Towards Efficient Nonblocking Persistent Software Transactional Memory

H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott

Technical Report #1006
Department of Computer Science, University of Rochester
{hbeadle,wcai6,hwen5,scott}@cs.rochester.edu

April 2019

Abstract

Nonvolatile memory technologies (NVM) will likely begin to displace dynamic RAM over the next few years. While developed largely for higher density and lower power, NVM can also allow data to persist across program runs and system crashes without the need to flush to disk or flash. If data is to be recovered after a crash, however, care must be taken to ensure that the contents of memory are consistent at all times. This can be challenging in multithreaded applications with write-back caches. Several persistent software transactional memory (STM) systems have been devised to address this problem. To our knowledge, however, all such prior works have been blocking. We present QSTM, a nonblocking word-based persistent STM. We describe our system and give arguments for safety and liveness. We compare QSTM’s performance to that of the Mnemosyne persistent STM.

1 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) [6,34], resistive RAM (ReRAM, a.k.a. memristors) [51], 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 files, long-lived data might “simply remain in memory” across program runs and even system crashes.

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 non-volatile main memory; the semantics of this ordering comprise the memory persistency model [46].

*This work was supported in part by NSF grants CCF-1422649 and CCF-1717712, and by a Google Faculty Research award.
analogous to traditional memory consistency [1]. Various schemes and their hardware include epoch persistence [34], buffered epoch persistence [31, 46], explicit epoch persistence [30], DPO [32], and HOPS [42].

On top of these persistency models, several research groups have built high performance software for persistent applications. Example projects include concurrent data structures [9,43,44,49], transactional key-value stores [33,35,55,60], file systems [11,58,59], and databases [13,47,57]. 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. Some of this general-purpose work is based on locks, with failure atomicity guaranteed for outermost critical sections. Atlas [8] uses UNDO logging at the level of individual loads and stores. NVthreads [24] uses REDO logging via page-level copy-on-write. JUSTDO [28] logs the program counter prior to each persistent store, and uses the code of the original application to push each critical section through to its completion during crash recovery. iDO [36] 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 [27, 29] allows UNDO and JUSTDO systems to support this composition in a manner analogous to “boosting” in software transactional memory [21,22].

Other general-purpose work assumes a transactional programming model, with speculatively concurrent execution of programmer-specified atomic blocks. Mnemosyne [56], NV-Heaps [10], SoftWrAP [17], NVML [48], and Romulus [12] 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. SoftWrAP, also a REDO system, uses shadow paging and Intel’s now deprecated pcommit instruction [26] 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 [25, 54], uses UNDO logging on persistent objects and implements several highly optimized procedures that bypass transactional tracking for common functions. All of these systems use locks internally.

PHyTM [3] is a hybrid TM based on PHTM [4]. PHTM and PHyTM both utilize hardware transactions to provide a convenient interface for persistent memory. PHTM uses an STM fallback path while PHyTM is a hybrid TM with multiple code paths and constraints on which of these paths can run concurrently. Unfortunately, both PHTM and PHyTM rely on hardware support for a “log bit” to be written to main memory atomically when a commit instruction executes. Our work does not rely on this feature, which to our knowledge is not expected to be available in commercial hardware anytime soon.

We present QSTM, a novel design for nonblocking durable software transactions, built around the use of a durable global REDO log. Our design draws partial inspiration from RingSTM [50] but is based on the persistent lock-free queue of Friedman et al. [16]. (Several non-durable STM systems have also been nonblocking, though most of these have been object based [15,23,37,38,52], with space reserved for metadata in each object header. A few have been word based [19,40].) QSTM is to our knowledge the first nonblocking persistent STM.
We describe QSTM’s design in detail in section 2. Section 3 presents informal proofs of durable linearizability and lock freedom for QSTM. In section 4 we give performance results and compare QSTM to Mnemosyne. Section 5 concludes and outlines several goals for future work.

Like RingSTM, QSTM is privatization safe [39]. Preliminary experiments, reported in Section 4, show that although QSTM suffers from fundamental scalability limitations, it is highly tolerant of frequent preemption. QSTM may be desirable in use cases where preemption tolerance is needed or when threads may fail independently, and it would be difficult to run a nontrivial recovery procedure in parallel with ongoing execution in other threads.

We are in the process of developing a nonblocking memory allocator for use with QSTM. Once it is available, the overall system will be entirely nonblocking. In the meantime, we have used a blocking allocator in our tests. We do not believe that this significantly affects our performance results.

2 Design

QSTM is based on a simplified version of the persistent lock-free queue of Friedman et al. [16], and is entirely nonblocking if used with a nonblocking memory allocator. QSTM borrows inspiration from RingSTM [50], replacing its ring buffer with the lock-free persistent queue.

RingSTM uses a globally shared ring buffer to log committed transactions. Each ring 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 checking for intersections between their read filter and the write filters of all transactions that have committed since the validating transaction started. If such an intersection exists then the transaction must abort. The validation step is performed during each transactional read operation and is repeated one more time before attempting to commit.

Before validating, a committing transaction reads the global ring counter index. After validating, it performs an atomic compare-and-swap (CAS to increment the counter. If successful, this CAS reserves the resulting index for that transaction. (If unsuccessful, additional validation is required.) All other threads must wait for the successful thread to populate the reserved entry before they can attempt to commit.

The differences between QSTM and RingSTM are substantial due to the nonblocking requirement and the change of underlying data structure. In particular, QSTM’s detailed write sets must be durable and reachable from redo log entries. This is important for crash recovery as well as for nonblocking progress.

In RingSTM, transaction log entries are populated after they have been reserved by some thread using CAS. This is not suitable for a nonblocking solution since the successful thread must populate the ring entry before other threads can continue. 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 successful thread to do anything more.

In RingSTM each thread is responsible for performing its own writes after successfully committing a transaction into the ring. In QSTM, we allow any thread to perform the writes and persistence operations of any transaction. To enable this, each queue entry contains a pointer to the corresponding write set. Note that since values can be read directly out of these write sets, progress in performing the committed writes is not necessary for the application to continue to make progress.
The possibility of several threads attempting to help perform the same writes poses a new problem. If multiple threads attempt to perform writes for the same transaction, then any writes occurring after the first thread has finished could cause an inconsistent state to be visible to the other threads (that is, an earlier write might undo a later write from the same transaction while the record is already marked as complete). Pruning the write sets to represent the minimal set of writes (e.g., at most one write to any single address) prevents this from occurring when several threads are performing writes for the same transaction, but it is still possible for a similar problem to occur when two threads concurrently perform writes for two different transactions with overlapping writes. This can be solved in several ways.

One solution would be to use a multi-word CAS, emulated by other primitives, similar to techniques presented by Feldman et al. [14] or Pavlovic et al. [45]. However doing this for every write in every transaction would most likely have very high overhead. Multi-word CASes also have the disadvantage of reserving one bit of each word of memory being modified.

QSTM instead solves the problem by locking a queue entry before performing the writes, and by performing write-back serially in order of commit time. If a thread stalls while holding a lock, the queue will begin to grow longer with each commit but application progress will not be prevented. Although in practice this could result in unbounded memory usage, it nonetheless provides nonblocking progress (assuming enough memory is available) and offers practical benefits which will be shown in section 4.

Each queue entry is tagged with a state, which can be any of not writing, writing, or complete. The initial state of a queue entry is not writing. When a thread is going to attempt to perform writes corresponding to a queue entry, it tries to CAS the state of that entry to writing. If it succeeds then it may perform the writes without any risk of overlapping with another writer. When it is done, it will set the state of that entry to complete, which indicates that the entry is no longer needed to ensure consistent reads.

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.

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 calling thread reclaims any queue entries that are marked complete and are not reserved, first dequeueing 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.

We were able to use a somewhat simplified version of the Friedman et al. lock-free queue. Because we don’t actually need to dequeue content, 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 dequeueing also allows us to remove a large number of elements at the same time using a single CAS.

Structures and pseudocode for QSTM are shown in listings 1 and 2 respectively. The function tm_write is unmodified from RingSTM [50] and is not shown.

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 performed and persisted. After that the queue can be reinitialized to contain a single dummy entry and execution can resume.
3 Proofs

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

**Theorem 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 performed (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.
Figure 2: Methods of Transaction class

Theorem 2. QSTM satisfies durable linearizability

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. [30] defines well-formedness of 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 the durability point of each operation must always fall between the invocation and response of that operation, and that for any execution on an object, there must exist a linearization of that execution which matches the durability order of the operations in the execution. The definition of Friedman et al. does not extend easily to buffered durable
linearizability [30], but it is more convenient for the unbuffered case, so we use it in our proof.

We are concerned only with \texttt{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 durability point of each operation (transaction) is between its invocation and response (of \texttt{tm\_end}); (2) There exists a linearization of $E$ whose order of operations matches the durability order of the operations in $E$.

The first property is straightforward. We first assume that any queue entries added during prior calls to \texttt{tm\_end} became durable in queue order, meaning that all reachable queue entries are durable including the pointers that link them. A transaction record becomes durable upon the \texttt{PERSIST} of the next pointer of the prior transaction (line 46 in Listing 2), since the transaction record itself will already be durable by this time as shown in lines 34 and 35 in Listing 2. Thus the record can be fully recovered as long as the pointer was persisted before the crash. This is guaranteed to occur before the end of \texttt{tm\_end}, satisfying the first property.

The second property is also straightforward. We again assume that any queue entries added by prior calls to \texttt{tm\_end} already became durable in queue order. As lines 43 and 44 in Listing 2 show, \texttt{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 record into the queue (all threads ensure that the tail is fully updated before attempting a CAS). Hence the durability order of queue entries will always be the same as the queue order, which we have already shown is a valid linearization.

\begin{theorem}
QSTM is lock free
\end{theorem}

\begin{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 \texttt{tm\_begin} was called and retries the transaction from there. The only places where an abort can occur are: (1) when \texttt{tm\_check} is called from \texttt{tm\_end} or (2) during validation in \texttt{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 \texttt{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 \texttt{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.
\end{proof}

4 Performance Results

We compare QSTM to Mnemosyne [56] on several benchmarks and microbenchmarks. All tests were conducted using the Makalu [5] persistent memory allocator, which has been shown to substantially improve the performance of Mnemosyne and was also convenient to use with QSTM. Rather than use Mnemosyne’s compiler-based transaction annotations, we made direct calls to the Mnemosyne STM in order to ensure that the usage of either STM was consistent across our tests.

We ran tests on the Vacation and Intruder applications from the STAMP suite [7], together with several microbenchmarks: a stack, a queue, an ordered list, and a hash map.
All of our tests were run on a machine equipped with two Intel Xeon E7-2699 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. All tests were built with GCC 6.3.1.

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. All reported statistics are produced from an average of three runs.

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. For the compute-intensive “competitor” threads, a simple brute-force prime number generator was used. These 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. Our implementation calls the write-back worker after every successful commit, but it returns immediately if it detects another thread performing writes.

For Vacation, we ran tests of 1,000,000 transactions, with 5 queries per transaction, 16384 relations, 90 percent of relations queried, and 98 percent user transactions. We encountered an unusual problem when running the benchmark with Mnemosyne [56] and using the Makalu [5] 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 [53]. 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 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 \texttt{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 \texttt{remove} or \texttt{insert}. Transactions on this data structure have very low contention with a low rate of true conflicts between transactions.

We ran each of these for a duration of 5 seconds and recorded the number of operations per second. 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). 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. All four of these benchmarks were prefilled to 50% of the key range.

All results are shown in figure 3. Most of the tests clearly show the limitations of QSTM’s global log. Some QSTM tests scaled to 16 threads but it is clear that QSTM is best suited to small machines or to applications with limited scaling potential. 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.

Mnemosyne’s performance on the hash map is unusual. In earlier tests the performance continued to improve to 32 threads before dropping, but when we applied 50% prefill the performance instead declined after 12 threads. We believe that this is due to the increased number of reads when traversing the lists in each bucket, causing aborts for similar reasons to the list benchmark.

The preemption tests illustrate one advantage of our nonblocking algorithm, even though the memory allocator is still blocking. When application threads are frequently preempted, Mnemosyne experiences more severe performance degradation 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 (aside from the memory allocator). 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.

5 Conclusions and Future Work

We have presented QSTM, a nonblocking persistent software transactional memory. If paired with a nonblocking persistent memory allocator, it would provide a complete solution for nonblocking management of persistent “in memory” structures. We are currently in the process of developing such an allocator. We are also exploring ways in which to mitigate the unbounded memory usage that is possible when a QSTM thread is preempted while performing writes.

Unfortunately, QSTM suffers from limited scalability due to its reliance on a global log. We are exploring more scalable designs for nonblocking persistent STMs. We also plan to investigate use cases in which it may be important to tolerate the failure of individual threads—e.g., when a persistent STM manages data that is shared among processes with independent failure modes.
Figure 3: Throughput (transactions per second).

- (a) STAMP Vacation
- (b) STAMP Intruder
- (c) Stack Microbenchmark
- (d) STAMP Vacation, preemption
- (e) STAMP Intruder, preemption
- (f) Stack Microbenchmark, preemption
- (g) Queue Microbenchmark
- (h) List Microbenchmark
- (i) Hash Map Microbenchmark
- (j) Queue Microbenchmark, preemption
- (k) List Microbenchmark, preemption
- (l) Hash Map Microbenchmark, preemption
References


