Transactional Read-Modify-Write Without Aborts

WENJIA RUAN, YUJIE LIU, and MICHAEL SPEAR, Lehigh University

Language-level transactions are said to provide “atomicity,” implying that the order of operations within a transaction should be invisible to concurrent transactions and thus that independent operations within a transaction should be safe to execute in any order. In this article, we present a mechanism for dynamically reordering memory operations within a transaction so that read-modify-write operations on highly contended locations can be delayed until the very end of the transaction. When integrated with traditional transactional conflict detection mechanisms, our approach reduces aborts on hot memory locations, such as statistics counters, thereby improving throughput and reducing wasted work. We present three algorithms for delaying highly contended read-modify-write operations within transactions, and we evaluate their impact on throughput for eager and lazy transactional systems across multiple workloads. We also discuss complications that arise from the interaction between our mechanism and the need for strong language-level semantics, and we propose algorithmic extensions that prevent errors from occurring when accesses are aggressively reordered in a transactional memory implementation with weak semantics.

Categories and Subject Descriptors: I [Computing Methodologies]; B.3.2 [Shared Memory]: Algorithms; C.1.4 [Computer Systems Organization]: *Multicore architectures*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Transactional Memory, Conflict Detection, Language-Level Semantics

ACM Reference Format:

Wenjia Ruan, Yujie Liu, and Michael Spear. 2015. Transactional read-modify-write without aborts. *ACM Trans. Architect. Code Optim.* 11, 4, Article 63 (January 2015), 24 pages.

DOI: <http://dx.doi.org/10.1145/2688904>

1. INTRODUCTION

Language-level transactions are often described as providing atomicity in the sense that the operations that comprise the transaction appear to happen “all at once,” as an indivisible operation without any intervening memory operations from concurrent transactions. Although Hardware Transactional Memory (HTM) [Herlihy and Moss 1993] is largely able to provide this illusion for small transactions, Software Transactional Memory (STM) typically cannot due to the high overheads that arise [Menon et al. 2008; Guerraoui et al. 2010]. Instead, STM implementations fall back to semantics based on multiple logical locks, in which transactions appear to acquire locks covering their read and/or write sets dynamically during the course of their execution.

This work is presented as a new article, not an extension of a conference paper. A preliminary version of this work appeared at the WTTM 2013 workshop, which does not have archival proceedings. This work was supported in part by the National Science Foundation under grants CNS-1016828, CCF-1218530, and CAREER-1253362.

Authors' address: W. Ruan, Y. Liu, and M. Spear, 27 Memorial Drive West, Bethlehem PA, 18015; emails: {wer210, yul510, spear}@cse.lehigh.edu.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2015 ACM 1544-3566/2015/01-ART63 \$15.00

DOI: <http://dx.doi.org/10.1145/2688904>

```

1  transaction {
2      expensive_function_1(x);
3      stats++;
4      expensive_function_2(y);
5  }

```

Fig. 1. A transaction that modifies a highly contended variable. If line 3 could be delayed until commit time, concurrent invocations of this transaction would not abort due to accesses to stats.

The unfortunate consequence is that compilers must be conservative when reordering memory accesses that occur within a transaction. In particular, under the most popular Encounter-time Lock Atomicity (ELA) semantics proposed by Menon et al. [2008], a transaction appears to acquire a lock for each location it reads at the time each location is first read. In practical implementations of STM [Dice et al. 2006; Felber et al. 2008], writes are treated as implicit reads and also appear to acquire locks at the time the location is first accessed, even though ELA permits writes to appear to delay lock acquisition until as late as the commit point of the transaction.

Although reordering accesses is typically regarded as a low-level compiler issue with the ideal order based on register pressure and predicted cache misses, Transactional Memory (TM) presents a wrinkle: Virtually every TM implementation detects conflicts during transaction execution. In HTM, this is usually achieved by monitoring cache invalidations; in STM, this is achieved by determining if a transaction’s logical lock acquisition overlaps with prior lock acquisitions by concurrent, not-yet-committed transactions. For highly contended (“hot”) locations, the real-time ordering of lock acquisitions can create unnecessary conflicts that result in aborts and wasted work.

To illustrate this point, consider the code in Figure 1. In every invocation, considerable work is done by the first function, after which a statistics counter is incremented. The increment consists of a shared memory read, local computation, and a shared memory write. Subsequently, another expensive computation is performed. During the second computation, the transaction is vulnerable to aborts on account of concurrent accesses to the counter.

Suppose two transactions T_1 and T_2 execute this code in lock-step but with different values of x and y . It is possible for their only conflict to be on accesses to stats. If the underlying TM is eager (e.g., best-effort HTM [Chaudhry et al. 2009; Intel Corp. 2012; Chung et al. 2010]) or STM that acquires locks upon first write access ([Felber et al. 2008; Dice and Shavit 2010; Wang et al. 2007]), one transaction will lock stats, the other will attempt to read stats, and the resulting conflict will cause at least one transaction to be aborted. If the underlying TM is lazy [Minh et al. 2007; Dice et al. 2006; Dalessandro et al. 2010], then both transactions will continue to their commit point, at which time the conflicting accesses to stats will be detected and at least one transaction will abort. In both cases, a significant amount of computation will be devoted to a transaction that does not complete, resulting in wasted work.

If it were known that stats was never accessed by either function, the increment could be moved to the end of the transaction, thus avoiding conflicts in many cases [Martin et al. 2013]. More aggressively, since the result of the increment is also never used within the transaction, but the increment still must be performed atomically with the rest of the transaction, it is even possible to delay the increment until some point *within the commit operation*. Doing so would eliminate all possibility of aborts due to stats accesses.

The aborts that result from shared counters arise in real code, as demonstrated by efforts to transactionalize memcached [Pohlack and Diestelhorst 2011; Ruan et al. 2014b]. Worse yet, in memcached the relevant counters are, themselves, incremented in nested function calls that occur only on some branches. When the prospect of nested

transactions is also considered, we must conclude that manually refactoring code to place hot increment operations at the end of transactions is not possible.

We propose an algorithm for dynamically reordering increment and other simple Read-Modify-Write (RMW) operations within a transaction. Our mechanism employs runtime tracking to ensure correctness even when the target of a deferred operation is read or written by other instructions within the transaction. We present the algorithm and tracking mechanisms in Section 2, and then, in Section 3, we present candidate implementations that use annotations or live-out analysis to instrument operations so they can be delayed. Section 4 discusses the surprising consequences that our algorithm can have on the publication idiom [Menon et al. 2008] and proposes solutions. We evaluate our algorithms in Section 5, discuss related work in Section 6, and then conclude in Section 7.

2. AN ALGORITHM FOR DELAYING RMWS

In this section, we briefly review STM implementation details. We then present an algorithm for safely delaying transactional RMWs. For now, our focus is on correctness in the absence of concerns about language-level semantics. Likewise, we do not yet discuss how to identify candidate RMW operations.

2.1. STM Background

STM implementations typically expose an API with four operations: `TxBEGIN` creates a checkpoint and initializes per-thread metadata to support tracking conflicts on reads and ensure serializability of writes; `TxREAD <T>` reads a location of type `T` and ensures the read is consistent with all prior reads and writes performed by the transaction (this property, called *opacity* [Guerraoui and Kapalka 2008], provides a basis for asserting the correctness of STM implementations); `TxWRITE <T>` updates the value of a location such that subsequent reads within the same transaction will see the update, but other transactions will not (yet) see the update. `TxCOMMIT` makes writes visible to other transactions only if doing so will produce a result indistinguishable from an execution history in which one transaction runs at a time. Typically, `TxCOMMIT` ensures that all to-be-written locations are locked by the transaction, verifies that no concurrent transaction updated any location read by the transaction, then finalizes writes and releases locks.

For simplicity of presentation, we assume throughout this section that transactions operate only on locations of a single type. We also assume that the language is type safe, such that for any two operations on locations L_1 and L_2 , both of size S , $L_1 = L_2$, or $|L_1 - L_2| \geq S$. Note that these assumptions do not apply to the C and C++ languages, which require a more complicated implementation described in Section 5.

The basic framework for an STM algorithm is presented in Table I and Algorithms 1–2. If we assume that `TxRMW` is never called, then the `rmws` set will be empty, and the resulting algorithm resembles TL2 [Dice et al. 2006] or the commit-time-locking variant of TinySTM [Felber et al. 2008]. Note that this algorithm delays all writes until commit time, buffering them in a per-thread log during transaction execution. The correctness of the algorithm hinges on three properties. First, on Algorithm 1 line 6, every read first checks if there was a previous write to the same location by the current transaction. This check ensures processor consistency: An in-flight transaction observes every modification that it intends to make to main memory. Second, the `TxCOMMIT` function ensures that transactions appear to commit atomically. This property is provided via the use of versioned locks (ownership records, or orecs) and a global timestamp. Briefly, locks can only be acquired if their version number is no greater than the time at which the transaction was last known to be valid, and all reads are verified after all locks are acquired to ensure serializability. Third, transactions

Table I. STM-Related Variables

global variables		
<i>timestamp</i>	Integer	timestamp
<i>orecs</i>	OwnershipRecord[]	orec table, the MSB of each orec is set to 1 if orec is acquired
per-transaction variables		
<i>my.lock</i>	(Integer, Integer)	(1, <i>thread.id</i>)
<i>start</i>	Integer	start time
<i>end</i>	Integer	end time
<i>writes</i>	WriteSet	pending writes
<i>reads</i>	ReadSet	locations read
<i>locks</i>	LockSet	locks held
<i>rmws</i>	RMWSet	pending RMWs
<i>rdflag</i>	Boolean	see Algorithm 3
<i>rdaddrs</i>	AddressSet	see Algorithm 4

ALGORITHM 1: A lazy STM algorithm with support for delayed RMWs. Underlined code represents additions relative to a traditional lazy STM algorithm.

```

1 TxBEGIN()
2   start  $\leftarrow$  timestamp
3   reads, writes, locks, rmws  $\leftarrow$   $\emptyset$  // clear read-modify-write set
4 TxREAD(addr)
5   if addr  $\in$  rmws then return PROMOTE(addr) // if there's a previous RMW on addr,
    promote it
6   if addr  $\in$  writes then return writes[addr]
7   while true do
8     o1  $\leftarrow$  orecs[addr]
9     v  $\leftarrow$  *addr
10    o2  $\leftarrow$  orecs[addr]
11    if o1 = o2  $\wedge$  o2  $\leq$  start then
12      reads  $\leftarrow$  reads  $\cup$  {(addr, o1)}
13      return v
14    else if  $\neg$ o2.LOCKED() then EXTEND()
15    else ABORT()
16 TxWRITE(addr, v)
17   if addr  $\in$  rmws then
18     rmws  $\leftarrow$  rmws  $\setminus$  {(addr, -)} // forget this RMW, since it will be overwritten
19     TxRead(addr) // but perform a read to preserve consistency
20   writes  $\leftarrow$  writes  $\cup$  {(addr, v)}
21 TxRMW(addr, f)
22   rmws  $\leftarrow$  rmws  $\cup$  {(addr, f)} // on an RMW, log the address and operation
23 TxCOMMIT()
24   if writes =  $\emptyset$   $\wedge$  rmws =  $\emptyset$  then return // immediate return for read-only transactions
25   ACQUIRELOCKS()
26   end  $\leftarrow$  ATOMICINC(&timestamp, 1)
27   if start  $\neq$  end - 1 then VALIDATEBOTH() // validate both read set and rmws set
28   WRITEBACK()
29   REPLAY() // replay RMWs
30   RELEASELOCKS(end)

```

ALGORITHM 2: Helper Functions

```

1 ACQUIRELOCKS()
2   foreach addr in writes do
3     o  $\leftarrow$  orecs[addr]
4     if o.LOCKED()  $\vee$  o > start then ABORT()
5     if  $\neg$ CAS(&orecs[addr], o, my_lock)
6     then
7       | ABORT()
7     locks  $\leftarrow$  locks  $\cup$  {addr}
8   foreach m in rmws do
9     m.orec  $\leftarrow$  orecs[m.addr]
10    o  $\leftarrow$  m.orec
11    if o = my_lock then continue
12    if o.LOCKED() then ABORT()
13    if  $\neg$ CAS(&orecs[m.addr], o, my_lock)
14    then
15      | ABORT()
15    locks  $\leftarrow$  locks  $\cup$  {addr}

16 PROMOTE(addr)
17   if addr  $\in$  writes then v  $\leftarrow$  writes[addr]
18   else
19     while true do
20       o1  $\leftarrow$  orecs[addr]
21       v  $\leftarrow$  *addr
22       o2  $\leftarrow$  orecs[addr]
23       if o1 = o2  $\wedge$  o2  $\leq$  start then
24         | reads  $\leftarrow$  reads  $\cup$  {(addr, o1)}
25         | break
26       else if  $\neg$ o2.LOCKED() then EXTEND()
27       else ABORT()

28   foreach m in rmws do
29     | if addr = m.addr then v  $\leftarrow$  m.f(v)
30   rmws  $\leftarrow$  rmws  $\setminus$  {(addr, -)}
31   writes  $\leftarrow$  writes  $\cup$  {(addr, v)}
32   return v

33 REPLAY()
34   foreach m in rmws do
35     | *m.addr  $\leftarrow$  m.f(*m.addr)

36 VALIDATEBOTH()
37   foreach (addr, v) in reads do
38     foreach m in rmws do
39       | if m.addr = addr  $\wedge$  m.orec  $\neq$  v
40       | then
41         | ABORT()
41     o  $\leftarrow$  orecs[addr]
42     if o  $\neq$  v  $\wedge$  o  $\neq$  my_lock then ABORT()

43 VALIDATEREAD()
44   foreach (addr, v) in reads do
45     | o  $\leftarrow$  orecs[addr]
46     | if o  $\neq$  v  $\wedge$  o  $\neq$  my_lock then ABORT()

47 EXTEND()
48   t  $\leftarrow$  timestamp
49   VALIDATEREAD()
50   start  $\leftarrow$  t

51 RELEASELOCKS(end)
52   foreach addr in locks do
53     | orecs[addr].RELEASETO(end)

54 WRITEBACK()
55   foreach (addr, v) in writes do
56     | *addr  $\leftarrow$  v

56 ABORT()
57   foreach addr in locks do
58     | orecs[addr].RELEASETOPREVIOUS()
59   rollback

```

discard speculative states and restart whenever they observe inconsistencies. The use of incremented values of timestamps as the version numbers is the key to providing this property efficiently.

2.2. Problem: Aborts on Read-Modify-Write

RMW operations on highly contended locations can open a window of vulnerability that makes transactions likely to abort. To illustrate the problem, we focus on lazy ore-based STM implementations. In these implementations, a TxWRITE operation logs the location and the new value in a transaction-local write set; these locations are acquired at the end of the transaction (in TxCOMMIT) and then written back to the shared memory in order to be visible to other transactions. Since TxWRITE operations are local, they never cause the transaction to abort or violate opacity. In contrast, TxREAD operations must check the ore of the location to ensure that reading the new location results in a view of memory consistent with all prior reads (i.e., equivalent to an order in which the

transaction ran in isolation). The transaction aborts if such check indicates a potential inconsistency.

A RMW operation takes a function f and a location X as parameters. It applies function f to the value x stored at location X and writes back the new value $f(x)$ to X . We assume the function f is pure. Using the existing TM interface, an RMW operation in a transaction is instrumented as follows:

```
RMW( $X$ ,  $f$ ) :
   $x \leftarrow \text{TxRead}(X)$ ;
   $\text{TxWrite}(X, f(x))$ ;
```

For a given transaction T , executing an RMW operation on location X makes the transaction vulnerable to conflicts on X . In most implementations, the transaction aborts if either (1) X is already acquired when T performs the TxRead , or (2) any other transaction writes to X and commits before T .

Let us now return to our example in Figure 1. Suppose that, in an execution trace, the function calls by concurrent transactions T_1 and T_2 both observe only unlocked orecs with versions equal to 0 and that these sets of orecs observed by the transactions have an empty intersection. We also assume that the orec related to `stats` has an initial value of 0. Suppose T_1 commits after T_2 has read `stats`: It will lock `stats`'s orec, update the global timestamp to N , and then update `stats`'s orec version to N , where N is larger than T_2 's start time. As a result, T_2 must abort. The abort will occur either when T_2 tries to lock `stats`'s orec within TxCommit and sees that N is greater than the time at which it was last valid (either T_2 's start time or the time of its last validation) or else on account of a validation within the Extend function, which would also observe a "too new" orec value of N .

We say an RMW operation by transaction T is *final* if the location it modifies is not subsequently read or written by T . Our key observation is that a final RMW operation can be reordered to execute at the end of the transaction. Clearly, the transformation preserves the semantics of the transaction because, by definition, there is no dependency between the RMW and any following instruction. We achieve this transformation by replacing an RMW not with the sequence described previously, but instead with a single call to TxRMW that takes the address, a function, and an optional operand to the function. Trivial uses include the $++$ function, which takes no operand, and the $+=$ operator, which takes a scalar operand.

The transformation can improve performance by reducing the contention between transactions. In contrast to explicitly using a TxRead , executing a delayed RMW on location L prevents the transaction from being immediately vulnerable to conflicts. Instead, the transaction is first vulnerable to conflicts on L at the time the RMW is performed, during TxCommit .

2.3. The Basic Algorithm

Our basic algorithm to support transactional RMW operations is presented in Algorithm 1. When instrumenting source code with transactional constructs, candidate RMW operations are replaced with calls to TxRMW , which allocates an entry for the RMW operation and appends it to the RMW log (`rmws`). The TxRead , TxWrite , and TxCommit functions are extended to interact with the log. The design is guided by the assumption that the number of RMW operations tends to be small in comparison to read and write operations. Note that it would be possible to promote a TxRMW to a TxWrite if it is known that the location was previously read or written but at the cost of searching the read and write sets.

On a TxRead to location L , a lookup is performed in `rmws` (line 5). If delayed RMW operations on L are found, they are immediately transformed into their corresponding

TxREAD and TxWRITE operations via the PROMOTE function. This ensures that the return value of the TxREAD is consistent with an execution in which all delayed RMWs occurred before the TxREAD.

A TxWRITE operation on location L must check if L was previously modified by any RMW operations, and it removes all such entries from the `rmws` log. In this manner, we ensure that the effects of any prior RMWs on L are made obsolete by this TxWRITE. That is, these stale RMWs to L will not be performed at commit time. However, it is also necessary to preserve the implicit read of L within the RMW. To this end, we perform a read of L (line 19).

Last, to ensure that an RMW performed on L observes prior writes to L by the transaction, we order REPLAY after WRITEBACK in the TxCOMMIT function.

2.4. Correctness

For the time being, we focus on correctness in the absence of the publication idiom [Menon et al. 2008]. The algorithms presented so far and the discussion of correctness here are applicable to languages that enforce static separation [Abadi et al. 2008], as well as to programs that allow for privatization of shared data. Section 4 discusses algorithmic extensions needed for publication safety.

When implementing delayed RMWs within an STM system, we require the following properties:

- Opacity:** If transaction T_1 performs a read of X and then a delayed operation on X , then T_1 must abort if a concurrent transaction modifies X before T_1 commits.
- Atomic commit:** A delayed operation must not appear to happen either *before* or *after* the remainder of the transaction, but *atomically* with the rest of the transaction.

Preserving opacity is relatively simple. Calls to TxRMW do not remove entries from reads. Thus, if there is a delayed RMW on L , and L was previously read by the transaction, then during the execution of the transaction, conflicts on L with other transactions will still be detected when the transaction validates (in the EXTEND function). Note that since the RMW itself was delayed, its read cannot participate in the observation of inconsistencies by transactions until a call to PROMOTE, which transforms the RMW into a read, or TxCOMMIT, at which point inconsistencies are not visible to user code.

The second challenge is ensuring atomicity at commit time. There are again two requirements. First, we must ensure that updates performed via RMWs are atomic with respect to all other writes performed by the transaction. TxCOMMIT ensures this via two-phase locking: It first uses ACQUIRELOCKS to acquire locks covering all locations in both the writes and `rmws` sets. Only when this operation succeeds in locking all to-be-modified locations will the transaction call WRITEBACK and REPLAY to update memory, and no locks are released until after all updates are performed. In this manner, all writes, regardless of source, occur in a single critical section that appears atomic to concurrent transactions.

Second, we must ensure that the RMWs happen only at a time when all reads performed by the transaction are known to be valid. In the absence of delayed RMWs, this is achieved in TxCOMMIT by acquiring all locks then validating. If the validation succeeds, then all reads were valid at a time when all locks were held. However, there is a subtlety here: In lines 2–7 of ACQUIRELOCKS, locations are locked only if their version number is no greater than the transaction start time. This allows a simplification in VALIDATEREAD (line 46): If the location being validated is owned by the validating transaction, it is known that the read was valid. Unfortunately, we cannot place the same constraint on locks acquired for RMW operations or else much of the benefit of delayed RMWs would be lost. Instead we add lines 8–15 to ACQUIRELOCKS, so that RMW locations can be locked as long as the lock is currently unheld, even if the version

number is “too new.” We must then guard against the situation where an RMW was performed to location L after L was read, but there was an intervening modification to L by another transaction. This ordering is detected by storing the version at the time an RMW location was locked (ACQUIRELOCKS line 9). Then, during read set validation, we check if a location being validated was locked for RMW via VALIDATEBOTH lines 38–40. If so, we verify that the version number when the lock was acquired was no greater than the version of the lock at the time of the read.

3. IMPLEMENTATION

The naïve implementation of TxRMW in Algorithm 1 suffers from high asymptotic complexity: Since the `rmws` log stores an ordered list of delayed RMWs, if there are M RMW operations and R reads, there can be $O(M)$ overhead on each of R calls to line 5 of TxREAD and another $O(R \times M)$ overhead due to lines 38–40 of VALIDATEBOTH. Furthermore, if there are W calls to TxWRITE, there can be up to $O(W \times M)$ overhead to eliminate RMWs to locations that are overwritten (lines 17–19). We now present mechanisms for decreasing these overheads. We also discuss alternative mechanisms for guiding the compiler to produce calls to TxRMW.

3.1. TxRMW via Live-Out Analysis

The effort required to delay RMW operations is minimal, but the need to iterate through delayed RMWs at commit time necessarily leads to a small overhead. It is thus wise to avoid delaying every RMW operation. A simple heuristic is to delay an RMW only if the value produced by the operation is not live. Note that liveness analysis need not be constrained to RMW operations: It could be employed to defer individual writes. For simplicity, and to avoid delaying writes that are ultimately re-read (such as during a tree rebalance), we propose that the compiler reserve use of TxRMW for increment/decrement and compound assignment operators (i.e., `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, and `|=`) that take as an operand the address of a nonlocal variable and produce a value that is not live. Note that this analysis must be done in an early compiler pass, or these operators may already be lowered to loads and stores.

Because a compiler-based approach is likely to produce a large number of delayed operations, and hence a large value of M , it is beneficial to reduce the asymptotic cost of checking when a read and RMW are performed to the same location. To reduce the cost of line 5, it is possible to maintain a hash table storing all addresses involved in RMWs by the current transaction, so that each execution of line 5 has constant overhead. Alternatively, a technique proposed for TL2 [Dice et al. 2006] can be used, where a small Bloom filter [Bloom 1970] approximates the locations in `rmws` and can be consulted before performing a lookup.

To reduce overhead at commit time, we recommend a technique pioneered by the Amino CBB STM algorithm [IBM 2010]. In this technique, each `orec` has a second field that stores the previous version. When a transaction acquires an `orec`, it stores the old version number in the second field. At commit time, Amino acquires any unlocked `orecs`, even if the `orec` version number is “too new.” Then, during validation, any read whose `orec` is locked by the current transaction checks the second field to determine if the lock was acquired at a time when the read was consistent. While Amino CBB STM used this technique to minimize aborts for transactions that wrote to locations they did not read, we observe that it also eliminates $R \times M$ overhead at commit time for our algorithm.

3.2. TxRMW via Programmer Annotation

We expect the overhead of maintaining a hash table to be high and the imprecision of a Bloom filter to also be high. We are also concerned that a compiler might prove too

aggressive in its application of delayed TxRMW calls. Additionally, a programmer may have better insight into the frequency with which certain locations are accessed after an RMW, particularly with profiler feedback, and thus might wish to use TxRMW for some updates to variables that are subsequently accessed only on an uncommon code path.

For these cases, we propose an annotation that distinguishes between variables that can only be read and written and variables that can be read, written, and used in RMW operations. Only accesses to objects of the latter category incur the extra overheads associated with delayed RMWs.

Algorithm 3 refines Algorithm 1 to distinguish between accesses to annotated variables (which call TxRMWREAD, TxRMWWRITE, and TxRMW) and accesses to nonannotated variables (which call TxREAD and TxWRITE). These changes require a `rdaddrs` log to store the addresses of annotated variables that are read.

The annotations ensure that only TxRMWREAD promotes delayed RMW operations (line 19), and only TxRMWWRITE needs to discard pending RMWs (lines 22–24). These changes reduce two sources of overhead by eliminating lookups for common-case reads and writes. The third source of overhead in the naïve algorithm relates to RMW operations that follow reads to the same location and manifests as extra overhead during commit-time validation. To avoid this cost, we leverage the expectation that `rdaddrs` is small and perform a lookup in `rdaddrs` during every TxRMW. When there is a match, we do not delay the RMW but instead perform it immediately. As before, RMWs that follow writes do not require a special case because REPLAY follows WRITEBACK.

We assume that annotated RMW operations are infrequent since (a) a high number of contention hotspots would suggest that the program cannot scale, and (b), unlike live-out analysis, annotations do not silently convert many operations into delayed operations. This being the case, overheads should be significantly lower using annotations. With W writes and R reads of nonannotated variables, and W' writes, R' reads, and M' RMWs of annotated variables, we can expect the overhead of supporting delayed RMWs to drop to $O(R' \times M')$ in TxRMWREAD, and $O(W' \times M')$ in TxRMWWRITE. TxCOMMIT no longer has any overhead due to delayed RMWs, but TxRMW incurs $O(R' \times M')$ overhead. Note that this last quantity could be avoided with Amino-style orecs, but not doing so avoids adding additional instructions to commit-time validation. Since locks are held during this validation, and since both R' and M' should be small, we expect $O(R' \times M')$ overhead to be insignificant in practice.

3.3. Optimized Programmer Annotations

A final optimization appears in Algorithm 4. In this algorithm, we again expect the programmer to annotate the declaration of highly contended variables for which delaying RMWs may be profitable. However, we now assume that highly contended variables are rarely read or written in the same transactions as those that access annotated variables with RMW operations.

In this implementation, we no longer store a log of all annotated locations that the transaction has read. Instead, we use a boolean to remember if any annotated location has been accessed via a read or write operation by the current transaction. If there is any such read or write, then all delayed RMWs are immediately performed, and future RMWs by the transaction will not be delayed.

Although this approach is extremely aggressive (*any* TxRMWREAD or TxRMWWRITE disables delayed RMWs in the transaction even when the RMWs and other annotated accesses are to disjoint sets of locations), it has the least overhead. There is no commit-time overhead, and there is at most M' overhead incurred by all reads and writes within a transaction.

ALGORITHM 3: An algorithm for delayed RMWs that assumes the variables involved in delayed RMWs are annotated.

```

1 TxBEGIN()
2   start  $\leftarrow$  timestamp
3   reads, writes, locks, rmws, rdaddrs  $\leftarrow$   $\emptyset$ 
4 TxREAD(addr)
5   if addr  $\in$  writes then return writes[addr]
6   while true do
7     o1  $\leftarrow$  orecs[addr]
8     v  $\leftarrow$  *addr
9     o2  $\leftarrow$  orecs[addr]
10    if o1 = o2  $\wedge$  o2  $\leq$  start then
11      reads  $\leftarrow$  reads  $\cup$  {(addr, o1)}
12      return v
13    else if  $\neg$ o2.LOCKED() then EXTEND()
14    else ABORT()
15 TxWRITE(addr, v)
16   writes  $\leftarrow$  writes  $\cup$  {(addr, v)}
17 TxRMWREAD(addr)
18   rdaddrs  $\leftarrow$  rdaddrs  $\cup$  {addr}
19   if addr  $\in$  rmws then return PROMOTE(addr)
20   return TxREAD(addr)
21 TxOPWRITE(addr, v)
22   if addr  $\in$  rmws then
23     rmws  $\leftarrow$  rmws  $\setminus$  {(addr, -)}
24     TxREAD(addr)
25   TxWRITE(addr, v)
26 TxRMW(addr, f)
27   if addr  $\in$  rdaddrs then
28     v  $\leftarrow$  TxREAD(addr)
29     v  $\leftarrow$  f(v)
30     TxWRITE(addr, v)
31   else rmws  $\leftarrow$  rmws  $\cup$  {(addr, f)}
32 TxCOMMIT()
33   if writes = rmws =  $\emptyset$  then return
34   ACQUIRELOCKS()
35   end  $\leftarrow$  ATOMICINC(&timestamp, 1)
36   if start  $\neq$  end - 1 then VALIDATE_READ()
37   WRITEBACK()
38   REPLAY()
39   RELEASELOCKS(end)

```

4. IMPACT ON SEMANTICS

Menon et al. [2008] proposed several levels of transactional semantics that require varying amounts of serialization at the boundaries of transactions and that place varying restrictions on how programmers can transition data between a state in which they

ALGORITHM 4: Aggressive optimizations for the common case: If any annotated location is read or written, delayed RMWs are disabled for the transaction.

```

1 TxBEGIN()
2   start  $\leftarrow$  timestamp
3   reads, writes, locks, rmws  $\leftarrow$   $\emptyset$ 
4   rdflag  $\leftarrow$  false

5 TxREAD(addr)
6   if addr  $\in$  writes then return writes[addr]
7   while true do
8     o1  $\leftarrow$  orecs[addr]
9     v  $\leftarrow$  *addr
10    o2  $\leftarrow$  orecs[addr]
11    if o1 = o2  $\wedge$  o2  $\leq$  start then
12      reads  $\leftarrow$  reads  $\cup$  {(addr, o1)}
13      return v
14    else if  $\neg$ o2.LOCKED() then EXTEND()
15    else ABORT()

16 TxWRITE(addr, v)
17   writes  $\leftarrow$  writes  $\cup$  {(addr, v)}

18 TxOPREAD(addr)
19   if  $\neg$ rdflag then
20     rdflag  $\leftarrow$  true
21     for each m in rmws do PROMOTE(m.addr)
22   return TxREAD(addr)

23 TxOPWRITE(addr, v)
24   if  $\neg$ rdflag then
25     rdflag  $\leftarrow$  true
26     for each m in rmws do PROMOTE(m.addr)
27   TxWRITE(addr, v)

28 TxRMW(addr, f)
29   if rdflag then
30     v  $\leftarrow$  TxREAD(addr)
31     v  $\leftarrow$  f(v)
32     TxWRITE(addr, v)
33   else rmws  $\leftarrow$  rmws  $\cup$  {(addr, f)}

34 TxCOMMIT()
35   if writes = rmws =  $\emptyset$  then return
36   ACQUIRELOCKS()
37   end  $\leftarrow$  ATOMICINC(&timestamp, 1)
38   if start  $\neq$  end - 1 then VALIDATEREAD()
39   WRITEBACK()
40   if  $\neg$ rdflag then REPLAY()
41   RELEASELOCKS(end)

```

are accessed via transactions and a state in which they are accessed nontransactionally. At the most basic level, these semantics propose different levels of support for the publication idiom: When a thread initializes private data and then uses a transaction to mark those data as visible to other threads, the underlying STM must ensure that

Initially: data == 42, ready == false, val == 0	
Thread 1:	Thread 2:
1	1 transaction {
2	2 tmp = data;
3 data = 1;	3
4 transaction {	4
5 ready = true;	5
6 }	6
	7 if (ready)
	8 val = tmp;
	9 }
	Can val == 42?

Fig. 2. Basic publication example (reproduced from Figure 1 of Menon et al. [2008]). The vertical ordering of instructions is meant to imply the execution order on a sequentially consistent machine.

all threads agree that the initialization happens before the transaction; otherwise, a thread might see that the data are marked as safe to access, but then observe the uninitialized data.

The different levels of semantics differ in terms of which racy publication idioms are allowed in Java programs. However, the two least restrictive levels, Asymmetric Lock Atomicity (ALA) and Encounter-time Lock Atomicity (ELA), are both applicable to C++, where racy code is erroneous. In the context of C++, ALA and ELA differ in whether the compiler can reorder reads of a datum that might be concurrently initialized outside of a transaction.¹

The canonical example appears in Figure 2, where ALA ensures that the race accessing data is benign and does not produce the erroneous output `val == 42`. Note that when all transactions are protected by a single global lock, the race is also benign because 42 can never be used by Thread 2. Under ELA semantics, neither the programmer nor the compiler is permitted to transform the code `if (ready) val = data;` into the code run by Thread 2.

Now consider an extension in which the “ready” flag is a counter, where zero indicates that data is not initialized, and any other value is the number of transactions that have used data in a successful transaction. This code appears in Figure 3. While admittedly contrived, we hope the reader agrees that this code is not unrealistic. For example, a naïve transactionalization of legacy code that includes `auto_ptr` could result in this code.

ALA provides the illusion that all read locks are acquired at transaction begin, but write locks can be delayed until commit time. ELA, in contrast, gives the appearance that read locks are acquired immediately before the first read of the corresponding location within the transaction (write locks are acquired as in ALA).

If the RMW on line 2 is not delayed, the example in Figure 3 is correct for both ELA and ALA, and `val` will never equal 42. With delayed RMWs, the example breaks under ELA semantics: The read of `ready` by Thread 2 will result in a call to `TxPromote` and effectively cause the read and write of `ready` to occur *after* Thread 2’s transaction commits, instead of before `data` is read. Thus, an STM implementation that only provides ELA semantics cannot naïvely delay RMWs without risking publication violations. Note that this finding extends the work of Menon et al., which showed that,

¹Note that since Menon’s work focused on the Java language, there were additional constraints on the TM implementation that cannot be entirely separated from publication-related issues. For example, in a racy program, Java forbids out-of-thin-air reads [Manson et al. 2005]; thus, even at the weakest semantics levels, Java appears to require STM algorithms to use redo logs. In C++, an STM may use undo logs since the behavior of racy programs is undefined. Throughout this section, we ignore Java-specific issues related to undo logs. The algorithms presented herein, if applied to an STM algorithm with redo logs, would be valid for Java.

Initially: data == 42, ready == 0, val == 0

Thread 1:	Thread 2:
1	1 transaction {
2	2 ready++;
3	3 tmp = data;
4 data = 1;	4
5 transaction {	5
6 ready = 1;	6
7 }	7
8	8 if (ready > 1)
9	9 val = tmp;
10	10 else
11	11 ready--;
12	12 }

Can val == 42?

Fig. 3. Publication violation example with delayed RMWs.

under ELA, dependent reads cannot be reordered above the reads that establish a control-flow dependency. Here, dynamic reordering of accesses that are not dependent are similarly unsafe for both C++ and Java. With ALA or stronger semantics, delaying the RMW remains safe.

For STM algorithms that use ownership records and provide ELA publication safety, such as those presented in Section 2, we can resolve this problem so that delayed RMWs are safe. We extend the `rmws` data structure to store an additional field. Then, in `TxRMW`, we begin by reading the `orec` that corresponds to the address parameter and storing it along with the description of the delayed RMW, in this extra field.

If the delayed RMW is not performed until `REPLAY` in the `TxCOMMIT` function, then the added field is ignored. However, if the RMW is performed earlier on account of a call to `PROMOTE`, then we use the field. In `PROMOTE`, we replace line 20 with $o_1 \leftarrow z$, where z is the value stored in the field, and we remove line 26.

These changes restore publication safety by guaranteeing that a delayed RMW can only be promoted if the promotion is indistinguishable from a `TxREAD` and `TxWRITE` succeeding at the time the RMW was initially requested. By requiring `PROMOTE`'s read (lines 19–27) to fail unless the `orec` is unlocked and storing the same value as stored in `rmws`, the read no longer appears to delay until the time `PROMOTE` was requested. Note that the write performed by `PROMOTE` can still be delayed until commit time.

5. EVALUATION

In this section, we evaluate the performance of our mechanism for delaying RMW operations within transactions. We consider microbenchmarks, the STAMP benchmark suite [Minh et al. 2008], and a transactional version of the memcached application [Ruan et al. 2014b]. STM experiments were performed on a dual-chip 2.67GHz Intel Xeon 5650 system with 12GB of RAM and 12 cores/24 threads. HTM experiments used a single-chip 3.40GHz Intel Core i7-4770 with 4 cores/8 threads. Both machines run Ubuntu Linux 13.04, kernel 3.8.0-21, and a pre-release 4.9 GCC compiler with `-O3` and `-m64` flags. Results are the average of five trials that led to uniformly low variance.

5.1. Systems Evaluated

The default GCC STM implementation uses an eager STM in which locks are acquired on first write access by a transaction, and updates are performed immediately, with an undo log for recovering from aborts. A critical feature of this implementation is that it is correct for arbitrary C code: Well-known pitfalls [Yoo et al. 2008] related to unaligned accesses, unsafe casting, and overlapping accesses to variables of different sizes are all handled correctly. We added a new commit-time locking algorithm to GCC to assess the

effect of our delayed RMWs on both eager and lazy STM. Our implementation stores buffered writes in a tree, incurring $O(\log(n))$ overhead on each write-set lookup. To the best of our knowledge, our implementation is the only correct lazy STM for GCC; it also satisfies all of Menon's requirements for a Java STM implementation. Throughout this section, experiments using STM algorithms derived from the default GCC algorithm are labeled "Eager," and those derived from our commit-time locking algorithm are labeled "Lazy." Both algorithms provide ELA semantics by default.

On machines with Intel TSX support, GCC also offers an "HTM" runtime that attempts to run transactions in hardware and falls back to a single global lock for transactions that fail to commit. Causes of serialization include contention (multiple aborts by the same transaction), capacity overflow (accessing more distinct cache lines than the transactional hardware can monitor), and interrupts/exceptions (to include TLB misses).

For the Eager and Lazy STM systems, we compare performance against three implementations of our delayed RMW mechanism. Experiments labeled "naïve" use Algorithm 1 and do not take advantage of annotations to avoid lookups in the `rmws` log on every read and write. Experiments labeled "Annotated" correspond to the use of Algorithm 3, where we can statically distinguish between reads and writes to locations that might have delayed RMWs and those that do not. Finally, experiments labeled "Flag" indicate the use of Algorithm 4, which optimizes for the case where delayed RMWs tend to be to locations that are not otherwise accessed by the transaction. Unless otherwise noted, experiments do not include the extensions from Section 4.

We also added support for RMW operations to the GCC HTM runtime. For these tests, we use a variant of the "Annotated" algorithm: Regular reads and writes to shared memory from within a transaction do not incur function call overhead, but RMW operations and reads and writes to annotated variables are implemented as function calls into the TM library. The commit function is also slightly more complex because it must call *Replay*.

5.2. Microbenchmark Performance

The purpose of our microbenchmarks is to evaluate the performance of delayed RMWs in a predictable but somewhat irregular workload. Red-black trees are popular in STM research precisely because they involve complex pointer chasing and transactions with varying numbers of reads and writes but still have few conflicts and ought to scale well. By adding statistics counters to scalable data structure workloads, we can produce workloads with few conflicts other than at RMW hotspots. The expectation is that our mechanism should reduce the effect of the hotspots and improve scalability.

We consider two microbenchmarks based on the red-black tree implementation from the RSTM library [Marathe et al. 2006]. This tree implementation generally offers good scalability with no internal bottlenecks, it but provides only insert, remove, and lookup operations; there are no methods for iteration or statistics (e.g., tree size). We configured the tree experiments to perform an equal mix of insert, lookup, and remove operations, using 20-bit keys, and we prefilled the tree with 2^{19} entries. Charts present the average of five 5-second trials.

Our first variant of the tree adds a vector of counters. On every lookup (whether a distinct operation or as the first step of an insert or remove), the depth at which the lookup terminated determines which counter to increment. In this manner, all searches that terminate at the N th level of the tree will conflict on the N th counter. Each counter is padded to the size of a cache line to prevent false sharing, and the counters are the only operations that use TxRMW. This introduces a modest amount of contention and also results in a workload in which no transaction can take the read-only fast-path that is common in STM commit functions.

Fig. 4. Red-Black Tree experiments augmented with a global vector of counters to monitor the height at which searches terminate.

Fig. 5. Red-Black Tree experiments augmented with a global counter to monitor the number of elements in the tree.

Figure 4 presents the throughput for the vector-of-counters tree. The Eager baseline scales poorly; the Lazy baseline is even worse due to wasted work performed by transactions that conflict on a counter but do not detect the conflict until commit time. When our mechanism is used to defer RMWs on the counters, throughput increases dramatically, and the performance difference between eager and lazy vanishes. As expected, the variations on our mechanism that lower asymptotic overhead lead to the best performance, but even the naïve implementation offers significant improvement.

On the HTM system, the benchmark scales poorly even when the RMW hotspots are removed. Although we do observe a benefit for our technique, we caution the reader against drawing broad conclusions: When the scalability concerns are resolved, the workload is likely to scale, at which point the curves are likely to change dramatically. Nonetheless, the microbenchmark’s predictable behavior allowed us to confirm that our RMW techniques are compatible with HTM, and the added overheads to support delayed RMWs introduced little latency while preventing some aborts.

Our second variant of the tree considers adding a *size* method. We add a counter to record the number of elements in the set and modify the counter on every successful insertion or removal. This test mirrors the implementation of collections in the C++ Standard Template Library (STL): Lists, maps, and other collections in the STL all have counters to ensure that the *size* operation takes $O(1)$ time. The counter, naturally, is a source of contention. It is the only field in the data structure that uses TxRMW.

Figure 5 presents the throughput of the tree with a global count field. Since lookup transactions do not access the counter, their presence enables some scalability for the baseline algorithm. However, the counter becomes a bottleneck beyond two threads, dampening scalability significantly. Although some amount of dampened scalability is unavoidable because the cache lines holding the counter and its associated ownership record must move between cores, the difference between our algorithms and the baseline shows that some of the dampened scalability is due to aborts and that those aborts can be prevented by delaying RMWs to hot locations.

Table II. Frequency of RMW Operations in STAMP Benchmarks

Benchmark	Description
Vacation	RMWs are only to locations that are also read or written; no RMWs can be deferred until commit time. There are also RMWs within the list object used by the benchmark, but they already happen near the end of a large transaction.
Yada	Only one transaction contains RMWs, and it consists only of RMWs. The benchmark uses an AVL tree, which contains an RMW on its size field at the end of insertions. AVL tree insertions happen at the middle or end of transactions. The benchmark uses a heap, which has an RMW on its size. However, the heap size is always read prior to the RMW. The benchmark uses a vector, with RMWs on both the size and capacity fields. As with the heap, these fields are read before the RMW. RMWs to the list object's size field are also observed at the middle or end of transactions.
Genome	One large transaction ends with an RMW. RMWs within list operations occur only at the end of transactions.
SSCA2	One short transaction begins with an RMW. Other transactions consist solely of RMWs.
KMeans	Some large transactions consist solely of RMWs.
Bayes	The only RMWs are on the list's size field. These occur at the end of transactions.
Intruder	The only RMWs are to the list's size. These RMWs occur in the middle of the transaction, but the size is always read after the RMW.
Labyrinth	No RMWs.

As in the previous experiment, we see that the benchmark scalability characteristics are very different for HTM than for STM. With 20-bit keys, the average transaction accesses 20 unique locations, and the tree itself contains approximately a half million objects. Since accesses are random, and the TLB only holds 64 entries, many transactions experience a TLB miss. These misses cause HTM transactions to abort, then serialize, then retry, resulting in poor scaling on the HTM machine.

5.3. STAMP Performance

Next, we ran experiments on the STAMP benchmark suite [Minh et al. 2008]. Table II describes the frequency of RMW operations in STAMP. We discuss RMWs within the benchmark code separately from RMWs in libraries used by each data structure. Recall that our mechanism works best when transactional RMWs are followed by non-RMW work.

It is clear that the STAMP benchmark suite does not afford many opportunities to delay RMW operations. In most cases, there are either no RMW operations, or RMW operations occur at the end of the transaction. We believe this is a situation in which STAMP is not representative of real-world code. In particular, the red-black tree used by STAMP was written by concurrency experts at Sun Microsystems and hence does not include a counter. The other data structures (heap, list, AVL tree), designed elsewhere, do contain counters. If STAMP used the C++ STL, then all collections, including the much-used red-black tree, would have counters and would be more favorable toward our optimizations.

Note that STAMP uses a library interface to interact with the TM runtime rather than adhering to the Draft C++ TM Specification. To employ our extensions to GCC, we had to substantially modify the benchmarks, not only to annotate RMW operations, but also to make STAMP compatible with GCC's TM implementation. In particular, we had to remove or replace unsafe function calls and devise alternatives to the nontransactional reads that STAMP sometimes performs from within an atomic transaction. Thus, these results do not always correspond directly to prior published work. More details were recently published by Ruan et al. [2014a].

Fig. 6. STAMP results on the STM machine. HC and LC refer to high- and low-contention command-line configurations.

Figure 6 presents results for STAMP on the STM system. The “Flag” and “Annotated” benchmarks perform indistinguishably, so we present only the more general “Annotated” algorithm. We see a slight improvement for delayed RMWs in most cases for both Eager and Lazy STM. KMeans is an outlier: Its transactions consist entirely of RMWs, and, when contention is low, delayed RMWs can add overhead without avoiding aborts. This is especially true for a lazy STM, where aborts are an order of magnitude less frequent to begin with. Results for Bayes are not trustworthy, as Bayes exhibits high variability from one run to the next even in the absence of our mechanism.

Figure 7 repeats these experiments on the HTM machine. As with the microbenchmark experiments, we see that HTM trends are not always the same as STM trends. Most noticeably, since our SMT machine has four cores and eight hardware threads, experiments with more than four software threads rarely scale: The effective write capacity of HTM transactions is halved when the L1 cache is shared. Furthermore, many transactions serialize even at one thread. These serializations can be due to many factors, including TLB misses (as in Section 5.2) and overflowing the capacity of the cache [Yoo et al. 2013]. Naturally, once transactions serialize, delaying RMWs to the end of a transaction offers no benefit.

Nonetheless, we observe that the overhead of added instrumentation is not significant and that RMW operations typically offer a small improvement. The more important result is demonstrating that the added instructions and logging of our mechanism do not significantly affect HTM performance, and, even when opportunities to delay RMWs are rare, we are able to achieve an improvement on first-generation transactional hardware.

Taken as a whole, we observe that since STAMP transactions do not exhibit the pattern first described in Figure 1, the benefit of delaying RMWs is small. Although we do not incur noticeable overhead for delaying RMWs, the contention hotspots they are

Fig. 7. STAMP results on the HTM machine. HC and LC refer to high- and low-contention command-line configurations.

Fig. 8. Memcached performance on a two-chip, 12-core system.

designed to avoid are rare in STAMP, and thus our mechanism cannot substantially improve STAMP performance. We expect the impact to be greater on production systems, where software is more likely to use standard libraries and to employ statistics counters.

5.4. Memcached Performance

Last, we look at a real-world application that we expect to possess the attributes lacking from STAMP. We evaluate memcached, following the experiment configuration of Ruan et al. [2014b]. We use memslap to produce a workload, and we run memcached and memslap on the same machine to limit the effect of the network. The configuration results in a number of operations proportional to the number of threads: Flat curves indicate perfect scaling, higher values represent slowdown. Note that this configuration results in SMT effects beyond two threads on the HTM machine. Consequently, we report only STM results. Performance appears in Figure 8. Note, too, that we only

instrumented memcached statistics counters as opposed to all RMWs in memcached. This results in a program that matches the pattern from Figure 1.

5.4.1. Eager STM. Figure 8(a) compares the performance of the baseline eager GCC algorithm to three variants using our different delayed RMW algorithms. At low thread counts, we do not observe a significant difference in performance. Starting at four threads, the performance of the naïve algorithm becomes noticeably worse than the others. To gain more insight, we instrumented GCC to report abort rates. At 12 threads, the baseline GCC algorithm reached as high as 20 aborts per commit. We then elided all accesses to statistics counters, to assess the ideal performance (note that the values of statistics counters rarely affect program behavior in memcached). When the counters were removed, we observed neither a change in the abort rate, nor a change in overall running time.

In GCC's Eager STM, when a transaction encounters a locked ownership record, it immediately aborts, releases its locks, and restarts. This can rapidly inflate abort rates: If transaction T_L locks ownership record O , and transaction T_R attempts to read a location protected by O early in its execution, then T_R can experience dozens of aborts in a short time interval. More importantly, in memcached, many read/write conflicts appear to manifest early during transaction execution. Since a write to L by T_L in GCC's TM causes all conflicting transactions to convoy behind T_L (this is a natural consequence of eager TM and workloads with frequent conflicts), our delayed RMWs were not playing any beneficial role: The statistics counters were not, in reality, highly contended because by the time a transaction reached the point where it attempted to RMW a counter, it had already locked enough of its write set to prevent concurrent transactions from being able to reach their instructions for accessing the counter.

5.4.2. Lazy STM. The previous discussion illustrates a surprising consequence of eager TM: Early locking may constrain the speculative execution of transactions and interfere with scalability. However, doing so can also lead to a livelock-free execution because initial progress by a transaction T_L prevents other transactions from reaching code that could cause them to acquire locations that T_L will access in the future. Absent these later acquires, a cyclic dependency cannot be formed, and livelock will not occur.

In a similar manner, the results in Figure 8(b) show the consequence of laziness: Performance degrades significantly at high thread counts due to wasted work. When two transactions have conflicting accesses, both continue executing until one commits, at which point the other becomes invalid. Again, there is no livelock, but now the problem is that aborts are too infrequent. Indeed, aborts are an order of magnitude less common with the lazy algorithm than with the eager algorithm. However, now speculation is continuing past the point where it could be known that the speculation will not be profitable.

Since transactions speculate past their first read/write conflict, more transactions reach their accesses of statistics counters. This, in turn, allows us to observe the impact of delayed RMW operations. In Figure 8(b), even the naïve algorithm improves performance, and algorithms that use annotations do even better, reaching more than 20% improvement at 12 threads. Furthermore, the improvement correlates directly with a decrease in abort rate, which drops from 1.8 aborts per commit without delayed RMWs to 1.5 aborts per commit at eight threads. Unlike eager, here, aborts typically happen late in the transaction's execution, and every abort prevented is effectively another transaction committed.

5.4.3. Publication Safety. Up until this point, the implementations we tested did not include the modifications proposed in Section 4 and thus would not be correct if memcached were to use RMW operations in conjunction with variables used for publication

of previously private data. However, conducting the evaluation in this manner allows us to measure the best-case performance of delayed RMWs. We now look at the expected case behavior in which support for ELA publication safety is added.

While a valid approach would be for the programmer to use annotations instead of compiler analysis to select which RMWs to delay and then manually verify that those RMWs do not affect publication safety, we believe this to be too burdensome. Just as privatization safety requires reasoning about complex object lifecycles and is now a default feature of ELA and stronger semantics (and is required in C++), the appeal of publication safety is that it frees programmers from thinking about which variables participate in publication.

Figure 8(c) repeats the experiments from Figure 8(b) but adds additional bars to show the cost when publication safety is turned on. In memcached, statistics counters are never used for publication and, in fact, are never read by the same transactions that update them via RMWs. Consequently, modifying the algorithm to provide safe publication should not affect program behavior. We observe only a slight increase in execution time due to the increased logging overhead and its associated cache pollution. This cost is negligible in almost all cases.

6. RELATED WORK

6.1. Contention Management

Contention Management (CM) [Scherer III and Scott 2005; Yoo and Lee 2008; Dragojevic et al. 2008; Attiya and Milani 2009; Attiya et al. 2006] is the most popular approach to resolving conflicts among transactions. While the mechanisms vary greatly, at a fundamental level, CM aims to influence the scheduling of transactions to prevent conflicts. Simple nonblocking contention managers often incorporate backoff, such that when transactions T_A and T_B conflict, one will abort, wait briefly, and restart. Usually, this perturbation of the schedule suffices to prevent the conflict from manifesting again. Blocking approaches may instead explicitly deschedule one transaction (e.g., T_B) until the other (T_A) commits. At the extreme point, T_B may be rescheduled to run on the same processor as T_A to ensure that no conflict occurs and locality is maximized.

Whereas existing CM techniques are sufficient for ensuring forward progress, our work demonstrates that CM solutions are not necessarily optimal. By explicitly restarting one transaction, rather than reordering and coordinating conflicting operations, any CM approach should be expected to fail to scale in the face of highly contended variables. Although CM remains important for arbitrating transient conflicts in a manner that preserves the properties of the underlying TM, we believe our work shows that CM alone cannot guarantee optimal performance; explicitly reordering transactional accesses to eliminate conflicts seems necessary.

6.2. Conflict Detection

Similarly to CM, some TM implementations can vary their conflict detection mechanism such that some transactions use encounter-time locking, and others use commit-time locking [Ni et al. 2008]. Some implementations vary the locking mechanism on a per-variable basis [Sonmez et al. 2009], and others change the global choice of TM implementation based on the presence of high abort rates [Riegel et al. 2008; Lev et al. 2007; Wang et al. 2012]. Similar mechanisms have even been proposed for hardware TM [Shriraman et al. 2008]. In general, these approaches aim to prevent pathology: Upon repeated aborts due to conflicts, a transaction becomes pessimistic, locking locations eagerly in order to prevent concurrent transactions from interfering. However, as shown in our microbenchmarks, highly contended variables will still result in conflicts regardless of the conflict detection strategy employed by the TM. Thus, as

with CM, our recommendation is that these approaches be viewed as complementary to our work. When transactions experience pathology, or when false sharing is causing conflicts over ownership records, then increasing pessimism or changing algorithms can improve performance.

In a related manner, Zyulkyarov et al. [2010] developed tools for identifying contention in transactional programs and then used this information to rewrite code to decrease contention. We believe that the same analysis could guide the annotation of variables involved in RMW-based conflicts as an alternative to ad-hoc programmer techniques or the live-out analysis described in Section 3.

6.3. Nesting

Both open and closed nesting [Moss and Hosking 2005; Ni et al. 2007; Moravan et al. 2006] can improve performance in the face of highly contended variables. For example, when a hot counter is incremented in a closed nested transaction that comprises the tail of a long-running parent, then conflicts on the counter may require only the nested child transaction to restart. Although limited to the case where the hot counter is incremented at the end of the transaction, this approach still avoids much of the cost of aborts due to hot variables.

Similarly, if the hot counter was incremented via an open-nested transaction, then most conflicts on the counter simply would not manifest. With open nesting, the increment would occur immediately and become visible to concurrent threads. If the parent transaction subsequently aborted, then some programmer-specified compensating action would undo the increment. Although the infrastructure for supporting delayed operations resembles the infrastructure for registering undo actions, there is a fundamental difference: Our delayed operations do not break atomicity and thus can be invisible to the programmer. In contrast, open nesting often requires the programmer to employ ad-hoc abstract locking [Herlihy and Koskinen 2008] to prevent concurrent transactions from reading an incorrect counter value (e.g., if the counter is used to detect an empty collection). Additionally, our mechanism supports accesses to the counter after a delayed RMW, which can be difficult with open nesting [Ni et al. 2007; Agrawal et al. 2006].

An aggressive approach to closed nesting that preserves atomicity is to use Abstract Nested Transactions (ANTs) [Harris and Stipic 2007]. ANTs are similar to closed nested transactions except that, when they complete, they are not merged into the parent transaction. Should the parent later detect a conflict, then if the conflict is localized to an access performed within the ANT, the ANT can often be rolled back and re-executed without requiring the parent transaction to abort. This, of course, succeeds only when the parent does not read locations modified by its ANTs. Our work can be thought of as (a) demonstrating the value of ANTs, (b) providing a practical implementation of small ANTs for unmanaged languages, and (c) introducing the runtime mechanisms necessary for resolving parent accesses to values modified by its ANTs. Additionally, our work identifies and resolves questions related to semantics that had not yet been identified when ANTs were proposed.

6.4. Nonatomic Updates

As a last resort, operations on hot counters could, in some cases, be performed outside of transactions. For example, the Atomos language Carlstrom et al. [2006] and the TM proposed by Ni et al. [2008] and Carlstrom et al. [2006] both allow transactions to register “onCommit” functions whose execution is delayed until after the transaction commits. These functions do not execute within the context of the transaction, and their accesses to shared data must be manually synchronized. Depending on the implementation, they may be able to use transactions themselves. Clearly, such an approach is

not appropriate for the general case in which a transaction might read a hot variable after incrementing it. However, when precise counts are not required by the application logic, and when these hot variables are not used in other ways by the parent transaction, deferring their update until after commit via simple “onCommit” routines may be a viable alternative to the mechanisms proposed in this paper.

7. CONCLUSION AND FUTURE WORK

In this article, we introduced algorithms for delaying RMW operations in software and hardware transactional memory. Our mechanism employs static identification of candidate RMWs and then dynamic tracking to ensure that delaying an RMW to a location that is also read or written by the same transaction does not affect the correctness of the program. We also showed that for a large class of STM algorithms, delaying RMWs can break support for the “publication” pattern, but that a simple and low-overhead extension to our algorithms can restore publication safety.

Although experiments show that our technique can significantly improve performance, particularly for STM with commit-time locking, delaying RMWs until commit time does not change the fact that a memory location is being shared between two threads. Thus, although we believe our techniques can help to reduce aborts and improve performance, they are not a substitute for redesigning applications to avoid contention in the first place.

As future work, we intend to investigate mechanisms for preserving publication safety when delaying RMWs in STM algorithms that do not use ownership records. Of particular interest is the TLRW algorithm [Dice and Shavit 2010], which uses read locking. Although TLRW does not have versioned writes, we believe its readers/writer locks could be extended to support versions in a manner that would allow delayed RMWs and publication safety. An additional future direction is to evaluate delayed RMWs in STM systems that already offer ALA publication safety and thus do not require the extensions discussed in Section 4.

As a final research direction, we are exploring the extension of our technique from single-location RMWs to multilocation operations. This effort, which combines elements of Abstract Nested Transactions [Harris and Stipic 2007] and Safe Futures [Welc et al. 2005], will likely require extensive static analysis to predict the reads of the delayed operation so that any subsequent read that forces a promotion of the delayed operation can be identified at runtime.

ACKNOWLEDGMENTS

We would like to thank the TACO reviewers and editor, whose detailed suggestions greatly improved this article.

REFERENCES

- Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2008. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*.
- Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. 2006. Memory models for open-nested transactions. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.
- Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. 2006. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*.
- Hagit Attiya and Alessia Milani. 2009. Transactional scheduling for read-dominated workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*. Nimes, France.
- Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.

- Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. 2006. The Atomos transactional programming language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*.
- Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, and Sherman Yip. 2009. Rock: A high-performance sparse CMT processor. *IEEE Micro* 29, 2 (March–April 2009), 6–16.
- Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dan Grossman, and David Christie. 2010. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*.
- Luke Dalessandro, Michael Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*.
- Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*.
- 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*.
- Aleksandar Dragojevic, Rachid Guerraoui, Anmol Singh, and Vasu Singh. 2008. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*.
- Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.
- Rachid Guerraoui, Thomas Henzinger, Michal Kapalka, and Vasu Singh. 2010. Transactions in the jungle. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*.
- Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.
- Tim Harris and Srdjan Stipic. 2007. Abstract nested transactions. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*.
- Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.
- Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*.
- IBM. 2009–2010. Concurrent Building Blocks. Retrieved from <http://amino-cbbs.sourceforge.net/>.
- Intel Corp. 2012. *Intel 64 and IA-32 Architectures Software Developer's Manual* (325462-044us ed.). Intel Corp.
- Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*.
- Jeremy Manson, William Pugh, and Sarita Adve. 2005. The Java memory model. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*.
- Virendra J. Marathe, Michael Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.
- Jean-Philippe Martin, Christopher Rossbach, Derek Murray, and Michael Isard. 2013. Supporting efficient aggregation in a task-based STM. In *Proceedings of the 3rd Workshop on Systems for Future Multicore Architectures*.
- 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 ACM Symposium on Parallelism in Algorithms and Architectures*.
- Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*.
- Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*.
- Michelle Moravan, Jayaram Bobba, Kevin Moore, Luke Yen, Mark Hill, Ben Liblit, Michael Swift, and David Wood. 2006. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*.

- Eliot Moss and Antony L. Hosking. 2005. Nested transactional memory: Model and preliminary architecture sketches. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*. San Diego, CA.
- Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Rick Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proceedings of the 12th ACM Symposium on Principles and Practice of Parallel Programming*.
- Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowitz, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*.
- Martin Pohlack and Stephan Diestelhorst. 2011. From lightweight hardware transactional memory to lightweight lock elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*.
- Torvald Riegel, Christof Fetzer, and Pascal Felber. 2008. Automatic data partitioning in software transactional memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*.
- Wenjia Ruan, Yujie Liu, and Michael Spear. 2014a. STAMP need not be considered harmful. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*.
- Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 2014b. Transactionalizing legacy code: An experience report using GCC and memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*.
- William N. Scherer III and Michael L. Scott. 2005. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*.
- Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. 2008. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture*.
- Nehir Sonmez, Tim Harris, Adrián Cristal, Osman S. Unsal, and Mateo Valero. 2009. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*.
- Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2007. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*.
- Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. 2012. A transactional memory with automatic performance tuning. In *Proceedings of the 7th International Conference on High-Performance and Embedded Architectures and Compilers*.
- Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe futures for Java. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.
- Richard Yoo, Christopher Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel transactional synchronization extensions for high performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
- Richard Yoo and Hsien-Hsin Lee. 2008. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*.
- Richard Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin Lee. 2008. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*.
- Ferad Zylkyarov, Srdjan Stipic, Tim Harris, Osman Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. 2010. Discovering and understanding performance bottlenecks in transactional applications. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*.

Received June 2014; revised November 2014; accepted November 2014