POSTER: Transactional Composition of Nonblocking Data Structures

Wentao Cai
wcai6@cs.rochester.edu
Computer Science Department
University of Rochester
Rochester, NY, USA

Haosen Wen
hwen5@cs.rochester.com
Computer Science Department
University of Rochester
Rochester, NY, USA

Michael L. Scott
scott@cs.rochester.com
Computer Science Department
University of Rochester
Rochester, NY, USA

Abstract

We introduce *nonblocking transaction composition* (NBTC), a new methodology for atomic composition of nonblocking operations on concurrent data structures. Unlike previous software transactional memory (STM) approaches, NBTC leverages the linearizability of existing nonblocking structures, reducing the number of memory accesses that must be executed together, atomically, to only one per operation in most cases (these are typically the linearizing instructions of the constituent operations).

Our obstruction-free implementation of NBTC, which we call *Medley*, makes it easy to transform most nonblocking data structures into transactional counterparts while preserving their nonblocking liveness and high concurrency. In our experiments, Medley outperforms Lock-Free Transactional Transform (LFTT), the fastest prior competing methodology, by 40–170%. The marginal overhead of Medley's transactional composition, relative to separate operations performed in succession, is roughly 2.2×.

For persistent memory, we observe that failure atomicity for transactions can be achieved "almost for free" with epoch-based *periodic persistence*. Toward that end, we integrate Medley with *nbMontage*, a general system for periodically persistent data structures. The resulting *txMontage* provides ACID transactions and achieves throughput up to two orders of magnitude higher than that of the OneFile persistent STM system.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Theory of computation** → **Parallel computing models**; • **Hardware** → *Non-volatile memory*.

Keywords: nonblocking data structures, transactions, persistent memory

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577503

1 Background

Many multi-threaded systems need to compose operations into transactions that occur in an all-or-nothing fashion (i.e., atomically). One potential solution can be found in *software transactional memory* (STM) systems, which instrument and track every memory access, and convert almost arbitrary sequential code into speculative transactions. Several STM systems provide nonblocking progress [4, 7, 9, 10, 16].

The STM programming model is attractive, but its instrumentation typically imposes 3–10× overhead on transactional operations [14, Sec. 9.2.3]. Spiegelman et al.'s lock-based *transactional data structure libraries* (TDSL) [15] reduce overhead by tailoring STM to specific data—e.g., reducing read set size to only those on *critical nodes* whose updates may indicate semantic conflicts.

In work concurrent to TDSL, Zhang et al. [18] proposed the *Lock-Free Transactional Transform* (LFTT), a nonblocking methodology to statically compose nonblocking operations, based on the observation that only critical nodes matter in conflict management. Subsequently, LaBorde et al.'s *Dynamic Transactional Transform* (DTT) [8] generalized LFTT to dynamic transactions (specified as lambda expressions).

LFTT and DTT leverage the concurrency of existing nonblocking data structures. Unfortunately, the need to identify critical nodes tends to limit them to data structures representing sets and mappings. DTT's publishing and helping mechanisms also require that the "glue" code between operations be fully reentrant (to admit concurrent execution by helping threads [8]) and may result in redundant work when conflicts arise. Worse, for read-heavy workloads, LFTT and DTT require readers to be *visible* to writers, introducing metadata updates that significantly increase contention in the cache coherence protocol.

2 Our Contributions

We introduce *NonBlocking Transaction Composition* (NBTC), a new methodology that can create transactional versions of a wide variety of nonblocking data structures while preserving nonblocking progress and incurring significantly lower overhead than traditional STM. Specifically, NBTC

This work was supported in part by NSF grants CCF-1717712, CNS-1900803, and CNS-1955498. Full version available on arXiv [2].

can compose any nonblocking data structures in which each operation has an *immediately identifiable linearization point*, namely:

1. statically, we can identify every instruction that may potentially serve as the operation's linearization point. Such an instruction must be a load for a read-only operation or a compare-and-swap (CAS) for an update operation;
2. dynamically, after executing a potentially linearizing instruction, we can determine whether it was indeed the linearization point. A linearizing load has to be determined before the operation returns; a linearizing CAS has to be determined without performing any additional shared-memory accesses.

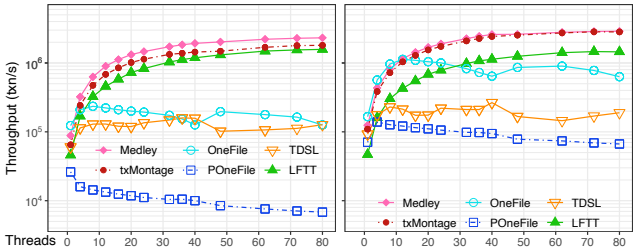
It is widely understood that most nonblocking operations comprise a "planning" phase and a "cleanup" phase [5, 17], separated by a linearizing instruction. Executing the planning phase does not commit the operation to success; cleanup, if needed, can be performed by any thread. The intuition behind NBTC is that in already nonblocking structures, only *critical memory accesses*—for the most part, the linearizing load and compare-and-swap (CAS) instructions—need to occur atomically, while the planning can safely be executed as the transaction encounters it, and the cleanup can be postponed until after the transaction commits.

Our survey of existing data structures and composition patterns reveals two principle complications with this intuition. The first complication involves the notion of a *publication point*, where an operation may become visible to other threads but not yet linearize. Because publication can alter the behavior of other threads, it must remain speculative until the entire transaction commits. An example can be seen in the binary search tree of Natarajan and Mittal [11].

The second complication arises when a transaction, t, performs two or more operations and one of the later operations (call it o_2) observes the speculative CAS of an earlier operation (call it o_1). Here the thread executing t must proceed as if o_1 has completed. If o_1 requires cleanup (something that NBTC will normally delay until after transaction commit), o_2 may need to speculatively help o_1 before o_2 can proceed. Meanwhile, other transactions should not be aware of o_1's existence.

Both complicating cases can be handled by introducing the notion of a *speculation interval* in which CAS instructions must be completed together for an operation to take effect as part of a transaction. This is similar to the *CAS executor* phase in a *normalized* nonblocking data structure [17], but not the same, largely due to the second complication.

With *critical instructions* defined to be the CASes in a speculation interval, plus the linearizing load for a read-only operation, the NBTC methodology is as follows: To atomically execute a set of operations on NBTC-composable data structures, we transform every operation such that (1) instructions prior to the speculation interval and non-critical



(a) get:insert:remove 0:1:1 **(b)** get:insert:remove 18:1:1

Figure 1. Throughput of transactional skiplists (log Y axis).

instructions in the speculation interval are executed on the fly as a transaction encounters them; (2) critical instructions are executed in a speculative fashion, so they will take effect, atomically, only on transaction commit; and (3) instructions after the speculation interval are postponed until after the commit.

To illustrate NBTC, we have written a system, *Medley*, that (1) instruments critical instructions, executes them speculatively, and commits them atomically using *M-compare-N-swap*, our variant of the multi-word CAS of Harris et al. [6]; (2) identifies and eagerly resolves transaction conflicts; and (3) delays non-critical cleanup until transaction commit.

Figure 1 reports throughput on skiplist microbenchmarks performed on a server with in total 80 hyperthreads. The transient systems we measure (in solid lines) include our Medley, the OneFile transient STM [13], TDSL [15], and LFTT [18]. Medley outperforms OneFile and TDSL by an order of magnitude, and LFTT by 40–170%. Given our invisible readers, the gap between Medley and LFTT is larger when the workload has a higher percentage of writes.

For persistent memory, we observe that failure atomicity for transactions comes for free with epoch-based *periodic persistence* [12]: if operations of the same transaction always occur in the same epoch, then they will be recovered (or lost) together in the wake of a crash. Building on this observation, we merge Medley with *nbMontage* [1], our epoch-based system for nonblocking periodic persistence, to create *txMontage*. All operations in a given transaction are labeled with the same epoch number, which is then validated along with the rest of the read set at commit time, ensuring that the transaction commits in this epoch.

The persistent systems in Figure 1 (dotted lines) represent our txMontage and the OneFile persistent STM (POneFile) [13]. While txMontage on Intel Optane non-volatile memory performs closely to Medley on DRAM, the persistent OneFile is roughly 10× slower than its transient version—in turn two orders of magnitude slower than txMontage.

We have transformed a variety of data structures using Medley and txMontage, and conducted experiments running the TPC-C [3] transaction processing benchmark on Medley, txMontage, and other compatible competitors. The results (reported in the full version on arXiv [2]) reconfirm the exceptional performance of our systems.

References

[1] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. 2021. Fast Nonblocking Persistence for Concurrent Data Structures. In *35th Intl. Symp. on Distributed Computing (DISC)*. Freiburg, Germany, 14:1–14:20.

[2] Wentao Cai, Haosen Wen, and Michael L. Scott. 2023. Transactional Composition of Nonblocking Data Structures. arXiv preprint arXiv:2301.00996.

[3] The Transaction Processing Council. 2010. TPC-C Benchmark (Revision 5.11.0). http://www.tpc.org/tpcc/.

[4] Keir Fraser. 2003. *Practical Lock-Freedom.* Ph. D. Dissertation. King's College, Univ. of Cambridge. Published as Univ. of Cambridge Computer Laboratory technical report #579, February 2004. www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[5] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *41st ACM Conf. on Programming Language Design and Implementation (PLDI)*. virtual conference, 377–392.

[6] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *16th Intl. Symp. on Distributed Computing (DISC)*. Toulouse, France, 265–279.

[7] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *22nd ACM Symp. on Principles of Distributed Computing (PODC)*. Boston, MA, 92–101.

[8] Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. 2019. Wait-Free Dynamic Transactions for Linked Data Structures. In *10th Intl. Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. Washington, DC, 41–50.

[9] Virendra Jayant Marathe and Mark Moir. 2008. Toward High Performance Nonblocking Software Transactional Memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, UT, 227–236.

[10] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the Overhead of Software Transactional Memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. Ottawa, ON, Canada, 11 pages.

[11] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Orlando, FL, 317–328.

[12] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st Intl. Symp. on Distributed Computing (DISC)*. Vienna, Austria, 37:1–37:16.

[13] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*. Portland, OR, 151–163.

[14] Michael L. Scott. 2013. *Shared-Memory Synchronization.* Morgan & Claypool Publishers, San Rafael, CA.

[15] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *37th ACM Conf. on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, 682–696.

[16] Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay, and Cong Wang. 2009. NZTM: Nonblocking Zero-indirection Transactional Memory. In *21st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Calgary, AB, Canada, 204–213.

[17] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Orlando, FL, 357–368.

[18] Deli Zhang and Damian Dechev. 2016. Lock-Free Transactions without Rollbacks for Linked Data Structures. In *28th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Pacific Grove, CA, 325–336.