POSTER: Nonblocking Persistent Software Transactional Memory

H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott
University of Rochester
Rochester, NY, USA
{hbeadle,wcai6,hwen5,scott}@cs.rochester.edu

Abstract

While developed largely for higher density and lower power, byte-addressable nonvolatile memory 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. We present QSTM, a persistent word-based software transactional memory (STM) system to address this problem. Unlike past such systems, QSTM is nonblocking and does not require either the modification of target data structures or the use of a wide CAS instruction.

CCS Concepts • **Computing methodologies** → **Shared memory algorithms**; • **Hardware** → *Non-volatile memory.*

1 Introduction and Overview

The past 16 years have seen the development of dozens of software transactional memory (STM) systems—far too many to cite here. Most have been lock-based or otherwise blocking. Perhaps a dozen have been nonblocking; of these, most but not all have tracked conflicts at object granularity—they have been object-based rather than word-based.

The past 8 years have seen the development of several *persistent* STM systems, whose transactions are not just isolated and consistent, but also atomic and durable in the face of whole-system crashes. Only one of these systems—OneFile [4]—is nonblocking, and it has some serious limitations: it allocates a full word of metadata next to every word of "real" data, complicating data declarations, imposing 2× space overhead, and necessitating the use of a wide CAS instruction.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374502

Our QSTM system is also word-based and persistent, but avoids the metadata-related limitations. It draws partial inspiration from the (transient) RingSTM system [5] but with a REDO log based on the persistent lock-free queue of Friedman et al. [2]. Each entry in QSTM's log represents a transaction, with a unique timestamp, a status (`not_writing`, `writing`, or `complete`), a Bloom filter of the locations written, and a pointer to a detailed write set. Each ongoing transaction maintains both read and write filters, but only the write filter is written into the queue entry. 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.

A thread commits by persisting a queue entry and then durably enqueueing it. Queue entries are kept until their respective writes have been performed and persisted, and then the entry can be freed or reused. Note that if an entry is removed before an ongoing transaction has a chance to validate against it then the transaction must abort. In our implementation ongoing transactions reserve entries by timestamp. To prevent a slow or stopped transaction from reserving too many entries these reservations can safely be ignored when the queue grows too large.

To allow any thread to perform the writes and flushes of any committed transaction, each queue entry contains a pointer to the corresponding write set. Since values can be read directly out of these write sets, other threads do not need to wait for write-back completion to make progress.

If we allowed multiple threads to concurrently perform writeback, 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). We solve this problem by locking a queue entry before performing the writes, and by performing write-back serially in order of commit time. The state field in each queue entry is used to indicate the progress of the write-back and as a lock.

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

If a thread stalls while holding a lock for write-back, the queue will begin to grow longer with each commit but application progress will not be prevented. Additional transactions can continue to commit indefinitely. In practice this could result in unbounded memory usage, but nonetheless provides nonblocking progress (assuming enough memory is available). The memory usage problem can be solved by any mechanism that supports bounded-time completion of the write-back since the problem occurs only if a thread stalls while performing write-back.

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; this pointer is used during validation traversals.

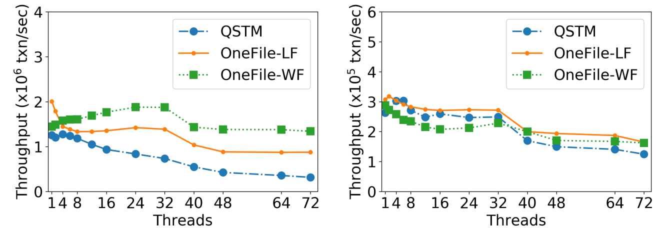
After a crash, it is necessary only to recover the head pointer and any entries that can be reached from it (in addition to any persistent memory used by the application, but this is application-specific). All other memory formerly used by QSTM can be reclaimed. The writes in recovered entries must be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

2 Performance and Usability

We compare the performance of QSTM to that of OneFile [4] using a hash map microbenchmark included with OneFile but with some modifications. In the original version writers used two separate transactions to delete and replace a key while in our version these operations share a single transaction. We also ran a version in which writers replaced ten keys in each transaction to observe the effect of larger transactions. QSTM was configured to use 128-byte Bloom filters and used the Makalu [1] persistent memory allocator. These tests used 50% writer transactions and 50% reader transactions. We tested both the lock-free and wait-free versions of OneFile, shown in the figure as OneFile-LF and OneFile-WF respectively.

We had also hoped to retrofit existing transactional benchmarks to use OneFile, but ran into significant difficulties with the API. All OneFile persistent data structures must be declared using special reserved words to allocate metadata adjacent to every word of "real" data. OneFile transactions then take the form of C++ lambdas, and transactional field accesses rely on operator overloading to update the metadata. These conventions are not particularly onerous for newly designed data structures, but proved difficult to retrofit into code originally developed for transient data.

OneFile usually achieves higher throughput than QSTM, especially when transactions are small. This is because of the smaller number of steps required to commit a OneFile transaction and the larger amount of contention on QSTM's global log. OneFile's advantage, however, shrinks with larger transactions (right subfigure), because QSTM allows any number of transactions to commit while some other transaction is performing write-back, while OneFile serializes this sequence. Significantly, OneFile's use of per-word metadata also imposes nearly 2× space overhead relative to QSTM.



Hash Map throughput: 1 write/txn (L), 10 writes/txn (R)

3 Conclusions and Future Work

Nonblocking progress sets QSTM apart from most previous persistent TM systems. QSTM also consumes much less space than OneFile, and requires neither a wide CAS instruction nor changes to data structure declarations. OneFile has higher throughput, but both systems have a serial bottleneck that limits scalability. We hope in future work to develop a more scalable nonblocking persistent STM. We also hope to mitigate the potential for unbounded memory usage when a QSTM thread blocks while performing writes. One step may be to use a protected library such as Hodor [3] to enforce QSTM as the sole accessor to a memory region and to guarantee timely completion of QSTM write-back operations, despite preemption or process crashes.

References

[1] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *31st Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, Amsterdam, The Netherlands, Nov. 2016.

[2] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *23rd Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Vienna, Austria, Feb. 2018.

[3] 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 *2019 USENIX Annual Technical Conf. (ATC)*, Renton, WA, July 2019.

[4] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, Portland, OR, June 2019.

[5] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable transactions with a single atomic instruction. In *20th Symp. on Parallelism in Algorithms and Architectures (SPAA)*, New York, NY, June 2008.