. Walloc: An efficient wear-aware allocator for non-volatile main memory. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)* (2015), IPCCC '15, pp. 1–8.

[55] ZHANG, Y., AND SWANSON, S. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies* (May 2015), MSST '15, pp. 1–10.

[56] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), MICRO-46, pp. 421–432.

[57] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09, pp. 14–23.