Brief Announcement: On Implementing Software Transactional Memory in the C++ Memory Model

Matthew Rodriguez
Lehigh University
mar316@lehigh.edu

Michael Spear
Lehigh University
spear@lehigh.edu

ABSTRACT

High-performance software transactional memory (STM) implementations rely on nuanced use of synchronization variables to coordinate speculative accesses to program data. We discuss some consequences of the C++ memory model on STM, identify an easy-to-fix implementation error, and describe an unavoidable formal race condition that occurs in an important class of STM algorithms.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent algorithms**.

KEYWORDS

Synchronization, Speculation, Transactional Memory, C++

ACM Reference Format:
Matthew Rodriguez and Michael Spear. 2020. Brief Announcement: On Implementing Software Transactional Memory in the C++ Memory Model. In *Symposium on Principles of Distributed Computing (PODC '20), August 3–7, 2020, Virtual Event, Italy.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3382734.3405746

1 INTRODUCTION

Transactional Memory (TM) [12] is a concurrency control mechanism in which programmers are responsible for determining *which* regions of code need to run atomically with respect to each other, but not *how* to achieve that atomicity. Instead, special hardware (HTM) or a software run-time system (STM) monitors the execution of a program's transactions, ensuring that they have the same effect as if they were executed serially, while running them concurrently whenever possible.

C++ has robust concurrency support, and strictly defines data races [1]. However, as discussed by Boehm [2], speculative synchronization mechanisms are difficult to implement correctly in C++: Even if a synchronization mechanism identifies that a read was racy and avoids using its result, the mere fact that the data race occurred means the program behavior is undefined. Furthermore, the C++ memory model provides weaker ordering than many programs expect, especially for reads of synchronization data.

Boehm speculates that TM implementations in C++ are particularly vulnerable to implementation errors. We affirm this hypothesis. First, we apply Boehm's reasoning to memory accesses within transactions. We discuss STM-specific implementation challenges, and measure the overhead of complying with the C++ memory model. Second, we consider the "privatization" problem [14], wherein a transaction transitions a datum from a state in which it can be accessed by transactions, to a state in which it is only accessed non-transactionally. We show that one of the most scalable strategies for privatization is incompatible with the C++ memory model.

PODC '20, August 3–7, 2020, Virtual Event, Italy

ACM ISBN 978-1-4503-7582-5/20/08…$15.00
https://doi.org/10.1145/3382734.3405746

2 PREVENTING RACES ON PROGRAM DATA

To avoid races, C++ STM implementations synchronize threads via atomic variables (`std::atomic<>`) [5–9, 11, 17]. This is true even for most blocking STMs, which use atomic variables to implement custom lock objects (i.e., "ownership records", or "orecs"). In these STMs, transactions execute *optimistically*: they do not ensure mutual exclusion before performing a write to a location. Instead, they acquire "ownership", and thus concurrent reads to that location must "validate" to ensure they do not overlap in time with writes. In the simplest case, where the STM uses a single lock to protect all memory, this style of synchronization is indistinguishable from a sequence lock.

Columns 1 and 2 of Figure 1 present a simplified version of how orec-based STM reads and writes memory. We consider both "undo logging" (W01-W08) and "redo logging" (W09-W10) for writes. The surprising instructions are R05 and W07, which cast accesses to program data into atomic variable accesses. The C++ memory model requires these to be atomic: R05 and W07 are to program data, but a writer could perform W01-W06 after a reader's execution of R01-R04. At that point, simultaneous execution of W07 and R05 would constitute a race. It does not matter that the reader will detect the race and not *use* the result of R05: if the two accesses are not via atomic variables, program behavior is undefined. Thus to obey the memory model, all accesses to program data by transactions must be via atomic variables. This also includes "undoing" in the case of an abort after line W08, or "redoing" at commit time, in the case of implementations that uses W09-W10.

When accessing program data, a natural desire is to use a *relaxed* atomic, to avoid unnecessary fence instructions and allow the compiler maximum opportunity to reorder instructions. However, the access at R05 cannot be relaxed. In C++, the strongest ordering guarantee that a read can provide is "acquire" semantics. Roughly, acquire semantics ensure that accesses *after* a read do not get reordered to *before* the read. The fence at R04 ensures that R05 does not happen before R04. The fence at R06 ensures that R07-R12 do not happen before R06. Notice that neither fence prevents R05 from delaying until after R06. Making R05 a non-relaxed access (or inserting an explicit acquire fence) is necessary to ensure a strict order R04-R05-R06.

```
R01 TM_Read(addr)
R02   lock_idx = hash(addr) % lock_table.size
R03   lock_ref = &lock_table[lock_idx]
R04   pre_check = lock_ref->read(acquire_fence)
R05   val = *(atomic<>)addr;
R06   post_check = lock_ref->read(acquire_fence)
R07   // ensure consistent read
R08   if (invalid(pre_check) ||
R09       (post_check != pre_check))
R10     return TM_Read(addr)
R11   ... // logging, validation
R12   return val
```

```
W01 TM_Write(addr, value) // undo version
W02   lock_idx = hash(addr) % lock_table.size
W03   lock_ref = &lock_table[lock_idx]
W04   if (!lock_ref->acquire(my_id))
W05       abort()
W06   undos.log(addr, *addr)
W07   *(atomic<>)addr = val
W08   ... // logging, validation

W09 TM_Write(addr, value) // redo version
W10   writeset.insert(<addr, value>)
```

```
P01 transaction {
P02   if (!TM_Read(&flag))
P03     TM_Write(&flag, true)
P04     TM_Write(&found, true)
P05 }
P06 if (found)
P07   sharedData++
```

```
T01  transaction {
T02    if (!TM_Read(&flag))
T03      tmp = TM_Read(&sharedData)
T04      tmp *= 2
T05      TM_Write(&sharedData, tmp)
T06  }
```

Figure 1: Simplified transactional read and write pseudocode (columns 1/2). An example of privatization (columns 3/4).

In C++17, casting an address to `std::atomic<>` is not standards-compliant [3]. The C++20 `atomic_ref<>` proposal [18] will provide a standards-compliant solution. Until then, the best an STM implementation can do is to (1) ensure variables do not span cache lines, (2) limit `TM_Read` and `TM_Write` to primitive types that can be read in a single atomic hardware instruction, and (3) verify that the compiler produces the same assembly code as would be produced via `atomic_ref<>`. Additionally, some STM implementations use `memcpy` for redo and undo operations. These instructions can race with R05, and thus until C++ adds support for `atomic_memcpy`, STM implementations should not use `memcpy` on program data.

3 PRIVATIZATION

The above discussion focuses on the challenge of making sure that concurrent accesses to a location, by two transactions, do not cause undefined behavior. In this section we consider another challenge. Columns 3 and 4 of Figure 1 depict a behavior known as "privatization". Thread T_P, executing transaction P (lines P01-P05) transitions a datum from a state in which it can be accessed by many transactions, to a state in which it is only accessed by T_P (on lines P06-P07). Another thread T_T may be executing transaction T (lines T01-T06) simultaneously. Past work observed that speculation can lead to races between T and lines P06-P07. In particular: (1) If T is using redo logging, has already reached T06, but has not yet written back its update to `sharedData`, P07 can read stale data. (2) If T is using undo logging and finished T05 before P reached P06, then T must abort. Until it cleans up, P07 could read a value written by failed T. Our contribution is the observation that there is a third problem: (3) If T is about to execute the read on line T03, then even if T subsequently aborts, that read could race with P07. Note that while our example is somewhat contrived, privatization bugs are significant. If `sharedData` was on the heap, and P07 was a `free` instruction, then the access at T03 could cause a segmentation fault if P07 returned memory pages to the operating system.

Clearly, it is inappropriate to transform P07 to use an atomic variable: the data is logically private to the thread, and it is unrealistic to require a programmer to transform arbitrary data (potentially all data in the program) to be atomic. Furthermore, simply making the access atomic would not remedy the first two problems.

There are three approaches that address the first two problems with privatization. The first, *quiescence*, requires any committing transaction to wait for all concurrent transactions to commit or abort *and clean up* before its thread can execute nontransactionally. That is, at line P05, T_P would wait until T completed. Only then could P06 execute. A slightly better approach, which does not work for undo-based STM, provides a third condition upon which T_P may resume execution: once T starts validating, T_P knows that if T conflicted with P, then T will abort. If T does not abort, then T does not conflict with P, and thus it cannot access `sharedData`.

The most ambitious approach, which also only works for redo-based STM, does not require waiting [5, 17]. Suppose that some mechanism was in place to serialize all writing transactions' commits. Then one transaction could not commit until the previous had released all its locks, and every transaction could validate whenever it detected that some new transaction had committed. In a redo-based STM, the first of our problems above could not happen. To avoid the third problem, thread P could count on the fact that while thread T might *observe* the nontransactional change to `sharedData` within its `TM_Read` operation (e.g., if T03 was concurrent with P07), it would not *use* that value, because of the validation that would happen on account of transaction P committing. As appealing as the third approach appears, it is incorrect in C++: a write on line P07 to a non-atomic variable could be concurrent with a read on line R05. Even when R05 casts to atomic, P07 does not.

Furthermore, we present one case in which the second approach is incorrect. Many STM and Hybrid HTM/STM algorithms [4, 5, 13] use "value-based validation", where transactions validate by checking *program data*, not by checking orecs or other metadata. In these systems, P cannot wait for concurrent transactions to *start* validating; they must must *complete* a validation; otherwise, validation reads could race with P07.

4 EVALUATION

We briefly assess the cost of adhering to the C++ memory model, using a small suite of STM algorithms and the STAMP [15] benchmark suite. Figure 2 presents the results of our experiments. We compare an undo STM with orecs (Eager), a redo STM with orecs (Lazy), and a non-orec STM (NOrec). We consider incorrect implementations that do not cast to atomic (Incorrect), and correct implementations (Fixed). NOrec_Incorrect and NOrec_NoPriv use a privatization strategy that is not compatible with the C++ memory model. All "Fixed" algorithms use quiescence. All experiments were performed on a Xeon 8160 CPU with 24 cores/48 threads. Code was compiled with Clang/LLVM 10.0, using -O3 optimizations and the TM plugin for LLVM [19]. Results are the average of three trials.

The experiments show that atomic variables themselves have negligible overhead. Eager_Fixed and Lazy_Fixed are almost indistinguishable from their Incorrect counterparts. While a positive result, we caution that the Xeon 8160 is a TSO processor, and thus read-read ordering does not require a memory fence instruction. On

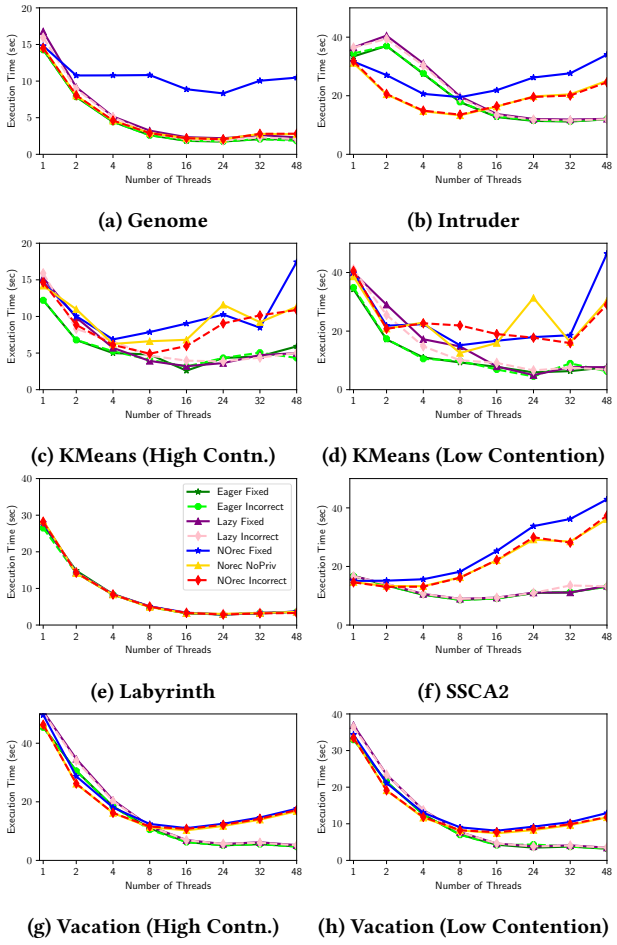

Figure 2: Experimental results on STAMP benchmark suite.

a CPU with relaxed memory consistency, such as ARM or POWER, R05 and W07 would incur fence instruction overheads. Further experimentation is left as future work.

However, converting NOrec to use a correct privatization strategy had a measurable impact on scalability. The result is most pronounced for Genome, but can also be seen in Intruder and KMeans (though KMeans had less consistent behavior in general). For classic STM, this is not a particularly significant outcome, since NOrec was rarely the best algorithm for these workloads. However, NOrec is the most promising foundation for HTM-accelerated "Hybrid" TM, and further experimentation is needed to determine if the impact on scalability carries forward to hybrid TM.

5 CONCLUSIONS

Our work reveals nuanced relationships between STM implementations and the C++ memory model. We showed that in addition to long-standing but oft-overlooked guidance about sequence locks, the need for atomic casting limits the ability to use `memcpy` or to access large primitive types nonatomically. To some degree, C++20's `atomic_ref` will reduce this burden. We also showed that scalable support for the privatization idiom is incompatible with the C++ memory model, obviating a key benefit of some STM algorithms.

A few STM algorithms are immune to some of the concerns we discussed: TLRW [7] and InvalSTM [10] use pessimistic locking, and thus reads are never concurrent with writes; cohorts [16] uses phased commits to prevent redo concurrent with reads. These approaches are also privatization-safe. As future work, we believe that it will be worthwhile to re-investigate STM design decisions and implementations following the finalization of C++20.

ACKNOWLEDGMENTS

This work was supported in part by the NSF under Grant CNS-1814974. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 28th POPL*. Austin, TX.
[2] Hans-J. Boehm. 2012. Can Seqlocks Get Along with Programming Language Memory Models?. In *Proceedings of the 2012 MSPC*. Beijing, China, 12–20.
[3] Hans-J. Boehm. 2014. N4013: Atomic operations on non-atomic data. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4013.html
[4] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael Spear. 2011. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th ASPLOS*. Newport Beach, CA.
[5] Luke Dalessandro, Michael Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th PPoPP*. Bangalore, India.
[6] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th DISC*. Stockholm, Sweden.
[7] David Dice and Nir Shavit. 2010. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. Santorini, Greece.
[8] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th PPoPP*. Salt Lake City, UT.
[9] Keir Fraser. 2003. *Practical Lock-Freedom*. Ph.D. Dissertation. King's College, University of Cambridge.
[10] Justin Gottschlich, Manish Vachharajani, and Jeremy Siek. 2010. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proceedings of the 8th CGO*. Toronto, ON, Canada.
[11] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd PODC*. Boston, MA.
[12] Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*. San Diego, CA.
[13] Alexander Matveev and Nir Shavit. 2015. Reduced Hardware NORec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the 19th ASPLOS*. Istanbul, Turkey.
[14] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. 2008. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th SPAA*. Munich, Germany.
[15] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the 2008 IISWC*. Seattle, WA.
[16] Wenjia Ruan, Yujie Liu, Chao Wang, and Michael Spear. 2013. On the Platform Specificity of STM Instrumentation Mechanisms. In *Proceedings of the 11th CGO*. Shenzhen, China.
[17] Michael Spear, Maged M. Michael, and Christoph von Praun. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th SPAA*. Munich, Germany.
[18] Daniel Sunderland, H. Carter Edwards, Hans Boehm, Olivier Giroux, Mark Hoemmen, David Hollman, Bryce Adelstein Lelbach, and Jens Maurer. 2018. p0019R8: Atomic Ref. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0019r8.html
[19] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael Scott, and Michael Spear. 2019. Simplifying Transactional Memory Support in C++. *ACM TACO* 16, 3 (July 2019), 25:1–25:24.