

# Nonblocking Persistent Software Transactional Memory


H. Alan Beadle   Wentao Cai   Haosen Wen   Michael L. Scott<sup>1</sup>

*Department of Computer Science*

*University of Rochester*

Rochester, NY, USA

{hbeadle,wcai6,hwen5,scott}@cs.rochester.edu

<sup>1</sup>  <https://orcid.org/0000-0001-8652-7644>

**Abstract**—Newly emerging nonvolatile alternatives to DRAM raise the possibility that applications might compute directly on long-lived data, rather than serializing them to and from a file system or database. To ensure crash consistency, such data must, like a file system or database, provide failure-atomic transactional semantics. Several persistent software transactional memory (STM) systems have been devised to provide these semantics, but only one—the OneFile system of Ramalhete et al.—is *nonblocking*. Nonblocking progress is desirable to avoid both performance anomalies due to process preemption or failures and deadlock due to priority inversion. Unfortunately, OneFile achieves nonblocking progress at the cost of  $2\times$  space overhead, sacrificing much of the cost and density benefit of nonvolatile memory relative to DRAM. OneFile also requires extensive and intrusive changes to data declarations, and works only on a machine with double-width compare-and-swap (CAS) or load-linked/store-conditional (LL/SC) instructions.

To address these limitations, we introduce QSTM, a non-blocking persistent STM that requires neither the modification of target data structures nor the availability of a wide CAS instruction. We describe our system, give arguments for safety and liveness, and compare performance to that of the Mnemosyne and OneFile persistent STM systems. We argue that modest performance costs (within a factor of 2 of OneFile in almost all cases) are easily justified by dramatically lower space overhead and higher programmer convenience.

**Index Terms**—transactional memory, non-volatile memory, lock-free algorithm

## I. INTRODUCTION

For more than 40 years, computer memory has consisted primarily of DRAM, but the technology is nearing the end of its evolutionary life. Over the course of the coming decade, many uses of DRAM are expected to migrate to various new technologies, including phase-change memory (PCM) [4], [35], resistive RAM (ReRAM, a.k.a. memristors) [52], and spin-transfer torque magnetic memory (STT-MRAM) [2]. In addition to improving both density and power consumption, these newer alternatives are also *nonvolatile*—they keep their contents when the power is turned off. Nonvolatility raises the intriguing possibility that instead of being read from and written to a file system or database, long-lived data might “simply remain in memory” across program runs and even system crashes.

This work was supported in part by NSF grants CCF-1422649, CCF-1717712, and CNS-1900803, and by a Google Faculty Research award.

The principal obstacle to realizing this possibility is that coherence is traditionally implemented *on top of* a system’s caches, while memory resides below. Caches write their contents back to memory at unpredictable times. Unless a program takes explicit remedial action, the contents of memory in the wake of a crash are likely to be inconsistent, and thus unusable.

To enable such remedial action, computer architects are beginning to offer programmers fast and fine-grained control over the ordering and timing of writes-back from volatile caches into nonvolatile main memory; the semantics of this ordering comprise the *memory persistency model* [46] analogous to traditional memory consistency [1]. Various schemes and their hardware include epoch persistence [35], buffered epoch persistence [32], [46], explicit epoch persistence [31], DPO [34], and HOPS [42].

On top of these persistency models, several research groups have built high performance software for persistent applications. Such software is generally designed to provide *durable linearizability*, a safety criterion that requires atomic operations to be reflected in persistent memory (in linearization order) prior to returning to their callers [16], [31]. (In some cases, a more relaxed, *buffered* variant of this criterion may be used instead.) Example uses of persistence include concurrent data structures [8], [43], [44], [50] transactional key-value stores [33], [36], [56], [61], and the metadata of file systems [10], [59], [60] and databases [12], [47], [58]. Like many traditional concurrent data structures, these examples have been crafted by hand to maximize performance. Their crash consistency, in particular, is ensured through careful, parsimonious use of write-back and fencing instructions.

In contrast with these high-performance and specialized applications, a growing body of work, including that reported here, is designed to allow existing concurrent data structures—and collections of such structures—to be easily adapted to persistence, thereby avoiding the need to serialize to and from block-structured storage. Some of this general-purpose work is based on locks, with failure atomicity guaranteed for outermost critical sections. Atlas [7] uses UNDO logging at the level of individual loads and stores. NVthreads [25] uses REDO logging via page-level copy-on-write. JUSTDO [29] logs the program counter prior to each persistent store, and uses the code of the original application to push each critical section

through to completion during crash recovery. iDO [37] extends JUSTDO by leveraging compiler support to log only at the boundaries of *idempotent* instruction sequences. Extensions to some of these systems explore how to compose operations on hand-optimized persistent data structures, allowing them to be incorporated into larger failure atomic sections. For data structures that provide *detectable execution* [16], query-based logging [28], [30] allows UNDO and JUSTDO systems to support this composition in a manner analogous to “boosting” in software transactional memory [21], [23].

Other general-purpose systems assume a transactional programming model, with speculatively concurrent execution of programmer-specified atomic blocks. Mnemosyne [57], NV-Heaps [9], SoftWrAP [17], NVML [49], OneFile [48], and Romulus [11] extend the isolation+consistency semantics of transactional memory with (failure) atomicity and durability, providing the full ACID guarantees [18] for fine-grain memory transactions. Mnemosyne emphasizes performance; its use of REDO logs postpones the need to flush data to persistence until a transaction commits, and its use of fine-grain locking (hash function-identified *ownership records*) leads to high concurrency for non-conflicting transactions. SoftWrAP, also a REDO system, uses shadow paging and Intel’s now deprecated `pcommit` instruction [27] to efficiently batch updates from DRAM to NVM. NV-heaps, an UNDO log system, emphasizes programmer convenience, providing garbage collection and strong type checking to help avoid pitfalls unique to persistence—e.g., pointers to transient data inadvertently stored in persistent memory. PMDK (formerly NVML), Intel’s persistent memory transaction system [26], [55], uses UNDO logging on persistent objects and implements several highly optimized procedures that bypass transactional tracking for common functions.

With the exception of OneFile, all of these systems are *blocking*: a preempted thread (or a crashed process in a multiprogrammed system) can stall the progress of all other users, and deadlock can occur if persistent data is accessed by threads or event handlers that may experience priority inversion. Several non-persistent STM systems have provided nonblocking progress—most with an object-based API [15], [24], [39], [40], [53]; a few with a word based API [19], [38]. To the best of our knowledge, OneFile is the only previous nonblocking *persistent* STM.

To ensure that transactions persist in the same order they occur at run time, OneFile serializes all update transactions, but allows any thread to *help* the current active transaction to complete. Completion entails writing modified values back to their “natural” locations in memory. To avoid the possibility that a slow helper will overwrite a newer value with a now-stale update, OneFile adds an extra word of metadata adjacent to *every word* of real data, and uses the x86 instruction set’s double-width (128-bit) compare-and-swap (CAS) instruction to update the data and metadata together, atomically. To allocate the metadata, OneFile requires the programmer to rewrite all shared data declarations, using a special macro for each field. In principle, this burden could be delegated to a compiler,

but the  $2\times$  space overhead would remain and, as observed by the OneFile authors, compiler independence is highly desirable from the perspective of portability and system administration. We also note that interleaved metadata would be impractical on ARM or Power processors, where the CAS-like load-linked and store conditional instructions are limited to 64 bits. Fine-grain interleaving would require every 64-bit datum to be split in half, and software emulation of a double-width CAS would impose an unacceptable performance penalty on the critical path of persistent operations.

To address the limitations of past work, we introduce the QSTM transactional system. QSTM prioritizes ease of use for legacy applications, providing nonblocking persistence for unmodified data structures with minimal programmer effort, space overhead, and hardware dependences. It draws partial inspiration from the RingSTM of Spear et al. [51], but replaces that system’s array-based REDO log with the lock-free persistent queue of Friedman et al. [16]. By allowing the queue to describe the writes of an arbitrary number of committed-but-not-yet-completed transactions, QSTM eliminates OneFile’s need for nonblocking helping, interspersed data/metadata, and double-width CAS instructions. It also eliminates the need for the programmer or compiler to rewrite data declarations. To manage dynamically allocated data, it leverages our prior work on the Ralloc [5] nonblocking memory allocator.

QSTM design details appear in Section II. Section III presents informal proofs of durable linearizability and lock freedom. Section IV compares the performance of QSTM to that of Mnemosyne and OneFile, revealing that serial bottlenecks limit the scalability of both nonblocking systems. Both, however, are tolerant of frequent preemption, making them attractive for low-contention but latency-intolerant applications that access persistent data on a multiprogrammed system. OneFile is faster by 1.5–2 $\times$ , but QSTM avoids its  $2\times$  space overhead and intrusive source-code changes. Given its pairing with Ralloc, QSTM may also be attractive for data shared between processes with independent failure modes (we discuss this possibility in Section II). Section V concludes and outlines goals for future work.

## II. DESIGN

QSTM is based on a simplified version of the persistent lock-free queue of Friedman et al. [16]. It also employs the Ralloc nonblocking persistent allocator [5], making it fully nonblocking. Its design was inspired by RingSTM [51], which we briefly describe here first.

*RingSTM* uses a globally shared ring buffer to log committed transactions. Each buffer entry contains a unique timestamp, a status (either complete or writing), and a Bloom filter representing the locations written by the corresponding transaction. Each ongoing transaction maintains a read filter and a write filter, but only the write filter is written into the ring buffer. Transactions validate by comparing their own read filter to the write filter of each transaction that has committed since the validating transaction started. If the filters intersect then the transaction must abort. The validation step is performed

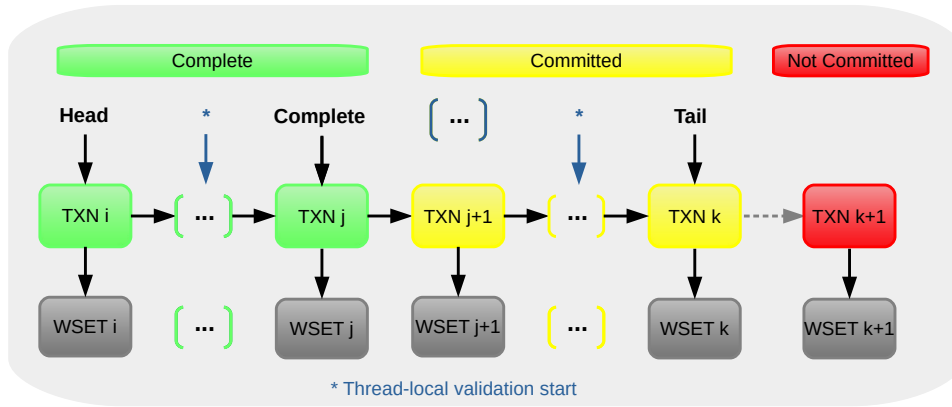


Fig. 1. QSTM Global Queue

during each transactional read operation and is repeated one more time before attempting to commit.

Before a RingSTM transaction validates for commit, it reads the current value of the global ring index. After validating, it performs an atomic compare-and-swap (CAS) to increment the index. If the CAS succeeds, then the transaction has reserved the corresponding buffer entry. If the CAS fails then it means that another transaction committed first and it is necessary to validate against it before making another attempt. All other threads must wait for the successful thread to populate the reserved entry before they can attempt to commit. This makes RingSTM fundamentally blocking.

*QSTM* introduces significant changes in order to achieve lock freedom and to accommodate a different underlying log. In particular, *QSTM*'s detailed write sets must be persistent and reachable from redo log entries in order to enable crash recovery and nonblocking progress. A diagram of the global queue used in *QSTM* can be seen in Figure 1. We explain the design of this structure below. Declarations and pseudocode for *QSTM* appear in Figures 2, 3, and 4.

In RingSTM, transaction log entries are populated after they have been reserved by some thread using CAS. This is not acceptable in a nonblocking system, since other threads are forced to wait until the reserving thread finishes writing to the entry. In *QSTM*, a transaction record, represented in a queue element, is fully initialized and persisted before being atomically enqueued. After the record is in the queue, other threads can commit without waiting for the prior committing thread to do anything more.

In RingSTM, each thread is responsible for completing its own writes after successfully committing a transaction into the ring. The same holds by default in *QSTM*. If a thread is preempted before it has completed its writes, however, all other write completion will be delayed until it is able to proceed. In practice, this is unlikely to be a significant problem in multi-threaded applications, which can generally assume that the underlying scheduler is fair. Crucially, even with such preemption, other threads can continue to commit transactions, reading values directly from write sets where appropriate.

```

1: class TRANSACTION
2:   QueueEntry *my_txn // queue entry for this transaction
3:   filter rf // read filter
4:   int start // logical start timestamp
5: end class
6: QueueEntry *head // global queue head
7: QueueEntry *tail // global queue tail
8: QueueEntry *complete // last known complete entry
9: class QUEUEENTRY
10:  int ts = 0 // commit timestamp
11:  filter wf // global copy of local write filter
12:  int st = writing // writing or complete
13:  writeset *wset // pointer to write set
14:  entry *next = NULL // pointer to next entry
15: end class
16: class WRITESSET
17:  addr[] // dynamic array of addresses to write
18:  vals[] // dynamic array of values to write
19: end class

```

Fig. 2. QSTM Structures

Note also that in *QSTM* it is possible for multiple threads to complete the writes of committed transactions concurrently, so long as those writes are to separate locations (otherwise conflicting writes might be performed out of order).

In addition to the usual head and tail queue pointers, *QSTM* maintains a complete pointer that refers to the most recent queue entry that is known to be marked complete. The complete pointer is used in `tm_start`, `check`, and `read` in order to eliminate the need for backward traversals of the redo log when validating.

*Garbage Collection:* We employ a periodic worker task that is occasionally triggered at the end of a successful call to `tm_commit` to perform garbage collection of queue entries. The worker task is performed by the thread that has just committed. The worker reclaims any queue entries that are marked complete and are not reserved, first dequeuing them and persisting the new head. All threads maintain a global reservation indicating the oldest queue entry that they might need to read in the future. The worker will free an entry only if it is older than the oldest reservation and is marked complete.

```

1: function TM_BEGIN
2:   my_txn = new QueueEntry
3:   my_txn→wset = new Writerset
4:   read tail to start
5: end function
6: function TM_READ(void* addr)
7:   success = False
8:   read *addr to memval
9:   if intersect(rf, my_txn→wf) then
10:     if my_txn→wset.addrs contains addr then
11:       return val from my_txn→wset.vals
12:     end if
13:   end if
14:   for all QueueEntry q from min(complete,start) to tail do
15:     if q.wf.contains(addr) then
16:       if q→wset contains addr then
17:         success = True
18:         read newestval from q.wset
19:       end if
20:     end if
21:     if intersect(rf, q.wf) and q.ts > start.ts then
22:       abort()
23:     end if
24:   end for
25:   rf.add(addr)
26:   if success then
27:     return newestval
28:   else
29:     return memval
30:   end if
31: end function
32: function TM_WRITE(void* addr, void* val)
33:   my_txn→wset.add(addr, val)
34:   my_txn→wf.add(addr, val)
35: end function
36: function TM_END
37:   if empty(my_txn→wset) then
38:     return
39:   end if
40:   PERSIST(my_txn→wset)
41:   PERSIST(my_txn)
42:   repeat
43:     read tail to oldtail
44:     if oldtail→next == NULL then
45:       check()
46:       read oldtail→next.ts to my_txn→ts
47:       success = CAS(oldtail→next, NULL, my_txn)
48:     else
49:       PERSIST(oldtail→next)
50:       CAS(tail, oldtail, oldtail→next)
51:     end if
52:   until success
53:   PERSIST(oldtail→next)
54:   wb_worker()
55:   if (my_txn → ts - head→ts) > Q_SIZE_LIMIT then
56:     gc_worker()
57:   end if
58: end function

```

Fig. 3. QSTM main methods

```

1: function CHECK
2:   read tail to curr_tail
3:   if curr_tail→ts==start→ts then
4:     return;
5:   end if
6:   for all QueueEntry q from start to curr_tail
7:     if intersect(rf, q.wf) then
8:       abort()
9:     end if
10:   end for
11:   read q to start
12: end function
13: function GC_WORKER
14:   for all QueueEntry q from head to complete do
15:     if !reserved(q) then
16:       dequeue(q)
17:       free(q→wset)
18:       free(q)
19:     end if
20:   end for
21: end function
22: function WB_WORKER
23:   for all QueueEntry q from complete→next to tail do
24:     read complete to oldcomplete
25:     if CAS(q.st, Not Writing, Writing) then
26:       q→wset.do_writes()
27:       q.st = Complete
28:       CAS(complete, oldcomplete, q)
29:     else
30:       return
31:     end if
32:   end for
33: end function

```

Fig. 4. QSTM helper functions

We were able to use a somewhat simplified version of the Friedman et al. lock-free queue. Because we don't actually need the content of dequeued entries, we can prune old nodes by CAS-ing head to the next node and then freeing the dummy node. (This also differs from the typical dequeue operation used in a Michael & Scott queue [41].) This method of dequeuing has the added advantage of allowing us to remove multiple elements with a single CAS.

*Post-crash Recovery:* After a crash, it is necessary only to recover the head pointer and any entries that can be reached from it (in addition to any persistent memory required by the application, but this is application-specific). All other memory formerly used by QSTM can be reclaimed. The writes from reachable entries must be completed and persisted. After that the queue can be reinitialized to contain a single dummy entry and execution can resume.

*Cross-application Persistence:* Longer term, we envision QSTM being used for data that is shared among mutually untrusting applications, with mutual isolation provided by a *protected library* mechanism. Our Hodor system, for example [22], ensures that shared data can be accessed only when executing trusted library code, and guarantees that library calls will execute to completion prior to preemption or process termination.

On a Hodor system, a QSTM thread can be confident of completing only a bounded number of writes—call it  $k$ —in a given library call. Between calls, it is vulnerable to preemption or to termination due to an error in another application thread. QSTM therefore allows *any* thread to complete the writes and persistence operations of any transaction. To facilitate this, each queue entry contains a pointer to the corresponding write set, each write set has a lock, and each write set entry indicates whether its datum has been written back to its natural location. A thread that wishes to complete the writes (e.g., because they stand in the way of completing its own subsequent writes) performs a Hodor call that attempts to acquire the lock and complete the next  $k$  writes. It releases the lock before returning from the library call.

Note that this “helping” mechanism is *not* required for nonblocking progress. Rather, it ensures, given Hodor’s guarantee of library call completion, that the space consumed by yet-to-be-completed writes will never grow without bound. It is also simple and fast, and requires no changes to user data structures. A multi-word CAS operation (KCAS) could in principle be used to complete a write and mark it as completed in the write set, thereby avoiding the need to lock write sets and ensuring that the queue remains bounded at all times. This strategy, however, would impose significant overhead, and would require a reserved bit in every word of user data [14], [45]. The nonblocking OneFile system [48] avoids the extra overhead, but as noted in Section I it requires a full word of metadata adjacent to every data word, doubling space consumption.

### III. PROOFS

In this section we provide arguments for several properties of QSTM.

*Theorem 3.1:* QSTM, seen as a single concurrent object, is linearizable.

*Proof:* In QSTM, the logical value of a memory location is the value found in a write set belonging to a transaction record nearest to the tail of the queue, if any such entry is in the queue. Otherwise the logical value is the value present in main memory at the specified location. A QSTM transaction must read only the current logical values of all locations that it reads and atomically update the logical values of all locations that it writes.

A transaction record becomes visible to other threads only when the corresponding transaction record is made reachable in the queue by an atomic CAS. It is sufficient to show that if this is successful then the committing transaction can be linearized after the transaction that precedes it in the queue, so that the queue always represents a correct linearization of all transactions contained therein.

Every time some transaction  $T$  performs a read of an address to which it has not written, it traverses the queue to ensure that no other transaction has committed a write that causes a conflict. If such a write has been committed then  $T$  will abort. Otherwise, the read is guaranteed to return the newest value for the address being read, even if the writes

have not yet been completed (that is, even if the most recent committed transaction to have written to that address is not yet in the complete state).

The check method validates against all queue entries against which the calling transaction has not yet validated. The read method contains similar checks, but also searches each non-complete entry for writes to the address that is being read. When validating, the transaction will abort if any value that has been read in the past has been modified since. This is sufficient to ensure that a transaction that may have read inconsistent values will not continue execution.

When a transaction attempts to commit, check is called prior to each CAS attempt but after determining the pointer on which to perform the CAS. This ensures that if the CAS succeeds, then the newly added transaction can be correctly linearized immediately following the transaction that precedes it in the queue. The successful CAS is the linearization point of the transaction. Thus, the queue will always represent a valid linearization of the transactions whose records it contains, and QSTM is linearizable. ■

*Theorem 3.2:* QSTM is durably linearizable.

*Proof:* We are aware of two published definitions for durable linearizability. Friedman et al. [16] showed that the two definitions are equivalent. The earlier definition by Izraelevitz et al. [31] is couched in terms of well-formed abstract histories, which are sequences of events that may include invocations and responses of object methods as well as full-system crashes. The more recent definition by Friedman et al. instead states that each operation must persist between its invocation and response, and that for any execution on an object, there must exist a linearization of that execution that matches the persistence order. The definition of Friedman et al. does not extend easily to *buffered* durable linearizability [31], but it is more convenient for the unbuffered case, so we use it in our proof.

We are concerned only with `tm_end`, as it is the only QSTM call that affects persistent data. To show that QSTM is durably linearizable, it suffices to show that for any multithreaded execution  $E$ : (1) The persist point of each operation (transaction) is between the invocation and response (of `tm_end`); (2) There exists a linearization of  $E$  whose order of operations matches the persistence order of the operations in  $E$ .

The first property is straightforward. We first assume that any queue entries added during prior calls to `tm_end` persisted in queue order, meaning that all reachable queue entries (including the pointers that link them) are persistent. A transaction record becomes persistent upon the `PERSIST` of the next pointer of the prior transaction (line 53 in Listing 3), since the transaction record itself will already have persisted by this time as shown in lines 40 and 41 in Listing 3. Thus the record can be fully recovered as long as the pointer persisted before the crash. This is guaranteed to occur before the end of `tm_end`, satisfying the first property.

The second property is also straightforward. We again assume that any queue entries added by prior calls to `tm_end` already persisted in queue order. As lines 49 and 50 in

Listing 3 show, `tm_end` always persists the newest next pointer before it “fixes” the outdated tail pointer, which is always done before any thread attempts to CAS another entry into the queue (all threads ensure that the tail is fully updated before attempting a CAS). Hence the persistence order of queue entries will always be the same as the queue order, which we have already shown is a valid linearization. ■

*Theorem 3.3:* QSTM is lock free.

*Proof:* To show that QSTM is lock free, it suffices to show (in the absence of recursion) that each loop can execute an unbounded number of times only if some other thread completes an operation. We will show that this is true for each loop in QSTM.

The outermost loop in a transaction is the transaction itself. That is, if a transaction aborts, it returns to the point where `tm_begin` was called and retries the transaction from there. The only places where an abort can occur are: (1) when `check` is called from `tm_end` or (2) during validation in `tm_read`. In both cases the transaction checks whether the read filter of the current transaction intersects with the write filter of any transactions that have committed since the call to `tm_begin`. If no other transactions have successfully committed, the transaction will not abort.

Another loop in the algorithm is the CAS retry loop in `tm_end`. This loop continues until either the transaction record is successfully enqueued or until the transaction aborts due to a conflict. Since the CAS will fail only if some other thread succeeds, this is lock free.

All other loops in QSTM are used to traverse the queue and will always traverse a finite number of entries since they read the timestamps at both ends of their traversal before starting. Thus, QSTM is lock free. ■

#### IV. PERFORMANCE RESULTS

We describe two sets of experiments. The first set compares QSTM to an earlier persistent STM, Mnemosyne [57]. Mnemosyne is performance-oriented but blocking; its throughput provides a useful baseline against which to compare nonblocking alternatives. Our second set of experiments compares QSTM to OneFile [48]—to our knowledge the only other persistent STM that provides nonblocking progress. We have made the QSTM sources available at <https://github.com/beadleha/qstm>.

##### A. Mnemosyne Comparison

Mnemosyne is based on the TinySTM system of Felber et al. [13]. As noted in Section I, it tracks conflicts at the granularity of individual words and uses a REDO log to avoid the need to fence each individual log entry. As noted in Section II, QSTM employs the Ralloc [5] persistent memory allocator in order to keep the overall system nonblocking. With Mnemosyne we used the Makalu [3] persistent memory allocator, which substantially outperforms the original built-in allocator. Rather than use Mnemosyne’s compiler-based transaction annotations, we made direct calls to the Mnemosyne API in order to ensure that transaction boundaries were identical in both systems.

*Hardware Platform:* These tests were run on a machine equipped with two Intel Xeon E7-2699 (2.2 GHz) CPUs, each with 18 cores and two hyperthreads per core, for a total of 72 hardware threads. The machine was equipped with 20 GB of memory. Binaries were compiled with gcc 6.3.1.

Although we have access to a machine equipped with persistent memory, we were unable to collect meaningful results for Mnemosyne on that machine due to highly unusual performance variation which was only present with Mnemosyne. However we were able to use the NVM-equipped machine for the comparison to OneFile below.

*Benchmark Suite:* We ran Mnemosyne comparisons on the Vacation and Intruder applications from the STAMP suite [6], together with several microbenchmarks: a stack, a queue, an ordered list, and a hash map.

For **Vacation**, we ran tests of 1,000,000 transactions, with five queries per transaction, 16,384 relations, 90 percent of relations queried, and 98 percent user transactions. We encountered an unusual problem when running the benchmark with Mnemosyne [57] and using the Makalu [3] allocator. The alignment of allocated structures resulted in a high rate of conflicts because the addresses were not being effectively hashed across the entire range of locks (orecs) in Mnemosyne. This problem did not occur with the default Mnemosyne allocator, nor did it occur with any of the other benchmarks. We solved the problem by making a minor adjustment to Mnemosyne’s hash function for this benchmark.

For **Intruder**, we used 10 percent traffic flows with injected attacks, 128 packets per flow, and 100,000 traffic flows.

The **stack** microbenchmark is a transactional variation on the linked Treiber Stack [54]. Each transaction is either a pop or a push. We read the address of the head node’s successor and switch the head pointer to it in a pop; we switch the head pointer to a newly created node whose successor is the old head node in a push. A pop operation frees the removed entry and a push operation allocates a new entry. Transactions on this data structure have high contention in comparison to the other three because every transaction must modify the head pointer.

The **queue** microbenchmark is a transactional variation on the M&S queue [41]. Each transaction is either an enqueue or a dequeue. We switch the tail (and the tail node’s successor) to the new node in an enqueue; we read the address of the next node and switch the head to it in a dequeue. Neither operation traverses the queue, operating only on the head or tail.

The **ordered list** microbenchmark is singly linked, and is based on Harris’s nonblocking algorithm [20]. Each transaction is either a remove or an insert. In a remove, we traverse the list until we find an equal or greater key, and remove the corresponding entry before freeing it; in an insert, we traverse the list and find the right place to replace or insert a node. If the node does not exist, a new one is allocated.

The **map** microbenchmark is a fixed-size hash map that uses the ordered list implementation for each bucket. In our test, we hash the key to find the right bucket and do a transactional

remove or insert. Transactions on this data structure have very low contention with a low rate of true conflicts between transactions.

*Methodology:* In all tests, we pinned the first 18 threads to different cores on one processor, then pinned the next 18 threads to the hyperthreads on the same processor, and then repeated this pattern on the second processor for the following 36 threads.

We ran several variations of each test. For QSTM we employed a “default” Bloom filter size of 2048 bytes, and then repeated the test with the filter sizes hand-tuned for best performance at 8 threads. We believe hand tuning to be a modest but reasonable burden; at the same time, all of our tests exhibited reasonable performance with the default size. One could also imagine dynamically tuning the filter size. Transitions from one size to another could be handled by aborting all ongoing transactions, clearing the queue, and resuming with the new size, or by maintaining up to two filters in each queue entry so that both sizes would be present when a size transition is taking place (the threads would need to maintain both sizes until no ongoing transaction was using only the old size).

To assess the impact of nonblocking progress, we also ran “preemption tests” in which there were 144 compute-intensive threads (two per hardware context) running at the same time as the benchmark. These additional threads were not pinned to any particular cores, although the benchmark threads were. This test demonstrates that preempted threads can never prevent their running peers from making progress. On the compute-intensive “competitor” threads, we ran a simple prime number generator. These competitor threads were killed and restarted prior to each trial.

The QSTM garbage collection worker interval was set to 50,000 transactions, but our implementation attempts to dequeue and reuse entries one by one to reduce allocator calls, so this threshold will rarely be reached under normal conditions. In our implementation a thread attempts to complete writes using the `wb_worker` routine after every successful commit, but the worker ends immediately if another thread holds the lock for the write set that it intends to complete writes from.

We ran each of these tests for 5 seconds and recorded the number of operations per second. All reported statistics reflect the average of three runs.

The stack, queue, and hash map benchmarks used a key range of 100,000. The list benchmark used a smaller key range of 10,000 due to the need to traverse the list in each transaction. In all four micro benchmarks, the structure was pre-filled to 50% of the key range.

*Discussion:* Results for all six benchmarks are shown in Fig. 5. The plots are organized to allow visual comparison of the non-preemption and preemption versions of each microbenchmark. The first and third rows of the figure contain the non-preemption plots; the second and fourth contain the preemption plots. The QSTM data labels indicate the Bloom filter size. For example, QSTM-128 indicates that a 128 byte filter size was used. For each of the six benchmarks, the same

scale was used for both the normal and preemptive graphs to allow for easier visual assessment of effect of increased CPU contention (however different benchmarks use different scales).

Most of the tests clearly show the limitations of QSTM’s global log. Some QSTM tests continued to gain throughput until 16 threads but it is clear that QSTM is best suited to small machines or to applications with limited contention. QSTM also struggles with large transactions due to the large Bloom filters needed to make false conflicts infrequent.

QSTM performed well on the linked list benchmark. This is because a QSTM transaction only aborts if some other transaction succeeded after the start of the aborting transaction, so despite the large number of aborts, some transaction always succeeds. The list benchmark requires each transaction to traverse the list to find the correct location to insert or remove a node, and if any locations read during the traversal are modified by another transaction, at least one transaction must abort. Also note that hand-tuning the Bloom filter size failed to improve on our default value, so the plots for the list microbenchmark show only two lines.

The limited scaling of QSTM is most evident in the hash map microbenchmark. QSTM fails to gain any throughput after 8 threads while Mnemosyne scales well up until some of the threads are assigned to the second socket.

The preemption tests illustrate the benefits of nonblocking progress. When application threads are frequently preempted, Mnemosyne experiences a dramatic loss in performance because locks might be held by a preempted thread. In QSTM all other threads can continue to make progress regardless of when a thread is preempted. Note also that QSTM throughput continues to increase well beyond the optimal number of threads seen in the non-preemptive tests. This is because each thread is running for only a fraction of the time, putting a reduced amount of traffic through the queue.

## B. OneFile Comparison

We also ran tests to compare performance to OneFile [48]. OneFile uses a global sequence counter and augments each data word with an adjacent reserved word to track modifications. When a data word is to be modified, the adjacent reserved word must also be changed in the same operation, requiring the use of a double-width CAS instruction. OneFile includes both wait-free and lock-free variants.

These tests used a different machine, equipped with two Intel Xeon Gold 6230 (2.1 GHz) CPUs, each with 20 cores and two hyperthreads per core, for a total of 80 hardware threads. The machine was also equipped with 1.536 TB of Intel Optane NVM and 384 GB of DRAM. Binaries were compiled with gcc 9.2.1.

OneFile allows transactions to construct write sets in parallel, but forces them to commit one at a time, and to help *complete* the writes of predecessors before proceeding. To support this helping mechanism, the programmer must rewrite the declarations of all shared data structures, and must specify every transaction as a lambda expression. The need for such

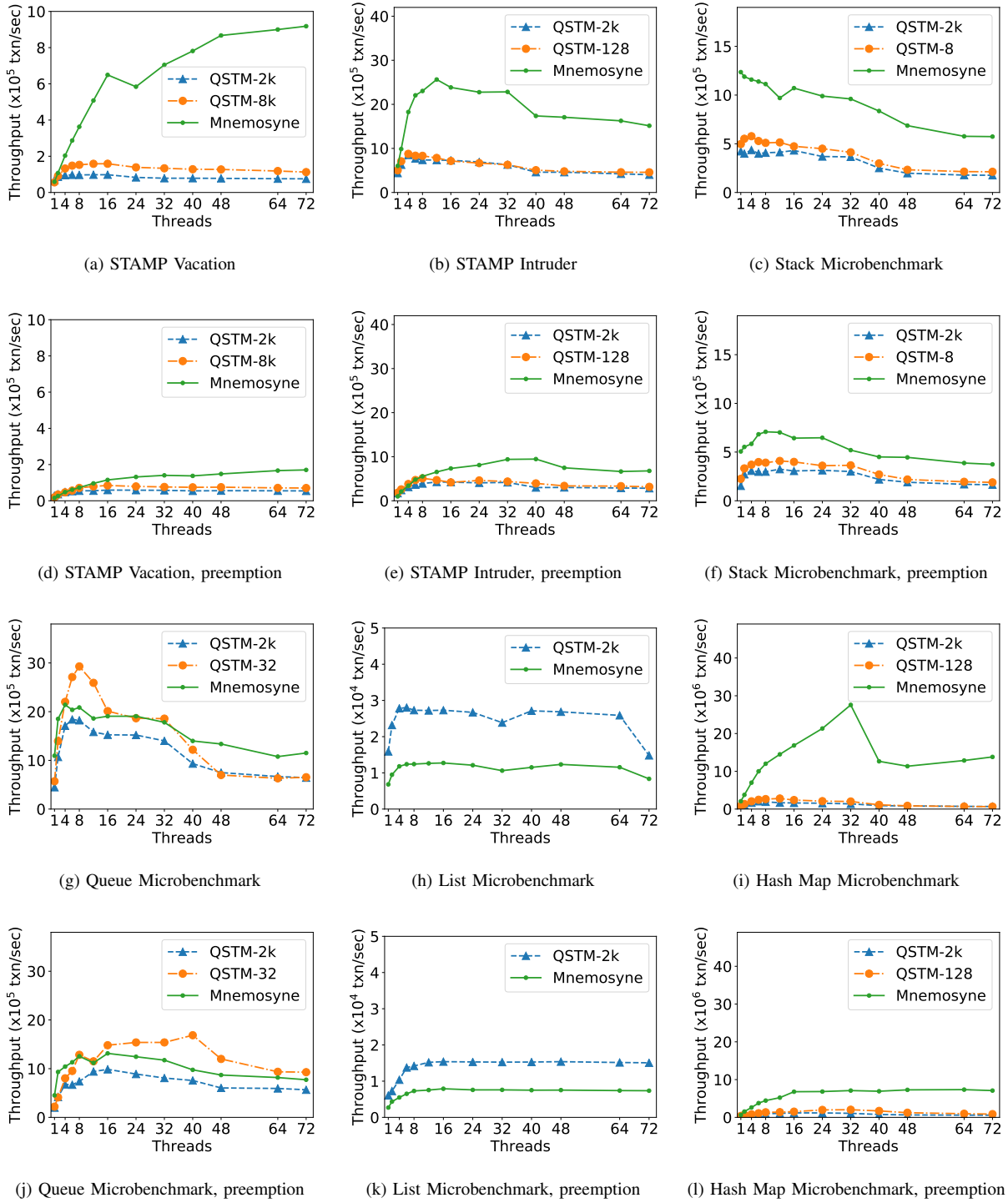


Fig. 5. QSTM vs Mnemosyne (transactions per second).

changes made it difficult to retrofit multiple microbenchmarks. Instead, we relied on the hash map microbenchmark included in the OneFile distribution, with minor modifications. In the original version, writers used two separate transactions to delete and replace a key while in our version these are done within a single transaction. The percentage of writer

transactions was varied among 100%, 50%, and 10%. We also ran a version in which writers replaced ten keys in each transaction to observe the effects of larger transactions. QSTM used a 128-byte Bloom filter, as in the Mnemosyne hash map tests.



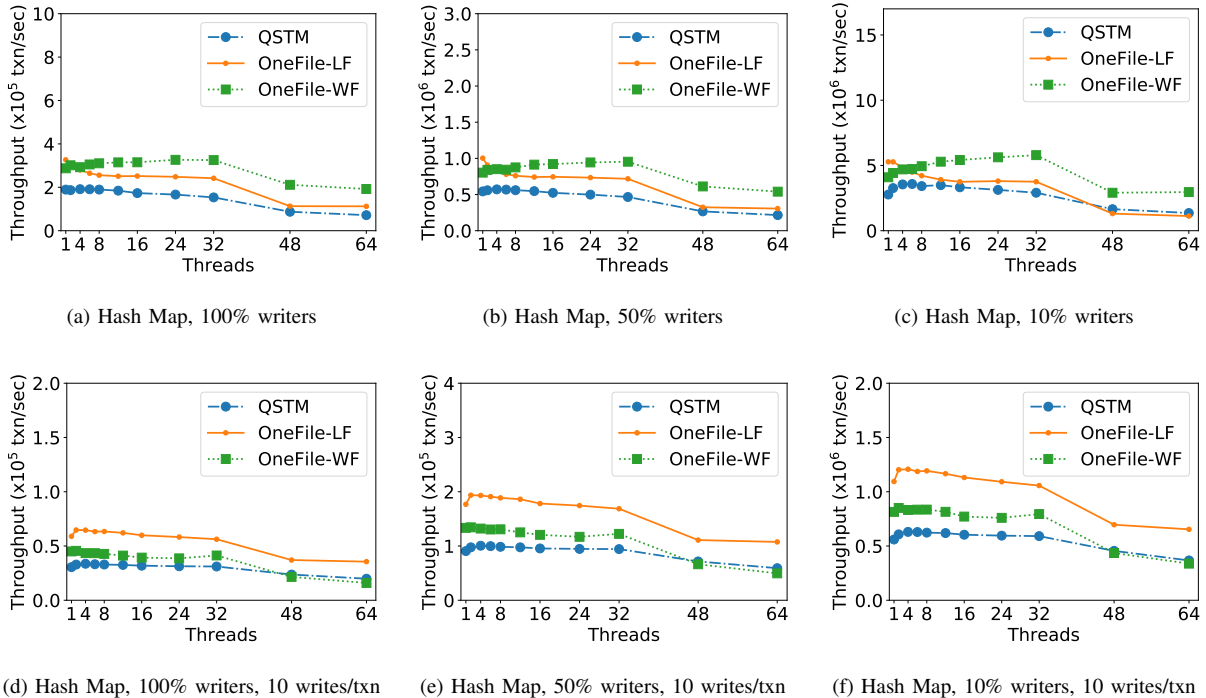


Fig. 6. QSTM vs OneFile (transactions per second).

*Discussion:* Results appear in Fig. 6. In the first row of plots, each write transaction replaces one key; in the second row, each write transaction replaces ten keys. Within each row the ratio of read and write transactions is varied as indicated in the figure captions. OneFile-LF and OneFile-WF correspond to the lock-free and wait-free variants of OneFile as described in the original paper.

OneFile achieves higher throughput than QSTM. This is due to high contention on QSTM’s global queue and to the smaller number of steps required to commit a OneFile transaction. The data structures needed for a QSTM transaction also impose a nontrivial amount of memory churn. OneFile, which permanently reserves space for versioning information inside the target data structures, has comparatively simple global metadata.

OneFile’s performance, however, comes with several costs—notably memory overhead. Per-word metadata *doubles* the size of all persistent data structures. The mechanisms used to declare this metadata introduce significant awkwardness; they also make OneFile problematic for applications written in C, or in any language that makes assumptions about data alignment. Individual variables within classes or structs must be derived from a special template and can be no larger than one machine word. If the application would normally use larger variables then they must be split into multiple pieces. OneFile’s wait-free variant requires that any thread be able to run the entirety of a committing transaction—specified as a lambda expression—on behalf of some other committing thread; this complicates the task of retrofitting existing data

structures, in which transactions may cross function boundaries and use thread-local transient variables. As a result of these restrictions, we found it prohibitively challenging to adapt the applications from our Mnemosyne comparison to use OneFile instead. In the end we simply gave up, and adapted OneFile’s benchmark to use QSTM.

We believe that QSTM will represent an acceptable tradeoff in cases where it is difficult or impractical to modify the data structure types used in a legacy program when making it persistent, or on architectures (e.g., ARM or Power) that lack a double-width CAS or store-conditional instruction.

## V. CONCLUSIONS AND FUTURE WORK

We have presented QSTM, a nonblocking persistent software transactional memory. Paired with the Ralloc [5] nonblocking persistent memory allocator, QSTM provides a complete solution for nonblocking management of persistent “in memory” structures. We are exploring ways in which to mitigate the unbounded memory usage that is possible when a thread is preempted while performing writes. One idea is to use a protected library such as Hodor [22] to guarantee completion of QSTM function calls. This would have the added safety benefit of ensuring that the persistent region is accessed only through QSTM.

QSTM provides nonblocking progress without the need to modify the layout of legacy data structures. Given OneFile’s use of an extra word for every data word, QSTM consumes only 50% as much space, allowing it to utilize the full capacity of NVM when retrofitting applications that previously used block storage. QSTM’s avoidance of double-wide CAS also

suggests that it would be easier than OneFile to adapt to architectures other than the x86, and its avoidance of templated data structure modification suggests it would be easier to adapt to languages other than C++. In both systems, scalability is limited by a fundamental serial bottleneck. More scalable alternatives will likely need to employ an object-oriented [15], [24], [39], [40], [53] or orec-based [19], [38] based approach.

## REFERENCES

- [1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.
- [2] D. Apalkov, A. Khvalkovskiy, S. Watta, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Kroumbi. Spin-transfer torque magnetic random access memory (STT-MRAM). In *ACM Journal on Emerging Technologies in Computing Systems (JETC)—Special issue on memory technologies*, pages 13:1–13:35, May 2013.
- [3] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 677–694, Amsterdam, The Netherlands, Nov. 2016.
- [4] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy. Phase change memory technology. *Journal of Vacuum Science and Technology*, 28(2):223–262, Mar. 2010.
- [5] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott. Understanding and optimizing persistent memory allocation. In *ACM SIGPLAN Intl. Symp. on Memory Management (ISMM)*, London, UK, June 2020.
- [6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, Sept. 2008.
- [7] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 433–452, Portland, OR, Oct. 2014.
- [8] S. Chen and Q. Jin. Persistent B+-trees in non-volatile main memory. *Proc. of the VLDB Endowment*, 8(7):786–797, Feb. 2015.
- [9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, Newport Beach, CA, Mar. 2011.
- [10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *22nd ACM Symp. on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.
- [11] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *30th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, Vienna, Austria, July 2018.
- [12] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dullloor. A prolegomenon on OLTP database systems for non-volatile memory. In *10th Intl. Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 57–63, Hangzhou, China, Sept. 2014.
- [13] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Trans. on Parallel and Distributed Systems*, 21(12):1793–1807, Dec. 2010.
- [14] S. Feldman, P. Laborde, and D. Dechev. A wait-free multi-word compare-and-swap operation. *Intl. Journal of Parallel Programming*, 43(4):572–596, Aug. 2015.
- [15] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):article 5, May 2007.
- [16] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *23rd ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Vienna, Austria, Feb. 2018.
- [17] E. R. Giles, K. Doshi, and P. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *31st Symp. on Massive Storage Systems and Technology (MSST)*, pages 1–14, Santa Clara, CA, June 2015.
- [18] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, Dec. 1983.
- [19] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 388–402, Anaheim, CA, Oct. 2003.
- [20] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *15th Intl. Symp. on Distributed Computing (DISC)*, pages 300–314, Lisbon, Portugal, Oct. 2001.
- [21] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *19th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 387–388, Orlando, FL, Feb. 2014.
- [22] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Usenix Annual Technical Conf. (ATC)*, pages 489–504, Renton, WA, July 2019.
- [23] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, USA, Feb. 2008.
- [24] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *22nd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 92–101, Boston, MA, July 2003.
- [25] T. C.-H. Hsu, H. Bruegner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical persistence for multi-threaded applications. In *12th European Conf. on Computer Systems (EuroSys)*, pages 468–482, Belgrade, Republic of Serbia, Apr. 2017.
- [26] Intel. Intel NVM Library. [github.com/pmem/nvml/](https://github.com/pmem/nvml/).
- [27] Intel Corporation. Intel architecture instruction set extensions programming reference. Technical Report 3319433-029, Intel Corporation, Apr. 2017.
- [28] J. Izraelevitz. *Concurrency Implications of Nonvolatile Byte-Addressable Memory*. PhD thesis, Dept. of Computer Science, Univ. of Rochester, Dec. 2018.
- [29] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *21st Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, Atlanta, GA, Apr. 2016.
- [30] J. Izraelevitz, V. J. Marathe, and M. L. Scott. Composing durable data structures (poster). In *8th Non-Volatile Memories Workshop (NVMW)*, San Diego, CA, Mar. 2017.
- [31] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *30th Intl. Symp. on Distributed Computing (DISC)*, pages 313–327, Paris, France, Sept. 2016.
- [32] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *48th Intl. Symp. on Microarchitecture (MICRO)*, pages 660–671, Waikiki, HI, Dec. 2015.
- [33] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *21st Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 399–411, Atlanta, GA, Apr. 2016.
- [34] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *49th Intl. Symp. on Microarchitecture (MICRO)*, pages 1–13, Taipei, Taiwan, Oct. 2016.
- [35] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *36th Intl. Symp. on Computer Architecture (ISCA)*, pages 2–13, Austin, TX, June 2009.
- [36] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th Usenix Conf. on File and Storage Technologies (FAST)*, pages 257–270, Santa Clara, CA, Feb. 2017.
- [37] Q. Liu, J. Izraelevitz, S. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *51st Intl. Symp. on Microarchitecture (MICRO)*, Fukuoka, Japan, Oct. 2018.
- [38] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *13th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 227–236, Salt Lake City, UT, USA, Feb. 2008.

- [39] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *19th Intl. Symp. on Distributed Computing (DISC)*, pages 354–368, Cracow, Poland, Sept. 2005.
- [40] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, Ottawa, ON, Canada, June 2006.
- [41] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, May 1996.
- [42] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *22nd Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 135–148, Xi’an, China, Apr. 2017.
- [43] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey, D. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. In *31st Intl. Symp. on Distributed Computing*, Vienna, Austria, Oct. 2017.
- [44] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *2016 Intl. Conf. on Management of Data (SIGMOD)*, pages 371–386, San Francisco, CA, June 2016.
- [45] M. Pavlovic, A. Kogan, V. Marathe, and T. Harris. Brief announcement: Persistent multi-word compare-and-swap. In *37th ACM Symp. on Principles of Distributed Computing (PODC)*, Egham, United Kingdom, July 2018.
- [46] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *41st Intl. Symp. on Computer Architecture (ISCA)*, pages 265–276, Minneapolis, MN, June 2014.
- [47] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *Proc. of the VLDB Endowment*, 7(2):121–132, Oct. 2013.
- [48] P. Ramalheite, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 151–163, 2019.
- [49] A. Rudoff. Persistent memory programming. [pmem.io/](http://pmem.io/). Accessed: 2017-04-21.
- [50] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *3rd VLDB Workshop on In-Memory Data Management and Analytics (IMDM)*, pages 4:1–4:8, Kohala Coast, HI, Aug. 2015.
- [51] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable transactions with a single atomic instruction. In *20th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–284, New York, NY, USA, June 2008.
- [52] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, June 2008.
- [53] F. Tabbà, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *21st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 204–213, Calgary, AB, Canada, Aug. 2009.
- [54] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.
- [55] U. U. and A. M. Rudoff. Introduction to Programming with Persistent Memory from Intel. [software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel](http://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel), Aug. 2017.
- [56] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th Usenix Conf. on File and Storage Technologies (FAST)*, page 5, San Jose, CA, Feb. 2011.
- [57] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *16th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, Newport Beach, CA, Mar. 2011.
- [58] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. of the VLDB Endowment*, 7(10):865–876, June 2014.
- [59] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th Usenix Conf. on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, Feb. 2016.
- [60] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *26th ACM Symp. on Operating Systems Principles (SOSP)*, pages 478–496, Shanghai, China, Oct. 2017.
- [61] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-tree: Reducing consistency cost for NVM-based single level systems. In *13th Usenix Conf. on File and Storage Technologies (FAST)*, pages 167–181, Santa Clara, CA, Feb. 2015.