# Durable Transactional Memory Can Scale with TimeStone

R. Madhava Krishnan Jaeho Kim[†] Ajit Mathew Xinwei Fu Anthony Demeri Changwoo Min

Sudarsun Kannan[‡] [*]
*Virginia Tech* [†]*Huawei Dresden Research Center* [‡]*Rutgers University*

## Abstract

Non-volatile main memory (NVMM) technologies promise byte addressability and near-DRAM access that allows developers to build persistent applications with common load and store instructions. However, it is difficult to realize these promises because NVMM software should also provide crash consistency while providing high performance, and scalability. Durable transactional memory (DTM) systems address these challenges. However, none of them scale beyond 16 cores. The poor scalability either stems from the underlying STM layer or from employing limited write parallelism (single writer or dual version). In addition, other fundamental issues with guaranteeing crash consistency are high write amplification and memory footprint in existing approaches.

To address these challenges, we propose TimeStone: a highly scalable DTM system with low write amplification and minimal memory footprint. TimeStone uses a novel multi-layered hybrid logging technique, called *TOC logging*, to guarantee crash consistency. Also, TimeStone further relies on Multi-Version Concurrency Control (MVCC) mechanism to achieve high scalability and to support different isolation levels on the same data set. Our evaluation of TimeStone against the state-of-the-art DTM systems shows that it significantly outperforms other systems for a wide range of workloads with varying data-set size and contention levels, up to 112 hardware threads. In addition, with our TOC logging, TimeStone achieves a write amplification of less than 1, while existing DTM systems suffer from 2×-6× overhead.

**Keywords** transactional memory; multi-version; logging; scalability; write amplification;

[*]Jaeho Kim had contributed to this work while he was at Virginia Tech.

**ACM Reference Format:**
R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *2020 Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland* ACM, NY, NY. 14 pages. DOI: https://doi.org/10.1145/3373376.3378483

## 1 Introduction

New emerging non-volatile main memory (NVMM) technologies, such as Intel Optane [2, 63], provide persistence along with traditional main memory characteristics [84, 94], such as byte-addressability and low access latency. In addition, the NVMM offers data durability and larger in-memory capacity at a significantly lower `$/GB` compared to traditional DRAMs [14, 59, 75, 82, 92]. Although NVMMs incur higher read-write latency compared to traditional DRAMs [17, 42, 54], they enable software to have a larger capacity and almost attain free durability of data.

While NVMM technology is promising, it poses system developers with several new challenges such as guaranteeing crash consistency with a minimal write amplification, scalability, and high performance at high core counts. Even for byte-addressable NVMMs, guaranteeing crash consistency requires high latency logging operations in the critical path, complicated by the modern out-of-order processors that can reorder cacheline evictions. As a consequence, achieving crash consistency without impacting the manycore scalability and performance has become an onerous task [6, 11, 15, 21, 25, 27, 51, 90].

Nevertheless, manycore scalability is becoming an inevitable design principle when designing NVMM software as NVMMs are expected soon to be a part of data center manycore servers [8]. For example, the first public Cloud service of DCPMM used by SAP HANA, an in-memory database system, which requires manycore parallelism [8]. So a competent NVMM library should provide better performance and scalability, have a minimal write amplification, be memory efficient, and have broad-ranging applicability.

Unfortunately, none of the prior work exhibit all the above capabilities. For instance, prior concurrent durable data structure (CDDS) libraries [4, 13, 30, 41, 55, 65, 66, 70, 71, 86, 91, 95, 96] leverage application's data structure knowledge to achieve better scalability but do not guarantee atomicity of

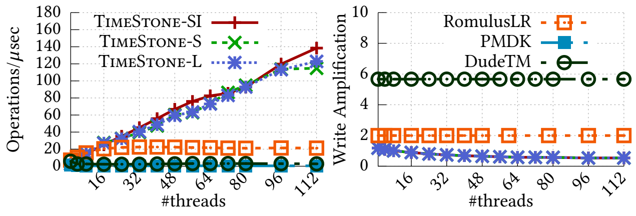


**Figure 1.** Performance comparison of DTM systems for concurrent hash tables with 2% update. Except TimeStone, prior systems suffer from poor scalability and high write amplification.

multiple operations, such as atomically adding two nodes to a list (durable composability). Moreover, current CDDS libraries do not guarantee consistency for data (full-data consistency), rather delegate it to the application developers. Failing to guarantee durable composability and full-data consistency delimits the usage of such libraries.

Contrary to CDDS libraries, existing DTM approaches [18, 24, 32, 57, 62, 85] support durable composability and provides full-data consistency. However, our analysis shows that none of the DTM systems scales beyond 16 cores (see Figure 1). For example, DudeTM [57] and Mnemosyene [85] scale poorly because of the underlying STM, which is known for its poor scalability [10]. They extend STM with an extra durability layer, which incurs a high write amplification (~4-7×), as shown in Figure 1 and Table 1 in the course of guaranteeing crash consistency. On the other hand, Romulus [18] and KaminoTX [62] minimize write amplification by maintaining a full backup of the NVMM, which derails the cost effectiveness of NVMM. Moreover, existing DTM systems support limited write parallelism (refer to Table 1) impacting their scalability, or leaving it entirely to the application developers to use locks [32]. More recently, Pisces [24] attempts to provide scalability by providing snapshot isolation. Unfortunately, only providing snapshot isolation delimits the use for applications requiring stronger isolation model. Importantly, the dual version concurrency control and the synchronous write during log reclamation in Pisces are bound to affect scalability [45], also increasing write amplification like other DTM approaches. [1]

To address all these problems, we propose a new DTM system, named TimeStone, which achieves *1) scalability across multiple cores*, guarantees *2) crash consistency with a significantly lower write amplification (< 1)* and also maintains *3) minimal additional memory footprint.* At its core, TimeStone adopts MVCC to achieve high concurrency and scalability but introduces several new principles to MVCC for scalable persistency in NVMM.

We believe that MVCC is a better design choice for DTM frameworks because of its inherent capabilities to support full-data consistency and the ability to run concurrent transactions with different isolation guarantees. To tackle the write amplification problem, we propose a novel multi-layered hybrid DRAM-NVMM logging scheme called the *TOC logging* with the ability to absorb the write-traffic to NVMM and significantly reduce write amplification. Further, to overcome the garbage collection and log reclamation overheads of MVCC [89], which impacts write throughput, we equip the TOC logging with a *scalable and concurrent log reclamation scheme.* This paper makes the following contributions:

- We introduce TimeStone, which is the *first highly scalable MVCC-based DTM system designed for NVMM.*
- We propose a *novel multi-layered hybrid DRAM-NVMM logging scheme, named TOC logging* to significantly reduce the write traffic and write amplification in the NVMM.
- We propose a *scalable and concurrent log reclamation scheme* to avoid log reclamation becoming a bottleneck in our MVCC-based design.
- We design TimeStone to concurrently support three different isolation levels (*i.e.*, linearizability, serializability, and snapshot isolation) on the same data set. To the best of knowledge, TimeStone is the first DTM framework to support *mixed isolation levels.*
- We provide a *familiar programming model* hiding the complexities of MVCC, concurrency, and durability from the user with C++ API.
- We evaluate TimeStone with key data structures and real-world workloads and our results show that TimeStone outperforms state-of-the-art DTM systems with significantly higher performance and lower write amplification.

## 2 Overview of TimeStone

We first elucidate our design goals and how we incorporate them in TimeStone, and then describe the key features of TimeStone with an illustrative example (see Figure 2).

### 2.1 Design Goals

**Write-Aware System Design.** Given the higher write latency, limited endurance, and high energy consumption of NVMM writes [17, 29, 36, 43, 52, 74, 75, 81], it is essential that NVMM applications should be write-aware. Unlike previous DTM systems [18, 32, 57, 62, 85] that suffer from high write amplification, we aim to significantly reduce amplification by making TimeStone write-aware. We adopt a hybrid multi-layered logging design (TOC logging) to absorb and coalesce a large chunk of redundant NVMM writes.

**Full-Data Consistency Guarantee.** To support crash consistency, it is critical to guarantee consistency for applications' data stores (data) as well as applications' internal data structure (metadata), which we term as *full-data consistency.* Failing to guarantee full-data consistency affects recovery and can lead to data corruption in the NVMM. For example, recent log-free data structure [19] designs guarantee only the consistency of pointers in a durable data structure (called *link consistency*) and delegates the data consistency to the application developer. Ignoring data consistency adds

[1]As of this paper publication, the source code of Pisces is not available.

| DTM Systems | Isolation Level | Parallelism RR | RW | WW | Durability How | When | Additional Memory DRAM | NVMM | NVMM Write Amplification |
|---|---|---|---|---|---|---|---|---|---|
| PMDK [32]†+ | LN | ● | ▲ | × | UNDO | $\Rightarrow]$ | - | `log` | $\geq 2$ |
| Mnemosyne [85]‡ | LN | ● | ▲ | ▲ | REDO | $\Rightarrow]$ | - | `log` | 4-7 |
| KaminoTX [62]+ | LN | ● | ▲ | ▲ | Backup | $\Rightarrow]$ | - | `NVMM` | $\geq 2$ |
| DudeTM [57]* | LN | ● | ▲ | ▲ | REDO | $\not\Rightarrow$ | `log + NVMM` | `log` | 4-6 |
| Romulus [18]* | LN | ● | × | × | Backup | $\Rightarrow]$ | - | `NVMM` | 2 |
| Pisces [24]+ | SI | ● | ● | ▲ | REDO | $\Rightarrow]$ | - | `log` | $\geq 2$ |
| TimeStone | SI/SR/LN | ● | ●▲ | ▲ | TOC | $\Rightarrow]$ | `TLog` | `OLog + CLog` | $\leq 1$ |

NOTE: SI: snapshot isolation SR: serializability LN: linearizability
×: not supported ▲: partially supported ●: fully supported $\Rightarrow]$: immediate durability $\not\Rightarrow$: eventual durability

**Table 1.** High-level comparison of DTM systems for NVMM. TimeStone is a DTM based on MVCC (Multi-Version Concurrency Control), which makes supporting multiple isolation levels (*i.e.*, mixed isolation levels) possible. Our novel TOC logging absorbs NVMM writes through three layers of logging so TimeStone can provide extremely low write amplification (below 1) on NVMM while providing immediate durability resulting in high performance and scalability. † While PMDK does not provide isolation, we assume that a PMDK transaction uses a readers-writer lock. We get write amplification by measuring by ourselves for $*$ or by analyzing the design of + or by referring the measured value for ‡ in Romulus [18]. We define write amplification as the ratio of the actual NVMM writes by application requests while Romulus counts only the *additional* data written in NVMM.

burden to the developer as it demands proper knowledge on correctly flushing the stores to ensure a proper recovery without any data corruption. We aim to provide full-data consistency without compromising the system performance and scalability.

**Immediate Durability.** For DTM systems, it is important to make updates *immediately durable* upon transaction commit. Prior DTM systems (*e.g.*, DudeTM [57]) defer durability to reduce commit latency; however adopting relaxed durability (*i.e.*, eventual durability) has the problem of losing some updates while recovering from a failure. In TimeStone, we make all the successfully committed updates immediately durable which enables our system to guarantee a deterministic and loss-less recovery and all of this without compromising the performance of the live transactions unlike some of the prior techniques [12, 23, 57, 58].

**Mixed Isolation Levels.** The level of required isolation guarantee depends on the application semantics and there is no single isolation level that can suit all application types. For example, even though snapshot isolation can provide good performance with more parallelism, it cannot be used for developing data structures without addressing write skew anomaly [9, 24, 45, 60]. On the other hand, stronger isolation levels such as linearizability might be an overkill for OLAP-class applications, which can tolerate weaker isolation reading slightly stale data [79]. We believe that supporting multiple isolation levels for the same data set is essential considering the rapid growth of data size, and the varied requirements of applications. Now, the beauty of MVCC is that it provides a way to represent multiple isolation levels. TimeStone supports any number of concurrent transactions running with three different isolation levels (linearizability, serializability and serializable snapshot isolation) to operate on the same data set.

**Decentralized Design for Scalability.** To achieve scalability on a manycore system, we should avoid any centralized scalability bottleneck in TimeStone. Prior DTM techniques [57, 62] use a centralized lock table notorious for scalability bottlenecks, and similarly, transaction techniques that use copy-on-write (CoW) for performing updates [22, 62] have centralized address mapping table that hinders scalability. We designed TimeStone not to have such a central entity by allocating resources at the thread level (*e.g.*, per-thread logging) and using hardware timestamps for coordination among threads.

## 2.2 Design Overview

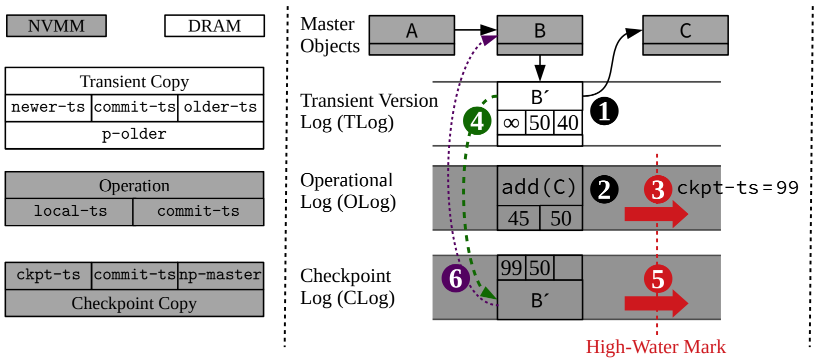
We explain the key design features of TimeStone and how we realized our design goals with an example in Figure 2.

### 2.2.1 Multi-Versioning

We adopt MVCC in TimeStone to exploit its inherent benefits for key features of TimeStone. Since MVCC makes out-of-place updates by composing a new version, which is a full replica of the respective original object, making the version durable guarantees full-data consistency. Importantly, because each version is discrete, it can concurrently support different isolation levels. Given these benefits, naive adoption of MVCC will incur a lot of write traffic, and affect the write endurance of NVMM with frequent writes defeating our design goals of minimizing write amplification and achieving high scalability. We solve these problems by using *TOC logging* and a scalable log reclamation scheme (see Figure 2 and §3).

### 2.2.2 TOC Logging

As illustrated in Figure 2, in TOC logging, we use a volatile log on the DRAM, named transient version Log (`TLog`), and two non-volatile logs, namely operational log (`OLog`) and checkpoint log (`CLog`). Essentially, each one of the logging



```
ts::ts_txn::run(
  ts::serializable,
  [&](){ add(C);});
// local-ts =45
// commit-ts=50
```

**Figure 2.** Illustrative example of adding a node in a TimeStone linked list. A thread adds a node `C` to a linked list in a TimeStone transaction (`ts_txn::run()`) with a serializable isolation level (`ts::serializable`). Consider a transaction that starts at timestamp 45 (*i.e.*, `local-ts` = 45) and commits at 50 (*i.e.*, `commit-ts`= 50). TimeStone first creates a copy of node `B` in `TLog` (`B'`) and updates its next pointer to node `C` ❶. When the transaction commits, TimeStone persists the executed operation (`add(C)`) to `OLog` making the transaction immediately durable ❷. Steps ❸ and ❺ denotes the log capacity crossing the high-water mark and this triggers the checkpointing for `TLog` reclamation ❹ and writeback for `CLog` reclamation ❻. During checkpointing, TimeStone checkpoints the latest transient copy (node `B'`) to the `CLog` so the `TLog` can be reclaimed ❹. In the `CLog` reclamation, TimeStone writes back the latest checkpoint copy (node `B'`) to the master object and the checkpoint log can be reclaimed safely ❻. The reclamation process is detailed in §3.9

layers is key to realizing our design goals. The volatile `TLog` reduces write amplification by absorbing redundant writes to NVMM and it is also key to achieving full-data consistency. The `OLog` is important to guarantee immediate durability and the `CLog` guarantees a deterministic recovery. For scalability without a central bottleneck, all three logs are per-thread logs and updates are synchronized using the hardware timestamp. The *TOC logging* is a combination of the traditional redo logging and the operational logging, but our novelty lies in the multi-layered hybrid placement of logs (in DRAM and NVMM) and their usage for DTM in tandem. Next, we explain how the three logs are used in a typical TimeStone transaction.

**Transient Version Log.** As shown in the step ❶ in Figure 2, before modifying an object, the thread first makes the full copy of the respective master object (transient copy) in the `TLog` by locking the object (`ts_lock` in Figure 3). It then executes transactions on the respective copy, and upon successful commit, the transient copy object is added to the version chain. During the NVMM writeback, the thread writes only the latest transient copy (❹ in Figure 2). Consequently, a large chunk of redundant NVMM writes is absorbed in the `TLog`, which is the key to achieving a lower NVMM write amplification.

**Operational Log.** The `OLog` is pivotal in guaranteeing immediate durability upon transaction commit. `OLog` stores only the transaction semantics (❷ in Figure 2), which is essentially a function pointer and its argument of a transaction, and re-executes them during recovery. Unlike the traditional undo or redo logging, `OLog` does not log the entire transaction data, and neither incurs read indirection nor requires multiple store flushes. As a result, `OLog` reduces write amplification, improves scalability, and guarantees durability using a single persistent barrier (`clwb` and `sfence`) per transaction.

**Checkpoint Log.** The `CLog` is essential to maintaining the master object in a consistent state and to tolerate any potential failure during writing back the checkpoint copy (❻ in Figure 2). If master objects are inconsistent, re-executing `OLog` does not guarantee recovery. During recovery, the master objects are first reset to the most recent checkpointed status available in the `CLog`.

### 2.2.3 Mixed Isolation Levels

With the goal of making TimeStone a generic framework suitable for a wide range of applications, TimeStone supports multiple isolation levels–linearizability, serializability, and snapshot isolation–on the same instance of an application such that transactions with different isolation levels can run concurrently. Applications that require high performance but can tolerate write-skew or slightly stale reads should use TimeStone's snapshot isolation, while applications that needs stricter isolation levels can fall back to linearizability or serializability. For linearizability and serializability, TimeStone additionally employs a read set validation at commit time where we check if any of the objects dereferenced has been updated during the course of current transaction; If so, we simply abort and retry again. Note that the serializability and linearizability differs in the object derefernce semantics and still follow the same read set validation procedure in our design.

### 2.2.4 Scalable Garbage Collection

TimeStone maintains fixed size logs and hence the memory usage of logs are limited. If one or more logs becomes full, this could block all writes until logs are reclaimed. Hence garbage

```
1  struct node : public ts::ts_object { // Inherit ts_object
2   int64_t val; // data on NVMM
3   // p_next: persistent pointer to a master object
4   ts::ts_master_ptr<node> p_next;
5  };
6  class list : public ts::ts_object { // Inherit ts_object
7   // p_head: persistent pointer to a master object
8   ts::ts_master_ptr<node> p_head;
9  public:
10  bool add(int64_t val) {
11   // p_prev, p_next: version-resolved pointer to a copy object
12   ts::ts_copy_ptr<node> p_prev = p_head;
13   ts::ts_copy_ptr<node> p_next = p_prev->p_next;
14   while (p_next) {
15    if (p_next->val >= val) {
16     // Lock the object (p_prev) before update
17     // creating a transient copy of p_prev in TLog
18     ts::ts_lock(p_prev);
19     // Allocate a master object on NVMM
20     ts::ts_copy_ptr<node> p_new_node = new node;
21     p_new_node->val = p_next->val;
22     // In assigning to a persistent pointer, a persistent
23     // master object pointer of a copy will be assigned.
24     p_new_node->p_next = p_next;
25     p_prev->p_next = p_new_node;
26     return true; } // end of if
27    // In assigning to a copy pointer, a version-resolved
28    // copy pointer will be assigned after version resolution.
29    p_prev = p_next;
30    p_next = p_prev->p_next; } // end of while
31   return false; } // end of add()
32 };
33 void thread_main(int64_t v) {
34  // Run a transaction with given isolation level and function.
35  // Upon abort, ts_txn::run internally re-tries the transaction.
36  ts::ts_txn::run(ts::serializable,
37                  [&]() { p_list->add(v); p_list->add(v+1); });
38 }
39 int main(int argc, char *argv[]) {
40  // Spawn a thread for concurrent transaction execution.
41  ts::ts_thread worker(thread_main, argc);
42  worker.join();
43  return 0;
44 }
```

**Figure 3.** A linked list adding two nodes in one transaction.

collection can directly impact write throughput. Also, a synchronous and non-scalable garbage collection scheme can quickly become a bottleneck hampering the performance of the system [89].

For the garbage collection to be scalable, TimeStone must identify safe objects to reclaim without any centralized lookup or coordination. Importantly, garbage collection must be NVMM-write aware so that it does not increase direct writes to NVMM. Hence, TimeStone employs a *timestamp-based reclamation* scheme where decisions like what/when to reclaim are solely made based on the object-local timestamp without accessing shared structures. To harness concurrency in the garbage collection, TimeStone delegates responsibility of reclamation to each thread that holds the log itself (*i.e.*, *concurrent reclamation*). To further reduce NVMM writes, we introduce *best-effort reclamation*, which reclaims objects that do not incur NVMM writes. We explain the details in §3.9.

#### 2.2.5 Programming Model

TimeStone follows the programming model of object-level, lock-based software transactional memory providing full ACID guarantee on NVMM. TimeStone provides C++ API so programming in TimeStone is just writing a typical C++11 code using TimeStone base classes and APIs as shown in Figure 3.

User-defined persistent structures (*e.g.*, `struct node` in Figure 3) that inherit `ts_object` (line 1, 6) will be allocated on the NVMM (line 20). To hide the complexity of NVMM memory management, concurrency control, and version resolution in MVCC, TimeStone provides two smart pointers; `ts_master_ptr` points to a master node on NVMM and `ts_copy_ptr` points to a version-resolved copy, which is part of the version chain on `TLog`. Essentially, `ts_copy_ptr` should be used in the function that accesses the `ts_object` such as `list::add()`. Type conversion between two smart pointer types involves version resolution (line 13, 24) in Figure 4. To modify an object, first it should be locked using `ts_lock` (line 18). Once a lock is acquired, `ts_copy_ptr` will be updated to pointing a new transient copy of the object (❶ in Figure 2). A code executed within `ts_txn::run` is a full ACID transaction and the required isolation level can be specified per transaction (line 37). Upon abort, `ts_txn::run` internally re-tries the transaction. `ts_thread` is a shallow wrapper inheriting `std::thread` that registers and deregisters a thread to TimeStone (line 41).

## 3 Design of TimeStone

We first describe the basic metadata structures of TimeStone's transactional object followed by versioning, logging, committing, and recovery mechanisms.

### 3.1 Object Representation

In TimeStone, updates to NVMM are in object granularity. To make TimeStone NVMM write-aware, we maintain frequently modified metadata (control headers) and persistent intermediate copy objects on DRAM in addition to application data objects. We next discuss their details.

**Master Object on NVMM.** In TimeStone, every persistent data structure is represented by a non-volatile *master object* that acts as a handle for these persistent structures. To reduce overheads of frequent access of a master object on slow NVMM, TimeStone also maintains a volatile *control header* per master object on DRAM as shown in Figure 4. This volatile pointer (`p-control`) is validated by matching the `gen-id` cached in the *master object* and a global `gen-id`, which increments each time the non-volatile heap is loaded.

**Copy Object.** TimeStone maintains two different copy objects: a *transient copy* and a *checkpoint copy*. A transient copy is created on `TLog` during update operations while a checkpoint copy is created on `CLog` when the transient copy is checkpointed during `TLog` reclamation (❹ in Figure 2). As illustrated in Figure 4, a copy object caches essential timestamp information that is required to make object-local decisions during version resolution and log reclamation.

**Control Header on DRAM.** As shown in Figure 4, *control header* stores per-object run time metadata such as the

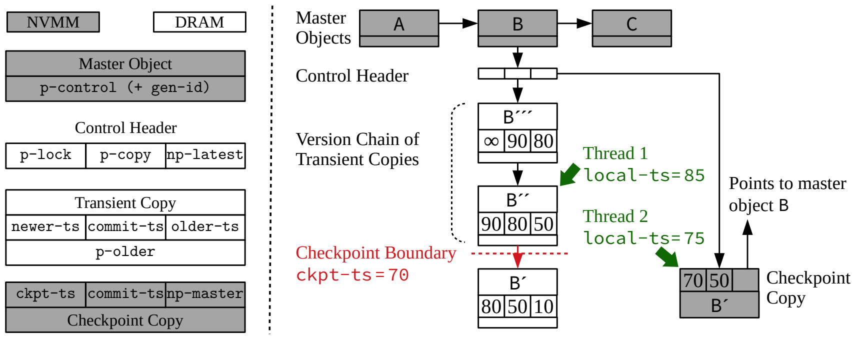


**Figure 4.** Object representation and version resolution in TimeStone. An object consists of one *master object* on NVMM and one or more *transient copies* in a transient version log (TLog) on DRAM and *checkpoint copies* in a checkpoint log (CLog) on NVMM. Each copy has a its commit timestamp (commit-ts) as well as timestamps of older and newer versions (older-ts, newer-ts) to make decisions without pointer chasing. A version chain is ordered from the latest to the oldest version starting at a *control header* on DRAM. Object dereferencing finds the closest past version (commit-ts) to when a thread starts a transaction (local-ts). For example, when local-ts of a thread is 85, the thread will read B" with commit-ts 80 (*e.g.*, Thread 1). The version chain traversal should stop at the last checkpoint boundary because transient versions older than the last checkpoint timestamp (ckpt-ts) would already have been reclaimed. In this case, a thread should fall back to the latest version on NVMM pointed to by np-latest on a control header (*e.g.*, Thread 2).

version chain head (p-copy) and lock-status (p-lock). The control header is created when the master object is first updated. The decentralization of metadata enables faster metadata lookup and update performance and also significantly reduces the frequency of NVMM access. The control header serves as the entry point to access copy objects and it also helps copy objects to reference their respective master.

### 3.2 Version Chain Representation

In TimeStone, the version chain is a singly linked list with the newest object at the head of the linked list. As illustrated in Figure 4, version chain traversal is delimited by the latest checkpoint timestamp (ckpt-ts) because transient copies older than the checkpoint boundary would have already been reclaimed. For any access request beyond the checkpoint boundary, the latest checkpoint copy is dereferenced via the control header. The latest transient copy is always stationed at the head, so new threads starting a transaction may traverse until the checkpoint boundary in the worst case. Therefore, the length of a version chain does not affect performance or chain traversal cost.

### 3.3 Object Version Dereferencing

Our design provides a generic versioning support that can satisfy snapshot isolation, serializability, and linearizability. A thread does a control header lookup to get to the version chain head and traverses the chain. The thread compares its local-ts against the commit-ts of the transient copies and dereferences the closest past version, which is the most recent transient copy with commit-ts lesser than the local-ts. If a thread reaches the checkpoint boundary (ckpt-ts), it falls back to the control header and then the latest checkpoint copy (np-latest) is dereferenced. If a control header does not exist yet, the thread dereferences the master object. Note that these dereference semantics are the same for both snapshot isolation and serializability. In case of linearizability, we dereference the latest transient copy in the version chain without any further traversal. This is because linearizability requires reading the latest object and in the situation where the latest transient copy is a future version for the current transaction, we abort and retry to preserve the object dereferencing correctness.

### 3.4 Updating an Object

Before updating an object, the writer-thread attempts to lock the control header of the associated master object; if the lock could be acquired, a transient copy of the respective master object is created on TLog (❶ in Figure 2) and p-lock in the control header (pointing to NULL) is atomically modified to point this transient copy. A non-NULL p-lock means that there is an ongoing update and hence the thread aborts (see §3.6). Note these lock failures (*i.e.*, p-lock is not NULL) are hidden from the user and in such cases, the writer-thread aborts and retries. The absence of a control header indicates no updates to the master object. Hence the current writer-thread is responsible for creating and locking the control header.

### 3.5 Committing a Transaction

A writer-thread maintains a private write set on TLog consisting of all updates made in a transaction. We first persist our OLog entries to make them durable and then we make all updates in the write set atomically visible; we add each of the copy objects (p-lock) to respective version chain head (p-copy) and then atomically update the commit-ts of the write set to the current hardware clock. Finally, we update the commit-ts field in the new copy objects with the write set commit-ts. For stricter isolation levels such as linearizability and serializability, an additional read set validation is carried out at the beginning of a commit procedure. The read set validation checks if the view of an object has changed since the arrival of this transaction. If so, then the current transaction is aborted and all updates are discarded.

### 3.6 Aborting a Transaction

A writer-thread aborts a transaction upon a `ts_lock` failure or in the event of stale-reads upon read set validation or reading a future version in linearizability. When aborting a transaction, the writer-thread unlocks all control headers by resetting `p-lock` to `NULL` and rolls back the log space. We also free the new master objects that were allocated inside the aborting transaction.

### 3.7 Timestamp Allocation

For timestamp allocations, we leverage the hardware clock (`rdtscp` in x86 architectures) to prevent the timestamp allocation from becoming a scalability bottleneck [44, 47, 56, 83, 93]. As hardware clocks can have a constant skew between processor cores which can lead to incorrect ordering, we use the ORDO primitive [44] and avoid this inconsistency. ORDO assumes there is a constant clock drift among cores and it compensates for the drift using the pre-measured ORDO boundary. ORDO is a software-based technique and only assumes invariant timestamp counter, which is already supported in x86 and many other architectures [44].

### 3.8 TOC Logging

All the logs in TimeStone are modeled as per-thread circular logs with new entries updated at the tail. `TLog` is created in the volatile memory while `CLog` and `OLog` are placed in the non-volatile heap. `CLog` and `OLog` are accessible from the root object of the non-volatile heap. Before terminating TimeStone, but after all threads safely exit the TimeStone transaction, we free all logs on the non-volatile heap and make the root object to point to `NULL`. Thus, a `NULL` root object upon starting TimeStone signifies safe termination in the previous run. We leverage this design invariant to trigger the recovery.

### 3.9 Log Reclamation

Log reclamation or garbage collection (GC) is critical as it directly impacts the write performance of the system.

**Concurrent Reclamation.** To prevent garbage collection being bottlenecked either by a single thread or due to synchronous waiting [60], we employ a self-reclamation scheme [45] in which, the thread that holds the log is responsible for the reclamation of its state. This design is scalable, asynchronous, thread-local, cache- and prefetcher-friendly [36]. Each thread at the transaction boundary checks if it needs to perform a log reclamation. If so, it triggers the `gp-detector` thread to broadcast the last detected grace period timestamp, which we will define shortly. To avoid race conditions, we add a reclamation barrier to avoid any new trigger before the currently running reclamation is finished. To ensure liveness of log reclamation, the `gp-detector` thread reclaims the log of a thread which did not initiate reclamation.

**What to Reclaim?** We employ a RCU-style grace period detection algorithm to identify the safe reclaimable objects [26, 45, 60, 61]. Grace period is an interval in which all threads in TimeStone are outside or have exited the transaction boundary. Grace period timestamp is the time at which the a grace period detection begins. A background `gp-detector` thread continuously detects the grace period and broadcasts the grace period timestamp to all thread when log reclamation is requested.

An object is obsolete if it has a newer version, so it is no longer visible to new threads entering transactions and does not have any new references in a transaction. When all threads reading an obsolete object exit the transaction, the obsolete object becomes invisible and is safe to reclaim. Thus, as per grace period semantics, *a copy object can be safely reclaimed if one grace period has elapsed since it became obsolete.* The grace period semantics guarantee that there cannot be any thread in a transaction with `local-ts` less than two grace periods. So TimeStone always waits for at least two grace periods to elapse before reclaiming any copy object. Note that we cannot reclaim a copy object if it is the latest version of the associated master object as it is still visible to all threads. The copy object has to be checkpointed, followed by the completion of one more grace period before the copy object can be safely reclaimed.

**When to Reclaim?** Ideally, deferring reclamation until the log comes to capacity improves the chance for coalescing updates which will reduce the frequency of NVMM writebacks. However, we can not afford the log resources to become full as it can block writers hampering system performance. Keeping this in mind, we maintain a preset *high-water mark* and *low-water mark* for all three logs. When a log utilization exceeds high-water mark, the log is fully reclaimed incurring NVMM writes (checkpoint reclamation in `TLog` and write-back reclamation in `CLog`). When the utilization is between low and high-water mark, the log is reclaimed in a *best-effort mode*. In the best-effort reclamation, a thread reclaims its log until the first writeback to NVMM is required. The first writeback is the point at which the thread encounters the latest transient copy object of the respective master object. Stopping at the first writeback allows coalescing updates as future updates on the same object is expected. The deferred object will be reclaimed or written back in the next reclamation cycle. Below we explain how each of log reclaims in detail.

**Transient Version Log Reclamation.** In checkpoint reclamation passing the high-water mark, a thread first checks if this transient copy object is the latest version, then it checkpoints the copy to the `CLog` if one grace period has elapsed since this copy object is committed. After checkpointing, any future reference to this object is served from `CLog` and the checkpoint boundary (`ckpt-ts`) is set to the grace period timestamp when starting the reclamation. We then wait for one more grace period to pass and then safely reclaim the copy from TLog. After this, all versions with commit-ts less than ckpt-ts are deemed safe to be reclaimed. This ensures that the object does not have any references. Note that we wait for one grace period before checkpointing and one additional grace period after checkpointing making it at least two grace periods since the arrival of the copy object and that makes it safe to reclaim as per grace period semantics. Alternatively, if the entry is not the latest version then it is simply skipped so that the thread that has the latest version in its TLog will checkpoint that entry.

In the best-effort reclamation of TLog, the thread reclaims its TLog entries until it encounters the first writeback to NVMM. It is prudent and optimal to defer writeback until log utilization goes above the high-water mark. This deferring helps reduce the frequency of writeback to NVMM.

**Checkpoint Log Reclamation.** Writeback reclamation in CLog works similar to checkpoint reclamation in TLog except that it writebacks the checkpoint copy to its corresponding master object. A thread performs writeback only if it is the latest checkpoint copy, else it is skipped and the thread that has the latest checkpoint copy in its CLog will writeback during its reclamation. Again, similar to TLog, we wait for two grace periods to pass before safely reclaiming an object. This asynchronous waiting for at least two grace period is to ensure that there is no any existing reference to this checkpoint copy in TimeStone transactions.

The best-effort reclamation of CLog also follows a similar semantics of TLog where the thread stops reclaiming its CLog when it encounters the first writeback to the master. Again, this deferring the writeback is done with a goal to coalesce multiple checkpoint copies associated with the same master object and then writeback the latest copy when the thread reclaims its CLog in the writeback reclamation.

**Operational Log Reclamation.** An OLog entry is deemed to be reclaimable based on the last checkpoint boundary (ckpt-ts). So all the entries with commit-ts less than the ckpt-ts can be reclaimed anytime, independent of any grace period semantics. When OLog utilization goes above the high-water mark, it triggers checkpoint reclamation of TLog and as a result, ckpt-ts is updated. At this point, OLog can discard all the entries with the commit-ts lesser than the ckpt-ts.

## 3.10 Freeing an Object

To free a master object, we first lock the object to avoid any race condition while freeing it. If the object to be freed is in TLog then we cannot immediately free it as there might be one or more checkpoint copies in CLog that would still require a reference to the master object during its reclamation. Hence, we insert a tombstone of the master object to CLog. Tombstone-marked master objects will be freed when we find them upon CLog reclamation. Note that the lock will not be released until the master object is freed to prevent double-free and update-after-free of the master object.

## 3.11 Recovery

TimeStone's recovery procedure guarantees no-loss recovery. As mentioned in §3.8, on safe termination of TimeStone, we free all logs in the non-volatile heap. When there are non-volatile logs upon start, TimeStone triggers recovery procedure. The recovery procedure is a two-step process, which first replays CLog and then replays OLog.

**Checkpoint Log Replay.** The goal of CLog replay is to find the latest checkpoint copy associated with each master object by sequentially examining all CLogs in NVMM. The replay routine constructs a control header for each master object that has a corresponding copy in the checkpoint log. np-latest field in the control header is updated to the newest copy by comparing commit-ts of all the copies of the same master object. This sets up the master objects to the last consistent state before the failure and thus prepares it for OLog replay.

**Operational Log Replay.** The goal of OLog replay is to restore back to the latest committed state before the failure occurred. To restore the application back to the last consistent state before the failure, the transactions that happened after the last checkpoint timestamp (*i.e.*, ckpt-ts stored in NVMM) should be re-executed from OLog. *The transactions in OLog should be re-executed in the original local-ts order and should be committed in the original commit-ts order to avoid any inconsistent view.* local-ts ordering is essential to ensure a consistent version view for replay transactions as it had during the live execution while a proper commit-ts ordering will bring back the application to the same old status that existed before the failure. Note that the OLog replay does not require any global state as the CLog replay will establish the required state (as in the live execution before failure) for the OLog entry to be correctly executed.

**Recovery Time.** The recovery cost in TimeStone is roughly constant and it is directly proportional to the utilization of OLog and CLog (number of entries in them) and the worst case being both the CLog and OLog are full. Apparently, the OLog entries have to be executed only for the TLog entries that have not been checkpointed yet. Since our log size is limited and our TOC logging scheme keeps checkpointing the updates regularly and hence we do not have to re-execute all OLog entries that occurred before the failure.

## 4 Implementation

We implement TimeStone in C and C++. The core library written in C, which comprises of around 7000 lines of code and C++ API comprised of 800 lines of code. To abort a transaction, we use setjmp and longjmp instead of C++ exception because we found throwing an exception in C++ is not scalable [77]. We use modified jemalloc allocator as our non-volatile memory allocator, nv-jemalloc, similar to some of the previous works [19, 57]. We could not use the PMDK allocator [33] or Makalu allocator [7] because of their poor scalability.

## 5 Evaluation

In this section, we first show that TimeStone achieve better scalability and performance across different data structures under various workload configurations compared to state-of-the-art DTM systems (§5.1). We then show the effectiveness of TimeStone for real-world workloads (§5.2). Finally, we thoroughly analyze the effectiveness of TOC logging in reducing write amplification (§5.3).

**Evaluation Platform.** We use a system with Intel Optane DC Persistent Memory (DCPMM) for our evaluation. The machine consists of two sockets with Intel Xeon Platinum 8280M processors equipped with 28 cores (56 logical cores) per socket (112 logical cores in total), 1.5 TB of NVMM (12 × 128 GB), and 768 GB of DRAM (12 × 64 GB). We used `gcc` 9.1.1 with `-O3` flag to compile benchmarks and ran all experiments on Linux kernel 5.0.9.

**Configuration.** We preset the size of `TLog` and `OLog` to 1 MB and `CLog` to 4 MB. We also set the high-water mark to 75% for all logs. We set low-water mark to 50% and 62.5% to `TLog` and `CLog`, respectively. We also present the performance analysis for varying log sizes and analyze the sensitivity of TOC logging in §5.3. We ran TimeStone for different isolation levels. Note that TimeStone-SI denotes snapshot isolation whereas TimeStone-L and TimeStone-S represents linearizability and serializability versions, respectively. In order to evaluate the data structures under the snapshot isolation, we removed the write-skew by locking the adjacent nodes in addition to the nodes that are being updated as prior work [24, 45, 60] did. We compare our TimeStone with the state-of-art DTM systems: DudeTM, Romulus, and Intel's PMDK. Both DudeTM and Romulus provide their own memory allocator, and we ported them such that they allocate memory on the NVMM. For Romulus, we handpicked RomulusLR with the best performance. PMDK's transactional library `libpmemobj` does not support isolation, so we use a standard readers-writer lock to protect a transaction from concurrent accesses.

### 5.1 Concurrent Durable Data Structures (CDDS)

We evaluate three persistent data structures–linked list, hash table, and binary search tree–for three types of workloads similar to several prior works [3, 31, 81, 96]: 1) read-mostly (2% update), 2) read-intensive (20% update), and 3) write-intensive (80% update) operations.

We present the performance and scalability in Figure 5 and present abort ratios in Figure 6 for further analysis of each DTM systems.

**Linked List.** We use a singly linked list with 10,000 items for our evaluation. TimeStone exhibits relatively a better scalability across all the workloads than the other DTM systems but the performance of DudeTM and Romulus are upto 2× better than TimeStone only for the read-mostly workload and this happens only in the case of linked list.

For the linked list, DudeTM performs well at a lower core counts and starts to collapse beyond 16 cores. Since DudeTM supports decoupled durability–replicating *all* the persistent data and logs on the DRAM, the foreground thread just accesses the replica on DRAM and writes to its volatile log and the background thread persists the log entries later. That makes foreground writes much faster and for the same reason persisting by the background thread becomes a high latency operation. The background thread becomes a bottleneck since it cannot keep up with the rapidly filling of log entries as the number of foreground thread increases. This eventually blocks incoming writes and leads to a performance collapse.[2] This is more evident in read-intensive and write-intensive workloads as it collapses after 6 cores. Unlike DudeTM, TimeStone guarantees immediate durability and hence it has to persist its `OLog` entry upon commit. TimeStone shows upto 10× lesser abort ratio than DudeTM and even the stricter isolation version of TimeStone shows 2.5× lesser abort ratio. This is because the underlying fixed-size central lock table in DudeTM causes spurious aborts. The decentralized resource allocation helps in achieving better scalability and lower abort ratio in TimeStone.

As with Romulus, the single writer thread becomes a bottleneck in read-intensive and write-intensive workloads and for the same reason Romulus performance starts to saturate after 40 cores in the read-mostly workload. Because of the inherently larger critical section in the linked list, the single writer latency is masked in the read-mostly scenario. Note that Romulus never aborts so its abort ratio is always zero.

Note that although PMDK uses a readers-writer lock due its additional logging and NVMM allocator overhead in the critical path causes the performance collapse and poor scalability.

**Hash Table.** For the evaluation, we create a hash table with 1,000 buckets, where each bucket points to the head of a singly linked list. TimeStone outperforms all the other DTM systems by upto 30× and exhibits a near-linear scalability. The cause for the performance collapse in the other DTM systems is same as observed in the linked list. Pisces [24] implements a similar hash table with the same load factor but with 10× more buckets and TimeStone still outperforms Pisces by 2×-6×. Note that higher number of buckets increases the concurrency and thereby aborts are reduced.[3] We believe that having multiple versions and employing TOC logging for effectively handling crash consistency makes TimeStone performs better than Pisces.

**Binary Search Tree.** In BST, TimeStone performs upto 10× better than the other and exhibits a better scalability.

[2] The DudeTM code supports only single-threaded background persist.

[3] Because Pisces code is not publicly available, we compare the performance reported in their paper against our similar hash table configuration.

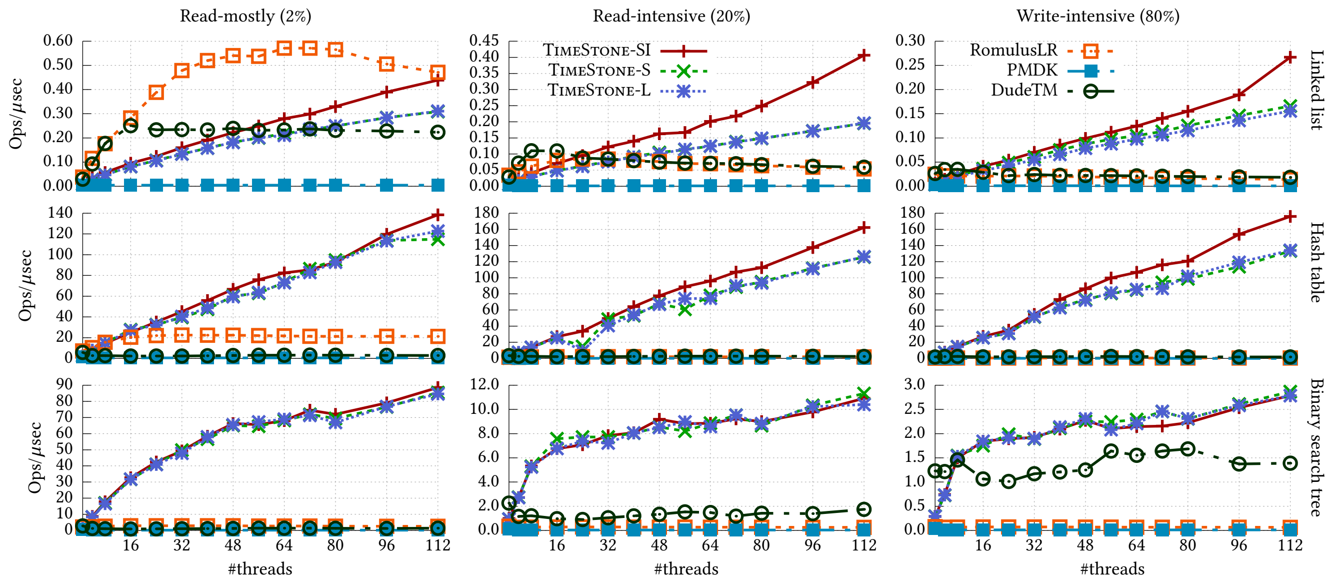


**Figure 5.** Performance and scalability of concurrent data structures: a 10,000 item linked list, hash table (1K buckets), and binary search tree with read-mostly, read-intensive, and write-intensive workloads.

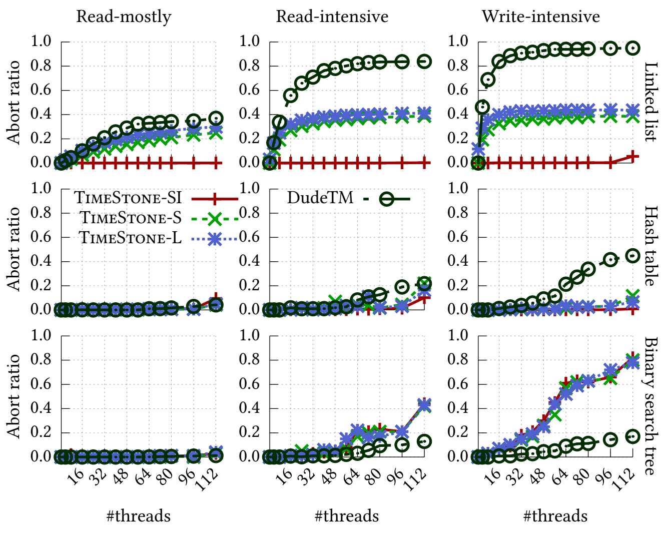


**Figure 6.** Abort ratio of concurrent data structures.

The slightly skewed scaling and the TimeStone-SI performing same as the stricter versions of TimeStone can be attributed to higher chances of lock failure in common ancestor nodes (*e.g.*, root node) causing a spike in the abort ratio. The cause for the performance collapse in the other DTM systems is same as observed in the linked list. **Scalability across Isolation Levels.** From our analysis, it is evident that TimeStone shows a better scalability for all the three isolation levels. TimeStone-S and TimeStone-L shows a superior scalability when compared to the linearizable DTM systems such as DudeTM and Romulus. Particularly, for hash table and binary search tree, TimeStone-S and TimeStone-L shows a similar throughput and abort ratio as that of TimeStone-SI because of better concurrency of those data structures. This is perceptible from the abort ratio also as all the three versions of TimeStone exhibits a similar abort ratio. In DudeTM, the lesser aborts of a hash table can also be attributed to the better concurrency levels in the data structure. Overall, the scalability across read workloads can be attributed to our MVCC-based design while our TOC logging equipped with efficient garbage collection makes TimeStone scalable even for the write-intensive workloads.

## 5.2 Real World Workload

To analyze the impact of TimeStone for real-world workloads, we use Kyoto Cabinet [1] and YCSB benchmark [16].

**KyotoCabinet.** KyotoCabinet is an in-memory database which is internally divided into a number of slots and each slot hosts a number of buckets that point to a binary search tree. Concurrent access of each slot is protected by a per-slot lock. We replaced the binary search tree to the TimeStone binary search tree to provide synchronization and crash consistency for database operations. We compare our implementation (KyotoCabinet-TimeStone) against the vanilla KyotoCabinet (KyotoCabinet-vanilla) that runs on the faster DRAM and KyotoCabinet-NVMM where the binary search trees are allocated on NVMM. Note that KyotoCabinet-NVMM does not provide crash consistency so there is no logging operations involved. As Figure 7 shows, TimeStone outperforms other systems and scales even with an additional overhead of providing crash consistency. It is important to note that KyotoCabinet in general is not scalable [20] and the performance starts to saturate after 16 cores. Using TimeStone, we make KyotoCabinet scale beyond 16 cores in addition to making KyotoCabinet crash consistent.

**YCSB.** We implemented a B+-tree for TimeStone and DudeTM to evaluate YCSB benchmarks. We set the B+-tree fan-out to

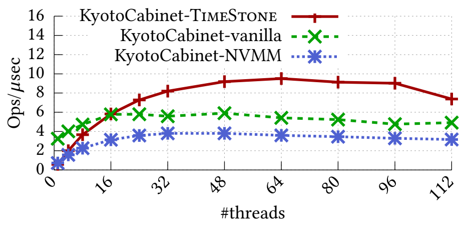


**Figure 7.** Performance comparison of TimeStone on KyotoCabinet for read-mostly workload.

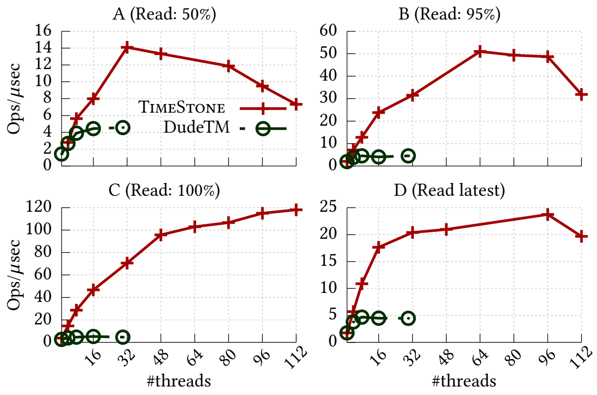


**Figure 8.** Performance of TimeStone and DudeTM with YCSB.

8 and ran 20 million operations using an index benchmarking tool, index-microbench [88]. TimeStone significantly outperforms DudeTM.[4] TimeStone scales reasonably well for read-dominant YCSB B, C, and D while TimeStone shows performance dip for YCSB A. That is because high update ratio in workload A (50%) causes the high latency split and merge operations in the B+-tree. Overall, TimeStone performs upto 6× more and scales better than DudeTM but there is 0.5× dip in the performance for write workloads caused due to the split and merge in B+-tree.

## 5.3 Analysis on Design Choices

In this section, we show how the TOC logging benefits the write amplification, how it provides stability to TimeStone even with a larger dataset size and smaller log size.

### 5.3.1 Write Amplification

We ran the persistent data structures for write-intensive workloads and compared it with Romulus and DudeTM in Figure 9. The write amplification is defined as ratio of the actual amount of data written to NVMM by the amount of user request data.

We infer that TimeStone consistently reduces direct NVMM writes compared to RomulusLR and DudeTM for the following reasons. First, RomulusLR must propagate every updates to the NVMM backup incurring 2× write amplification. Similarly, DudeTM also writes the user request data to redo log in NVMM first and writes it back to data location of the NVMM.

[4] We could not show full scalability results of DudeTM because the DudeTM crashes after 30 cores.

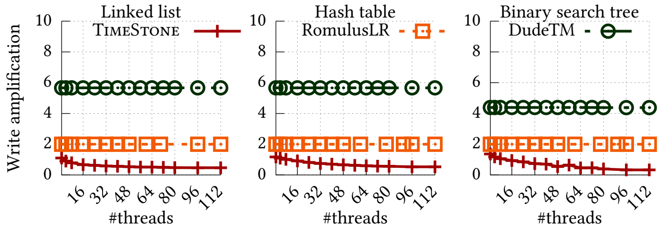


**Figure 9.** Comparison of write amplification incurred in different DTM systems for a write-intensive workload.

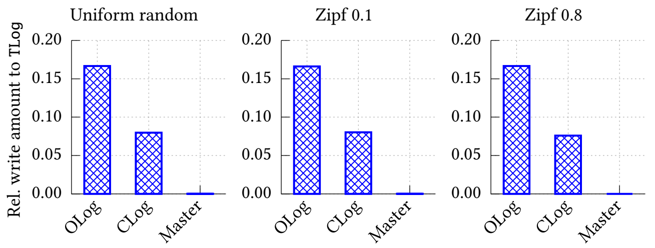


**Figure 10.** The total bytes written for each log in TimeStone for the varying skewness of read-intensive workload. Y-axis is relative to `TLog`.

Redo logging in DudeTM significantly increases the write amplification since it involves not only data but also other metadata (*e.g.*, address) per transaction. But in TimeStone only the `OLog` write goes to NVMM to guarantee immediate durability. Second, `TLog` absorbs a large chunk of redundant NVMM writes as Figure 10 shows. Only ∼ 6% of the total `TLog` writes is being checkpointed to `CLog` and less than 1% of it is written back to the master. The presence of `TLog` becomes more significant if the skewness of the access increases. Third, as depicted in Figure 9, the write amplification decreases as the thread count increases. This can be attributed towards higher write coalescing in `TLog`. So overall, the TOC logging reduces write amplification by absorbing redundant writes in `TLog` along with our garbage collection mechanism which prudently controls the NVMM writes. Note that write amplification for TimeStone and DudeTM reduces about ∼0.3× and 2×, respectively, in case of binary search tree. This is due to the better coalescing chance in BST as common ancestor nodes more frequently updated. Romulus write amplification is unchanged because it has to propagate the updates to the backup heap irrespective of the underlying data structure.

To glimpse the write amplification in PMDK, we modified the `pmemcheck` [34] and observed cacheline flushes incurred in a PMDK transaction with our hash table benchmark. In PMDK, each insert operation incurs 18 flushes consisting of 2 flushes for transaction initialization, 6 flushes for NVMM allocation, and 10 flushes for user data update and logging. Even a read-only transaction incurs 2 flushes for transaction initialization. *Overall we observed surprisingly high write amplification, 73.5×, for 1-million transactions with 2% writes.*

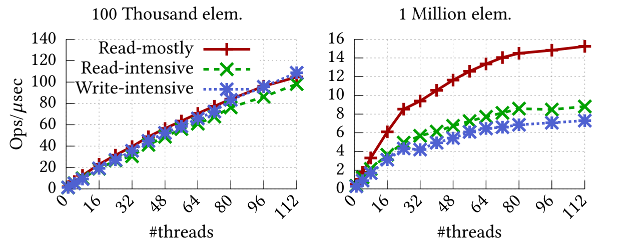


**Figure 11.** Performance of TimeStone for a larger hash table size.

### 5.3.2 Sensitivity Analysis

**Dataset Size Sensitivity.** In order to see how TimeStone behaves with varying dataset size, we ran the hash table benchmark for 10× and 100× larger dataset (*i.e.*, 100K and 1M elements) than Figure 5. As shown in Figure 11, TimeStone consistently scales even for the 10× and 100× larger datasets. However, we observe ~8× drop in the overall throughput for 1M elements; further analysis reveals high cache miss rate (~60%) for 1M elements due to increased memory footprint compared to mere 2% for 100K elements. Overall, TimeStone shows a stable performance and scalability for both small (Figure 5) and large dataset (Figure 11).

**Log Size Sensitivity.** To see how TimeStone behaves with different log sizes, we ran the hash table benchmark with the read-intensive workload. We varied all three log sizes from 1/8× to 8× and measured throughput, the amount of reclamation, and the number of triggered reclamation. We did not observe any drop in performance even when log size is reduced by 1/8×. The amount of log reclamation is about the same because we ran the same workload. The number of times the log reclamation triggered increases proportionally with the decrease in the log size. For example, if the log size is reduced by 4× then the number of times the log reclamation is triggered increases by roughly 4×. Since our log reclamation is asynchronous triggering it more frequently does not impact the throughput of the system. Overall, our effective log reclamation technique makes TimeStone insensitive to the varying log sizes and achieves a high throughput even for a smaller log size.

## 6 Related Work

**NVMM Optimized Logging.** There have been a lot of active research in storage systems to optimize the logging protocols for NVMM [5, 28, 40, 46, 48, 68, 69, 72, 73, 78, 87]. Such log optimization techniques consider NVMM as a fast caching layer for the disk and leverage it for reducing the recovery cost or the durability cost incurred in the traditional disk-based logging. Techniques such as [28, 68, 87] proposes asynchronous commit policies to reduce the durability cost and to hide the long latency disk persist operations. Other techniques such as [5, 40, 73] leverages NVMM to correctly restore the partial disk writes upon recovery. While the database log optimization techniques primarily focuses on reducing the durability cost, we in TimeStone propose *TOC* logging which is geared not only towards reducing the durability cost but also focuses on reducing write amplification to achieve better performance and scalability.

**FASE Techniques.** Another line of research for developing crash consistent NVMM applications utilize a failure-atomic critical section (FASE) approach, guaranteeing atomicity at the level of a critical section granularity [12, 23, 27, 49, 58]. This approach focuses on providing failure atomicity to the legacy lock-based code with little or no focus on the scalability and write amplification issues. Moreover, the traditional FASE-based techniques such as [12, 23, 27, 49] suffers from complex runtime dependency tracking. While the state-of-art iDO logging [58] reduces the dependency tracking overhead but still it needs a specialized compiler support.

**Hardware Assisted Techniques.** This class leverages STM- or FASE-based approaches and propose new hardware support for guaranteeing atomic durability [22, 35, 37, 39, 50, 51, 64, 67, 76, 80, 95]. They interface with hardware buffers to speed up logging [39, 80, 95] or delegate the process of ordering stores to hardware [22, 50, 51, 64], clearly demanding significant hardware changes and introducing new logging instructions. Some approaches in this class propose extending hardware transactional memory (like Intel RTM) for making atomic updates to NVMM [38, 38, 53]. The performance of these techniques are bound by the L1-L3 cache size and requires changes in the existing cache-coherence protocol [38]. Unlike these techniques, TimeStone is completely software-based capable of running on the modern hardware.

## 7 Conclusion

In this paper, we propose TimeStone, a scalable and high-performing DTM framework. We propose *TOC logging* to keep write amplification under the check. MVCC-based design helps TimeStone to achieve better scalability and full-data consistency. Also, we support three different isolation levels to improve the applicability of TimeStone. We evaluated the TimeStone against all of the latest DTM works and we showed that TimeStone outperforms all of them upto 40× and shows a better scalability. While the prior DTM systems suffers from 2×-6× write amplification, TimeStone maintains it below 1. We also presented the real world impact of TimeStone by evaluating it with KyotoCabinet and YCSB workloads. The TimeStone enabled KytoCabinet and B+-tree shows better performance and scalability. We will open source TimeStone.

## Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This work was partially supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

## References

[1] 2011. Kyoto Cabinet: a straightforward implementation of DBM. (2011). http://fallabs.com/kyotocabinet/

[2] Anandtech. 2018. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here! (2018). https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here

[3] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[4] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. In *Proceedings of the 44rd International Conference on Very Large Data Bases (VLDB)*. Rio De Janeiro, Brazil.

[5] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*. New Delhi, India.

[6] Paul Von Behren. 2015. NVML: Implementing Persistent Memory Applications. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA.

[7] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Amsterdam, Netharlands, 677–694.

[8] Nan Boden. 2018. Available first on Google Cloud: Intel Optane DC Persistent Memory. (2018). https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory

[9] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages.

[10] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue* (2008), 40:46–40:58.

[11] Dhruva Chakrabarti, Haris Volos, and Indrajit Roy. 2016. How Should We Program Non-volatile Memory? (2016). https://pldi16.sigplan.org/event/tutorials-how-should-we-program-non-volatile-memory-

[12] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 25th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, Oregon.

[13] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*. Hawaii, USA.

[14] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2016. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[15] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, MT.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[17] Intel Corporation. 2019. Intel® 64 and IA-32 Architectures Optimization Reference Manual. (2019).

[18] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*. Vienna, Austria.

[19] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[20] David Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2015. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*. ACM, Prague, Czech Republic, 188–197.

[21] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. [n. d.]. System Software for Persistent Memory *(EUROSYS14)*.

[22] E. R. Giles, K. Doshi, and P. Varman. MSST15. SoftWrAP: A lightweight framework for transactional support of storage class memory.

[23] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions *(PLDI18)*.

[24] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA, 913–928.

[25] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software Persistent Memory. In *ATC12*.

[26] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270–1285.

[27] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. Belgrade, Serbia.

[28] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. (Sept. 2014), 389–400.

[29] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[30] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, California, USA, 187–200.

[31] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, California, USA.

[32] INTEL. 2019. Persistent Memory Development Kit. (2019). http://pmem.io/

[33] INTEL. 2019. PMDK man page: pmemobj_alloc. (2019). http://pmem.io/pmdk/manpages/linux/v1.5/libpmemobj/pmemobj_alloc.3

[34] INTEL. 2019. Valgrind: an enhanced version for pmem. (2019). https://github.com/pmem/valgrind

[35] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[36] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance

Measurements of the Intel Optane DC Persistent Memory Module. (2019). https://arxiv.org/abs/1903.05714v2

[37] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-assisted Logging with Asynchronous and Direct-update for Persistent Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka, Japan, 520–532.

[38] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Los Angeles, California.

[39] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*. Barcelona, Spain.

[40] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. (Aug. 2017), 135–148.

[41] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA.

[42] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2018. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Toronto, Canada.

[43] Sudarsun Kannan, Moinuddin Qureshi, Ada Gavrilovska, and Karsten Schwan. 2016. Energy Aware Persistence: Reducing Energy Overheads of Memory-based Persistence in NVMs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, New York, NY, USA, 165–177.

[44] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*. ACM, Porto, Portugal, Article 34, 15 pages.

[45] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Providence, RI, 779–792.

[46] Junghoon Kim, Changwoo Min, and Young Ik Eom. 2014. Reducing excessive journaling overhead with small-sized NVRAM for mobile devices. *IEEE Transactions on Consumer Electronics* 60, 2 (2014), 217–224.

[47] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. San Francisco, CA, USA, 1675–1687.

[48] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[49] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2018. Language-level Persistency. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Toronto, Canada.

[50] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[51] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan.

[52] K. Korgaonkar, I. Bhati, H. Liu, J. Gaur, S. Manipatruni, S. Subramoney, T. Karnik, S. Swanson, I. Young, and H. Wang. 2018. Density Tradeoffs of Non-Volatile Memory as a Replacement for SRAM Based Last Level Cache. In *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Los Angeles, California.

[53] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. 2006. Hybrid Transactional Memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, NY, USA, 209–220.

[54] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. (June 2009), 2–13.

[55] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA.

[56] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*. ACM, Chicago, Illinois, USA, 21–35.

[57] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China.

[58] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Fukuoka, Japan, 258–270.

[59] M. Seltzer and V. Marathe and S. Byan. 2018. An NVM Carol: Visions of NVM Past, Present, and Future. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*. Paris, France, 15–23.

[60] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Monterey, CA, 168–183.

[61] Paul E. McKenney. 2012. RCU Linux Usage. (2012). http://www.rdrop.com/~paulmck/RCU/linuxusage.html

[62] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. [n. d.]. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx *(EuroSys17)*.

[63] Micro. 2019. 3D XPoint Technology. (2019). https://www.micron.com/products/advanced-solutions/3d-xpoint-technology

[64] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China.

[65] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*. Boston, MA.

[66] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dali: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Conference on Distributed Computing (DISC)*. Vienna, Austria.

[67] Tri M. Nguyen and David Wentzlaff. 2018. PiCL: A Software-transparent, Persistent Cache Log for Nonvolatile Main Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on*

*Microarchitecture (MICRO)*. Fukuoka, Japan, 507–519.

[68] M. A. Ogleari, E. L. Miller, and J. Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the 24th IEEE Symposium on High Performance Computer Architecture (HPCA)*. vienna, Austria, 336–349.

[69] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite Optimization with Phase Change Memory for Mobile Applications. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*. Hawaii, USA, 1454–1465.

[70] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. San Francisco, CA, USA.

[71] Ismail Oukid and Wolfgang Lehner. 2017. Data Structure Engineering For Byte-Addressable Non-Volatile Memory. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*. Chicago, Illinois, USA.

[72] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. 2017. SQL Statement Logging for Making SQLite Truly Lite. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*. TU Munich, Germany, 513–525.

[73] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. (Aug. 2013), 121–132.

[74] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. New York, New York.

[75] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. (June 2009).

[76] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Waikiki, Hawaii.

[77] ScyllaDB / seastar. 2015. Exceptions are not scalable #73. (2015). https://github.com/scylladb/seastar/issues/73

[78] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China.

[79] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC Using Fine-Granular High-Frequency Virtual Snapshotting. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*. Houston, TX, USA, 245–258.

[80] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Cambridge, MA.

[81] Seunghee Shin, James Tuck, and Yan Solihin. 2018. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. Toronto, Canada.

[82] Tom Talpey and Andy Ruddof. 2018. Advanced Persistent Memory Programming: Local, Remote and Cross-Platform. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, California, USA. https://www.usenix.org/conference/fast18/training-program

[83] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Farmington, PA, 18–32.

[84] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harad a, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*. Houston, TX, USA.

[85] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2016. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA.

[86] Qi Wang, Timothy Stamler, and Gabriel Parmer. 2016. Parallel Sections: Scaling System-level Data-structures. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. ACM, London, UK, 33:1–33:15.

[87] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*. Hangzhou, China.

[88] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*. Houston, TX, USA.

[89] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, TU Munich, Germany, 781–792.

[90] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China.

[91] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA.

[92] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2015. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey.

[93] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, Hangzhou, China, 209–220.

[94] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey.

[95] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. 2013. 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)*. Davis, CA, USA, 421–432.

[96] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA.