Efficient Nonblocking Software Transactional Memory

Virendra J. Marathe
University of Rochester
vmarathe@cs.rochester.edu

Mark Moir
Sun Microsystems Laboratories
mark.moir@sun.com

Abstract

Foundational transactional memory research grew out of research into *nonblocking* concurrent data structures, which aim to overcome the many well-known software engineering, performance, and robustness problems associated with lock-based implementations. Recently, many researchers have developed *blocking* STMs, recognising that they are much easier to design and that the software engineering benefits of STM can be delivered even by a blocking STM. But hiding blocking from the application programmer does not eliminate all of its disadvantages, and in some cases blocking is *unacceptable*, for example if STM is to be used to coordinate between an interrupt handler and the interrupted thread.

Recently, a common belief has emerged that blocking STMs are fundamentally faster than nonblocking ones largely based on Ennals's argument [2] that nonblocking STMs cannot store data in-place as most blocking STMs do. However, this argument is based only on intuition, not a formal proof. It misses the possibility of nonblocking STM designs that closely mimic blocking STMs in the common case, resorting to techniques such as displacing transactional data only when needed to avoid waiting for a thread that is delayed while modifying it.

We present a novel nonblocking word-based STM based on this approach. Our STM eliminates several significant sources of common-case overhead in the previous best nonblocking word-based STM, and also performs comparably with the simple blocking STM on which it is based.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent programming

General Terms Algorithms, Design

Keywords Nonblocking, software transactional memory

1. Design Overview

We have implemented a simple, blocking STM based in part on the one used in the Hybrid Transactional Memory prototype described in [1], and a nonblocking variant of it. Our blocking STM employs some novel optimisations which we retain in our nonblocking STM. In both STMs, a transaction maintains read and write sets, eagerly acquires exclusive ownership of locations to be written, and copies values from the write set back to memory upon successful commit, before releasing ownerships. They use invisible reads, and conservatively validate the read set after every read to ensure that user code never runs on inconsistent data.

Our STM is intended for use in languages such as C and C++. As such, it is not acceptable to do only "occasional" validation, as suggested as an optimisation by some authors, because this can result in arbitrary behaviour. It is also not acceptable to dictate data layout in this context, so we do not co-locate transactional metadata with the data. We emphasise that our desire to build a nonblocking STM does *not* require this choice, as suggested in [2].

Transactions may abort conflicting transactions (by `CAS`ing the conflicting transaction's status to `Aborted`). However, if a transaction has committed but has not released its ownerships, in the blocking STM other transactions must wait for the owning transaction to release its ownership before accessing locations it owns. In our nonblocking STM, a conflicting transaction is permitted to *steal* ownership of the location and continue execution. As described below, managing this stealing correctly accounts for most of the complexity in our nonblocking STM.

The primary data structures in our STMs are *transaction descriptors*, which are used to represent transactions, and a table of *ownership records* (orecs), which are used to represent ownership by transactions of memory locations. A many-one hashing function maps memory locations into the orec table.

A transaction descriptor consists of a transaction ID (`tid`), a version number (`version`), a status indicator, and read and write sets. The `tid` and `version` together uniquely identify a transaction. Transaction descriptors are reused by threads, eliminating the overhead of allocating them and managing their reclamation. The read and write sets are organised as per-orec *rows*. Each row contains an orec identifier, a snapshot of the orec, and an array of *entries*, each of which is an address-value pair for some address covered by the indicated orec.

Each orec is stored in one 64-bit word, and is atomically modified using a `CAS`. Orecs consist of: `tid` and `version` fields, used to identify the current owner transaction; a `mode` field indicating that the orec is `UNOWNED` or `OWNED`; and a `row` field, to identify the write set row in which the owning transaction stores entries for locations mapping to that orec. Storing the owner's `tid` and `version` in the orec enables a novel *fast release* optimisation whereby the transaction simply increments its `version` to release its ownerships. This eliminates the overhead of explicitly releasing the acquired orecs. These data structures have some additional fields in the nonblocking STM, as described below.

1.1 The Stealing Process

Before stealing an orec from a committed transaction that is copying back values to locations covered by the orec, the stealer copies entries from the victim's associated transaction row into its write set. This preserves the logical values written by the victim when the stealing occurs, regardless of whether the victim has completed its copyback or not. To record that the logical values of these loca-

PPoPP'07 March 14–17, 2007, San Jose, California, USA.
ACM 978-1-59593-602-8/07/0003.

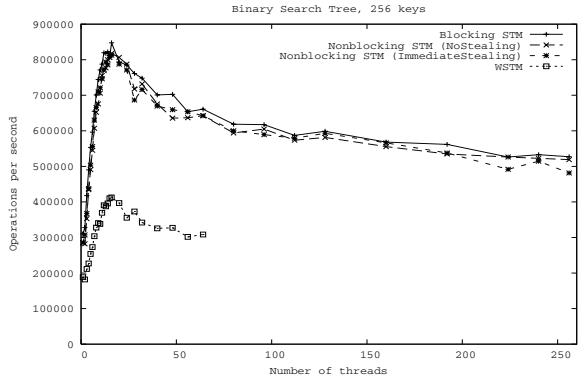

Figure 1. Performance on Binary Search Tree (`bst`)

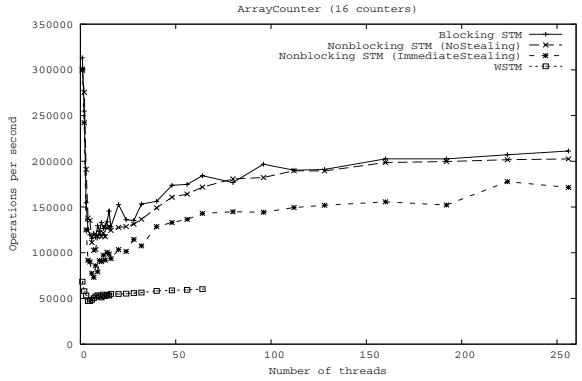

Figure 2. Performance on an Array of 16 Counters Benchmark

tions are now stored in the stealer's write set row, the stealer sets a `stolen_orec` flag in the orec.

Because it is more expensive to access locations covered by a stolen orec, it is desirable to reset a stolen orec to its normal, unstolen state when the victim has finished its copyback. To facilitate this transition, a new per-orec `copier_exists` flag records whether there is a transaction copying values back to locations covered by the orec; this flag is set true when the orec is stolen to reflect that the victim is doing so.

After its copyback, the victim resets the `copier_exists` flag for each orec stolen from it, thus indicating that it has finished its copyback. If a stealer commits *before* the victim finishes its copyback, logical values of stolen locations (in the stealer's write set) may differ from their physical values even after the victim completes. Thus, the stealer's write set row must be preserved until the entries in it are copied back to the physical locations or into another stealer's writeset. In the full paper, we explain how transaction descriptors can be reused despite this.

When stealing an already-stolen orec, if the stealer finds that the `copier_exists` flag has been reset, it can set the flag true again, assuming the role of copier itself. It can then copy the displaced values back to the physical locations (overwriting any "late writes" performed by a previous victim), and reset the orec to the normal, unstolen mode. Otherwise, the stealer refrains from copying back the values, leaving this task for a subsequent stealer. Thus, we maintain the invariant that only one transaction at a time copies values back to locations covered by a given orec, thus eliminating the redundant copying performed by WSTM [3].

The fast release optimisation complicates stealing somewhat because the stealer must explicitly inform the victim transaction that it should reset the `copier_exists` flag. We enable this by embedding a special `stealer_note` list in each transaction's `version` field. A stealer informs a victim of the theft by `CAS`ing a new note (comprising information about the stolen orec) into the victim's `stealer_note` list. The victim detects the theft when its `CAS` to increment its version fails. If the victim does a fast release when a stealer is stealing the victim's orec, the stealer's `stealer_note` `CAS` fails. In that case, the stealer can switch the stolen orec back to unstolen mode because the victim has completed its copyback.

2. Performance Evaluation

We compared the performance the previous best nonblocking word-based STM (WSTM) [3] to our blocking STM and our nonblocking STM with two simple stealing policies: The `NoStealing` policy evaluates the common-case cost of being nonblocking, despite never invoking the option to steal. The `ImmediateStealing` policy always immediately steals when a conflict with a committed transaction is encountered. Here we present results from experiments on a 144-processor Sun Fire E15K cache coherent multiprocessor with 1.5GHz UltraSPARC® IV+ processors (72 dual-core chips). Threading levels range from 1 to 256. (WSTM is limited to 64 threads; to allow more complete comparison, we plan to investigate in the future whether this limitation can be easily eliminated.)

We evaluated these systems on several microbenchmarks; here we report results for the binary search tree (`bst`) and array counter (`arraycntr`) benchmarks. All other benchmarks supported similar conclusions. The `bst` benchmark comprises 80% lookups, 10% inserts and 10% deletes, with key values ranging from 0 to 255. Figure 1 shows scalability results. Clearly, our novel design and optimisations significantly improve over the state-of-the-art nonblocking word based STM. Our nonblocking STM performs comparably with the blocking STM, with the stealing policy making little difference in this read-dominated, low-conflict benchmark.

In the `arraycntr` benchmark, each transaction increments each of 16 counters. This is a write-dominated workload characterised by frequent conflicts (and therefore more stealing). Figure 2 shows results for this benchmark. Again, our nonblocking STM significantly outperforms WSTM. However, with increasing concurrency the opportunity for stealing also increases. We see a distinct performance gap between our blocking and nonblocking STMs. This shows the overhead of stealing in our system, and points to the need for sensible stealing policies that allow a victim a chance to finish before deciding to steal. This performance gap does not imply a fundamental gap between blocking and nonblocking STMs. That question can only be answered by formal proofs. On the other hand, our results unambiguously support our claim that there is ample room for improvement over the best previous nonblocking STMs.

Future work includes adaptive stealing policies, and applying our ideas to more recent (faster) blocking STMs.

References

[1] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[2] R. Ennals. Software transactional memory should not be obstruction-free, 2005. http://www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

[3] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, 2003.