A Scalable Lock-Free Universal Construction with Best Effort Transactional Hardware

Francois Carouge and Michael Spear

Lehigh University
{frc309, spear}@cse.lehigh.edu

Abstract. The imminent arrival of best-effort transactional hardware has spurred new interest in the construction of nonblocking data structures, such as those that require atomic updates to k words of memory (for some small value of k). Since transactional memory itself (TM) was originally proposed as a *universal* construction for crafting scalable lock-free data structures, we explore the possibility of using this emerging transactional hardware to implement a scalable, unbounded transactional memory that is simultaneously nonblocking and compatible with strong language-level semantics. Our results show that it is possible to use this new hardware to build nonblocking TM systems that perform as well as their blocking counterparts. We also find that while the construction of a lock-free TM is possible, correctness arguments are complicated by the many caveats and corner cases that are built into current transactional hardware proposals.

1 Introduction

Transactional Memory (TM) [12] was initially proposed as a hardware mechanism to support lock-free programming. The subsequent development of Software Transactional Memory (STM) [21] continued this focus on nonblocking progress. Indeed, it was two nonblocking algorithms, the obstruction-free DSTM [11] and lock-free FSTM [10] that reinvigorated STM research in the past decade. However, as processors hit the heat wall and vendors embraced parallelism across the entire spectrum of products, the urgent need for low-latency STM algorithms led to a flurry of research into lock-based designs. Roughly speaking, lock-based STM algorithms are easier to design and implement, and with careful design (and OS support) are resilient to preemption [4] and priority inversion. Some lock-based STM algorithms are even livelock-free [3, 20, 25]. These results have not stopped exploration into nonblocking STM, and two recent obstruction-free STM designs show how a blocking system can be made nonblocking [16, 27] without much overhead.

The prospect of integrating STM into high-level languages presents a further complication. If data is to be used both transactionally and nontransactionally, then an STM implementation may introduce races between active or committing transactions and concurrent nontransactional code [18, 24, 28]. These races are implementation artifacts; they affect race-free code because the STM implementation does not provide strong correctness guarantees. While some STM designs are privatization-safe [3, 6, 15, 17, 25], solutions to the "privatization" problem appear to require blocking, either at transaction boundaries [17–19, 24], or during the execution of transactional [6, 15], or even nontransactional [22] code.

Two recent innovations provide hope that these obstacles can be overcome. First, "single-CAS" STM (SCSTM) algorithms [3,20,25] show that it is possible to construct scalable lock-based systems that are privatization-safe, livelock-free, and whose blocking behavior is both localized and easy to characterize. Secondly, interest in bounded or "best-effort" hardware support for transactional memory (BETM) promises to extend common instruction-set architectures with features to facilitate lock-free programming. It has been shown that a simple hardware extension called alert-on-update (AOU) [26] can make a traditional (weak semantics) STM implementation obstruction-free. However, the AOU proposal ignored implementation details, whereas BETM proposals do not. Given the weaker guarantees of BETM, we explore the creation of *lock-free* STM implementations that are scalable and provide strong semantics.

Our results are mixed. While using BETM is relatively easy, arguing about the correctness of a BETM algorithm is more difficult, as it requires knowledge of the underlying hardware, careful programming to avoid corner cases, and a mix of transactional and nontransactional accesses to metadata and program data. Assuming that we have not missed any causes of spurious aborts in the BETM systems we considered, we have made three SCSTM algorithms obstruction-free or lock-free without sacrificing performance or safety. The result is strong: using very little of what BETM offers, we can build algorithms that are lock-free, provide strong semantics, and scale well.

This paper is organized as follows. In Sections 2 and 3 we describe the hardware and software foundations of our new lock-free construction. We then present our lock-free construction in 4. In Section 5, we evaluate our nonblocking algorithms through simulation. We conclude in Section 6.

2 Background: Best Effort Transactional Hardware

Recent BETM systems from Sun Microsystems (the "Rock" processor [5]) and AMD (the "advanced synchronization facility" proposal [7], or ASF) embody a significant compromise: whereas previous HTM systems attempt to transparently support long-running transactions with large memory footprints, these best-effort systems provide limited guarantees. In exchange, BETMs are implementable in modern instruction set architectures. We briefly summarize the key attributes below:

- **Capacity:** BETMs limit the amount of data that can be accessed within a transaction. In ASF, transactions that speculatively access four locations or fewer will not fail due to capacity (the upper bound is not specified); in Rock, the L1 cache manages speculative loads, and a 32-entry store queue bounds the number of speculative writes. A software fallback is required if these resources are insufficient.
- **Mixed-Mode Access:** A transaction can execute "nonspeculative" loads and stores to lines that have not been accessed speculatively by the same transaction. These accesses can cause concurrent transactions in other processors to abort, but do not result in conflicts that cause the caller to abort. In ASF, nonspeculative stores occur immediately, which slightly alters the processor memory model. In Rock, nonspeculative stores are buffered in the 32-entry store queue, and do not occur until the transaction commits or aborts.
- **Progress:** BETM detects conflicts at the granularity of cache lines, using the "requester wins" model. Since "requester wins" resolves conflicts in favor of the most recent access to a location, it is prone to livelock [1]: from the moment transaction T speculatively accesses line X until it commits, any concurrent incompatible access from another processor will cause T to abort.
- **Forbidden Instructions:** BETM forbids certain instructions within a transaction. System calls are never allowed, and Rock also forbids floating point divide and certain control flow patterns.
- **Forbidden Events:** BETM transactions cannot survive a context switch. In addition, hardware exceptions and asynchronous interrupts cause the active transaction to abort. Particularly problematic are TLB misses and page faults. In Rock, some TLB misses can cause a hardware transaction to abort, and the TLB miss may need to be triggered outside of the transactional region before retrying. Page faults also cause hardware transactions to abort, with Rock requiring the fault to be re-created outside of the transaction before retrying.

To summarize, BETM transactions are limited in terms of the number of locations they can access speculatively, the number of locations they can access nonspeculatively, and the instructions they can execute. Interrupts and exceptions cause transactions to abort, and it is likely that there are undocumented reasons why a transaction in isolation may spuriously abort. Transactions that perform speculative stores are prone to livelock, and in general some form of contention management or software fallback is recommended. Nonetheless, we will show that BETM can be used to create a scalable, nonblocking, privatization-safe, unbounded STM.

3 Background: Single-CAS STM

The software foundation for our lock-free construction is the concept of Single-CAS STM (SCSTM). Example implementations of SCSTM include JudoSTM [20], RingSTM [25], and TMLLazy/NOrec [3]. These blocking algorithms are characterized by (1) livelock-freedom, (2) strong semantics, and (3) a simple linearizability condition [13].

3.1 SCSTM Behavior

Figure 1 presents a template for SCSTM algorithms. Specific SCSTM algorithms can be thought of as instantiations of this code using different metadata for conflict detection. The general behavior of SCSTM algorithms is very simple: concurrent updates are mediated via a single monotonically increasing counter. This counter serves both as a lock that protects shared memory when a transaction is finalizing its writes, and as a flag that is polled by in-progress transactions to determine when it is necessary to perform intermediate correctness checks ("validation").

Transactions buffer all updates to shared memory, and then commit by incrementing the counter to odd, optionally publishing some metadata describing the changes they are making to memory, committing their writes to memory, and then incrementing the counter again to even. Every read operation first checks for a pending local write, and then if no such write exists, performs a read from main memory. After reads, the global counter is polled to check if any new transactions have committed, possibly invalidating this transaction. If any are found, the transaction validates its entire read set and aborts if any reads have been invalidated by another transaction's commit.

A transaction only blocks when another transaction is in the process of committing. When writing transaction T is ready to commit (line 9 of `TMEnd`), its `my_ts` field stores the even value of the global counter (`global_ts`) at the last time that T performed a validation. Clearly, at that time T's reads and writes were all valid, or else T would have aborted. If T can atomically increment `global_ts` from `my_ts` to one greater, then it is sure no other writer transaction R committed in the interim. For R to

```
TMBegin:
  create_checkpoint()
  // wait if global lock is held
  while ((my_ts = global_ts) & 1)
    spin()

TMReadWord(addr):
  // check for buffered write
  if (writes.contains(addr))
    return writes.lookup(addr)
  // read from shared memory
  tmp = *addr
  // check for new commits...
  // if found, validate, then
  // re-read location
  while (my_ts != global_ts)
    TMValidate()
    tmp = *addr
  // log address for future
  // validation
  readset.add(addr, tmp)
  return tmp

TMWriteWord(addr, val):
  // record store in private buffer
  writes.insert(addr, val)
  // log address so it's easy to
  // announce set of modifications
  // at commit time
  writeset.add(addr)

TMAbort():
  reset_per_thread_metadata()
  restore_checkpoint()
```

```
TMEnd:
  // read-only fastpath
  if (writes.empty())
    reset_per_thread_metadata();
    return
  // acquire global lock before
  // committing.  Only attempt
  // to get lock when tx is
  // known to be in a valid state
  while (!CAS(global_ts, my_ts, my_ts+1)
    TMValidate()
  // announce set of changes
  publish_writeset()
  // update main memory
  for <addr, val> in writes
    *addr = val
  // release lock
  global_ts++
  reset_per_thread_metadata();

TMValidate():
  while (true)
    // wait if writeback is in progress
    tmp = global_ts
    if (tmp & 1)
      spin()
      continue
    // ensure reads still valid
    if (!all_reads_still_valid())
      TMAbort()
    // must re-validate if another
    // transaction committed
    if global_ts == tmp
      my_ts = tmp
      return
```

Fig. 1. Generic pseudocode for Single-CAS STM algorithms.

have committed first, it would have had to increment `global_ts`, which would have ensured that T's increment of `global_ts` would fail. If T cannot successfully increment `global_ts`, then it re-validates (and thus updates its local `my_ts`) before trying again.

The code in Figure 1 is general enough to describe NOrec, RingSTM, and TMLLazy: In TMLLazy, no `readset` or `writeset` is maintained, and calls to `TMValidate` are be replaced with calls to `TMAbort`. In RingSTM, the `writeset` is a signature that is published by appending it to a global list, and `TMValidate` intersects the per-thread `readset` (also a signature) with new entries on the list. In NOrec, no `writeset` is maintained, so `TMValidate` iterates through the $\{$address, value$\}$ pairs in the `readset`, making sure that each address still holds the expected value.

3.2 SCSTM Properties

In SCSTM, an in-flight transaction only blocks when another transaction is performing writeback (lines 11–15 of `TMEnd`). During writeback, reads and commits cannot be guaranteed an atomic view of metadata or shared memory, and thus threads wait until the lock is released (`TMBegin` lines 3–4, and `TMValidate` lines 3–6, called from `TMReadWord` and `TMEnd`).

By virtue of its single lock and commit-time validation protocol, SCSTM is livelock-free. The argument for livelock freedom in SCSTM is simple: a transaction only aborts if it fails a validation, and it only validates when the global counter is updated. Since the global counter is only updated when a transaction commits, the chain of events leading to an abort always begins with a transaction committing [25].

The single lock also serves to completely serialize the commit phase of writing transactions. On architectures with nonfaulting loads, the combination of polling on every read, validating whenever *any* transaction commits, and completely serializing writeback leads to privatization safety [24]. When nonfaulting loads are unavailable, privatization safety requires some additional measures to handle OS reclamation of memory in the middle of a concurrent `TMReadWord` operation [8, 14]. Thus SCSTM provides at least "ELA" semantics in the taxonomy of Menon et al. [18].

4 The Lock-Free Transformation

We next make SCSTM nonblocking using BETM. Our approach resembles that proposed by Spear et al. [26], where Alert-On-Update (AOU) was used to interrupt a thread while it held the lock protecting an idempotent critical section. The main differences in our technique center around the use of BETM. Specifically, previous work assumed an idealized TM in which the only cause of transactional aborts is transaction conflicts. We consider all published causes of BETM aborts, which makes our construction considerably more complex, but also realistic.

4.1 Making Stealing Safe

Our goal is to allow multiple threads to perform the same writeback operation simultaneously, and interrupt any partial writebacks when one thread completes the full writeback. This means that after a transaction has made `global_ts` odd on `TMEnd` line 9, any thread can perform the rest of the commit operation on that transaction's behalf.

To make writeback stealable, we must first make the list of pending `writes` visible. There are two means to provide this; either we can superimpose a pointer over the `global_ts` variable, such that an odd least significant bit indicates that the remaining bits are a pointer to the `writeset` (and any other needed metadata, such as "`next_ts`", the value to which `global_ts` must be set after writeback), or else we can use a hardware transaction to perform a multiword store. We use the former option for NOrec and TMLLazy, since `TMEnd` line 12 is a no-op, and the latter for RingSTM, since line 12 of `TMEnd` must be atomic with line 9.

Secondly, we must ensure that the result of multiple threads performing overlapping portions of the writeback simultaneously remains correct. In Spear's AOU work, writeback was done on a per-object basis, with at most one thread performing writes to an object at any time [26]. Since in SCSTM the entire write set is protected by a single counter, such an approach does not make sense for large write sets. Instead, we use a hash-based write log [20] to ensure that every address in the log is unique. We also require that no two addresses in the log overlap. This property is provided differently for type-safe and non-type-safe languages.

With these requirements in place, the writeback operation can be done in any order, and any number of {address, value} pairs in the write log can be written back multiple times, so long as (a) every pair is written back at least once and (b) as soon as one thread increments `global_ts` to even (`TMEnd` line 17), all helping immediately stops. If all writeback were performed using BETM speculative stores, this second property would be simple. However, with "requester wins" conflict detection, using BETM for writeback would rapidly lead to livelock. Instead, we use BETM to monitor to the `global_ts` and immediately interrupt a writeback, and then perform the writeback using nonspeculative stores.

Figure 2 presents the lock-free writeback code, assuming that when odd, `global_ts` represents a pointer to all needed metadata. To make SCSTM nonblocking, we replace the `spin()` codes (`TMBegin` line 4; line 5 of `TMValidate`) with calls to `TMStealWriteBack`, and replace `TMEnd` lines 14–15 with a call to `TMStealWriteBack`. The `HW_LIMIT` field addresses capacity constraints, and is discussed in the next section.

The use of `TMStealWriteBack` preserves privatization safety, livelock-freedom, and correctness. The argument is straightforward: in SCSTM, these properties are all preserved by the following pair of invariants: *an in-flight transaction never performs a read of shared data while write-back is in progress* and *the commit of writing transactions is serialized.* In SCSTM, these invariants are preserved via blocking. In our constructions,

```
TMStealWriteBack:
 1  // start BETM transaction
 2  BETM_BEGIN
 3  // if global_ts changes, this
 4  // thread returns to line 2
 5  BETM_LOAD(x, global_ts)
 6  // writeback may be finished
 7  if ((x & 1) == 0)
 8    BETM_COMMIT
 9    return
10  // global_ts is a pointer
11  metadata = x & ~1
12  // metadata->done_already estimates
13  // how much writeback is complete
14  c = metadata->done_already
15  // do writeback, starting at c
16  for <a, v> in metadata->writeset[c..end]
17    *a = v
18    if ((++c % HW_LIMIT) == 0)
19      metadata->done_already = c
20  // compute new global_ts value
21  n = metadata->next_ts
22  BETM_COMMIT
23  // nontransactionally update global_ts
24  // with CAS.  must use counted pointer
25  CAS(global_ts, x, n + 2)
```

Fig. 2. Lock-free writeback.

the invariants still hold, but the blocking is replaced by assisting with write-back.

4.2 Ensuring Progress via HW_LIMIT

In Figure 2, the entire writeback operation is performed using nonspeculative stores issued from within a transaction. By using nonspeculative stores, performing writeback simultaneously in multiple threads will not prevent progress. However, BETM transactions have limits that must be managed. Specifically, they can access a limited amount of memory and they cannot survive exceptions or interrupts. Our approach is to break the writeback into small pieces, and then use the `done_already` field to track how much of the writeback is complete. The `HW_LIMIT` field dictates how many writes are performed before updating `done_already`. We discuss constraints on its size below:

First, since BETM transactions cannot execute for longer than a quantum, we must not attempt more writes than can be reasonably finished in that time. Of greater practical concern, in virtualized environments with high I/O interrupt frequencies, `HW_LIMIT` must be small enough that a bundle of writes can be completed between interrupts.

The next danger occurs when the speculative region performs an instruction that causes an abort; if there is no mechanism to prevent the offending instruction from re-occurring when the region is restarted, then an infinite loop can occur. In a BETM such as Rock, where page faults and some TLB misses can cause aborts that must be handled by the user code, an abort handler will be required to update the TLB or simulate a page fault in nontransactional code. In ASF, where a page-fault will be handled automatically before the region restarts, this is not a concern. Additionally, in both systems the speculative region must not perform prohibited instructions, such as division, far calls, system calls, and some other function calls. This condition clearly holds in our code, which only performs loads, stores, and read-only traversal of a data structure.

Lastly, BETM may limit nonspeculative writes within a transaction. On Rock, stores may be issued to at most 32 locations from within a hardware transaction. Thus if the entries in the `writeset` map to more than 31 locations (we reserve one for `done_already`), the speculative region is guaranteed to abort. In addition to the store buffer, the TLB introduces limitations. Suppose that a processor has T entries in its TLB. If each variable in the `writeset` is located on a different page of memory, then if the TLB capacity is fewer than 31 entries, a tighter bound is required if TLB misses can cause aborts. In the worst case, if the instruction and data caches share a TLB, and every interesting memory element with a size greater than a single word happens to span a page boundary, the following restrictions would apply (note that we assume the page size is larger than the word size):

– Two TLB entries must be reserved for the code pages holding the `TMStealWriteBack` function.
– Two TLB entries must be reserved for the data pages holding the stack.
– The data pages holding `writeset` portion being written-back require two TLB entries.
– One TLB entry must be reserved for the data page holding the metadata object.

Thus if we attempt to write to more than $T - 7$ pages, the region may abort due to a TLB miss, and repeated attempts to execute the region will continue to abort due to TLB misses (caused by writes to different pages). If we generalize the size of the store buffer to a constant `MAX_STORES`, then the total number of stores that can be guaranteed to succeed is `HW_LIMIT = min(T-7, MAX_STORES)`. In practice, however, if multiple updates are performed to the same page and/or cache line, this limit may be exceeded without aborting the region.[1]

To overcome this limitation without being overly conservative, we use a lazy counter, `done_already`. When a transaction logically commits (the CAS on line 9 of `TMEnd` succeeds), it installs a pointer that indicates both

[1] These assumptions are conservative. Modern machines have large pages and separate instruction/data TLBs. Explicit alignment of data structures can decrease the number of reserved TLB entries from 7 to as low as 2.

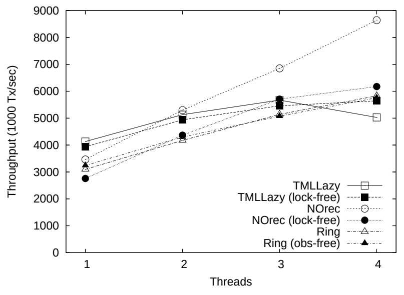
the `writeset` and a count of the number of `writeset` entries that have already been written back. `TMStealWriteBack` updates this count with its local progress after every `HW_LIMIT` stores. If a thread is executing `TMStealWriteBack` in isolation and its writeback aborts due to a hardware limitation, when the region restarts it resumes writeback at the last value of `done_already`. This ensures progress and avoids infinite loops, since at least `HW_LIMIT` stores were performed before the abort, and will not be performed again.

If `TMStealWriteBack` is executed by multiple threads, it is possible for the value of `done_already` to decrease. However, we observe that once all threads observe the value v in `done_already`, then the first v elements will not be written again, and that `done_already` will only hold values greater than v from that point until one thread completes writeback. Thus the writeback will ultimately succeed, though the worst-case bound completion time depends both on the number of threads (T) and the size of the `writeset` (W). This bound could be improved by making the value of c persist from one writeback attempt to the next, and updating it to `max(c, metadata->done_already)` within each attempt. Doing so would require additional common-case code to ensure that the value of x has not changed; for example, if thread 1 performs K writes, then delays and another thread both completes the writeback and installs a new `writeset` to be written back, then thread 1 cannot resume at the Kth element.

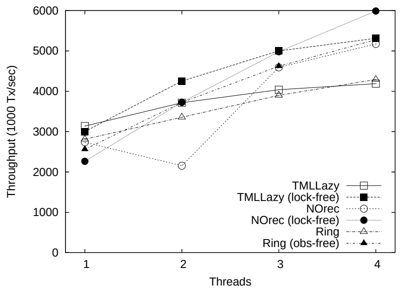
There are two other properties that our approach provides. First, existing limits on `HW_LIMIT` ensure that context switches during writeback will not force the entire writeback to restart; this is particularly important if writeback of an extremely large dataset could take longer than a quantum to complete. Second, the writeback could be parallelized by partitioning the `writeset` and protecting each partition with a different lock [9]. As long as a thread's call to `TMStealWriteBack` does not return until *all* partitions of the writeback have been performed, the behavior is indistinguishable from that of a single-lock writeback, but with the added benefit of parallel writeback. In the limit case, we could set the partition size to `HW_LIMIT`, and simplify the code significantly. We do not evaluate such an implementation in this paper.

4.3 ABA Safety

The code in Figure 2 can admit an "ABA" problem on line 25. Specifically, when `global_ts` equals some value V and two threads T_1 and T_2 both execute `TMStealWriteBack`, it is possible that T_1 pauses between lines 22 and 25. It has done *all* of the writeback, but it has not halted



(a) RBTree (no blocking)



(b) RBTree (with blocking)

Fig. 3. RBTree workloads with 8-bit keys and an equal insert/lookup/remove ratio.

writeback in any helper threads (such as T_2). If T_2 then also performs the writeback, executes line 25, and then commits a new transaction that re-uses the same location as its "metadata" object, then when T_1 awakes and issues its CAS, it can "clean up" from an in-flight writeback that it has mistaken for the writeback it previously performed. We avoid this problem by attaching a counter to the `global_ts` pointer and using a wide CAS. Techniques using garbage collection are equally applicable. However, the use of a hardware transaction does not would only provide obstruction-freedom.

Since RingSTM requires a multiword atomic update in order to commit, we are already limited to producing an obstruction-free version since a BETM-based nonblocking multiword update is only obstruction-free. That being the case, when making RingSTM nonblocking we make the writeback obstruction-free, by removing lines 23–25 and inserting the statement `BETM_STORE(global_ts, n+2)` before line 22.

5 Evaluation

We evaluate our algorithms using the ASF extensions to PTLSim [29]. PTLSim faithfully models the AMD64 instruction set architecture, with extensions to support ASF. Previous studies have found PTLSim to be sufficiently accurate when compared to real hardware [2].

For this work, we made TMLLazy and NOrec lock-free. Since we use a writing hardware transaction to achieve atomic multiword updates in RingSTM, it is only obstruction-free. Our codes do not use the parallel writeback optimization discussed in Section 4.2, and our algorithms use a hard-coded value of 16 for `HW_LIMIT`. Our RingSTM, NOrec, and TML-Lazy codes are based on their published implementations, which are more optimized than the pseudocode in Figure 1.

We first evaluate the cost of our nonblocking algorithms in comparison to their nonblocking counterparts. Figure 3(a) depicts a Red-Black Tree experiment, using 8-bit keys. In each trial, we perform 100K transactions per thread, using an equal mix of inserts, lookups, and deletes. The maximum number of locations written by a single transaction is 33. Our first finding, which we do not show, is that it is extremely easy to livelock hardware transactions that perform writes. In Ring (obs-free), we used a writing hardware transaction to commit transactions. Initially we had each thread execute a fixed no-op loop whenever its hardware transaction aborted. At three threads this lead to two orders of magnitude slowdown; the hardware transaction to effect a commit would abort 98K times per 67 transactions. To avoid this behavior, we added exponential backoff. Note that backoff was never needed for stealable writeback.

Regarding performance, we see that the cost of lock freedom in TML-Lazy is negligible, as is the cost of obstruction freedom in RingSTM. For NOrec, the cost of lock freedom seems higher. We attribute this to two causes: first, we note that NOrec's scalability on the simulator is much better than previously reported for this workload [3,23]. Secondly, we observed that NOrec steals writeback much more frequently than the other systems. In Ring, transactions validate before helping, and in TMLLazy they abort and roll back before helping; in NOrec they must help with writeback before validating, since validation uses the actual values instead of the value of `global_ts` (TMLLazy) or signature (RingSTM). We conjecture that making stealing less aggressive, by introducing a small delay before attempting to steal writeback, would mitigate this effect.

Secondly, we consider the impact of preemption. For these tests, we introduced a delay between lines 12 and 13 of `TMEnd`. The delay occurs on randomly selected writing transactions (roughly once per 32 commits), and consists of a loop executing 8K no-ops. This simulates the effect of preemption during the write-back phase. In our blocking implementations, this delay prevents any progress in any other thread, since the committing thread holds the lock; we expect dampened scalability. For nonblocking implementations, a concurrent transaction should not block when the committer experiences a delay; instead, the concurrent transaction should steal the writeback and then continue.

Figure 3(b) depicts the behavior for our algorithms with and without the nonblocking transformation. The fact of delays in the commit protocol immediately results in lower single-thread throughput, and the artificial blocking also substantially limits the scalability of the lock-based STM implementations. By stealing writeback, our nonblocking algorithms overcome fact that some transactions block while holding locks. This, in turn, leads to higher throughput at all thread levels for the nonblocking code, when compared to its lock-based counterpart.

6 Conclusions and Future Work

In this paper we considered the construction of lock-free STM, and showed how to make single-CAS STM obstruction-free (and in some cases lock-free) through the use of simple best-effort transactional hardware (BETM). Our results indicate that the cost of nonblocking progress guarantees is negligible, and the nonblocking systems are immune to pathologies such as preemption and livelock. Our algorithms provide the first scalable, lock-free TM implementations with strong semantics.

We hope that this research will encourage further consideration of how much must be guaranteed to make BETM useful. In our algorithm, transactions executing only one speculative load, no speculative stores, and 16 nonspeculative stores, were sufficient to build a privatization-safe lock-free STM. Looking forward, questions include:

1. In virtualized or I/O-intensive environments, will parameters like `HW_LIMIT` need to be dynamically adjusted?
2. Do aborted BETM transactions know immediately that they have aborted, or can they continue to execute nonspeculative stores for any duration after their abort? What impact would such "zombie" transactions or delayed stores have on our algorithms?
3. What unspecified sources of aborts can be overcome in practice? Are there any undocumented sources of aborts (we imagine branch misprediction is a possibility) that would break our progress guarantees?

Additionally, we hope that successive refinements of BETM will improve ease-of-use and performance. Clearly support for both speculative and nontransactional accesses is critical to progress. In our opinion, a platform in which very few locations can be accessed, but transactions are guaranteed to succeed if they are retried enough times in succession, is very appealing.

Acknowledgments: We thank our reviewers for many helpful suggestions, and Victor Luchangco for his advice during the preparation of our final manuscript. Stephan Diestelhorst, Dave Dice, and Hagit Attiya gave feedback on early drafts of this paper. We are also grateful to Martin

Pohlack and Stephan Diestelhorst for their continued assistance with the ASF toolchain, and Dave Dice for his help understanding the Rock processor. We also thank Michael Scott, Luke Dalessandro, and Torvald Riegel.

References

1. Bobba, J., Moore, K., Volos, H., Yen, L., Hill, M., Swift, M., Wood, D.: Performance Pathologies in Hardware Transactional Memory. In: Proc. of the 34th Intl. Symp. on Computer Architecture. San Diego, CA (Jun 2007)
2. Christie, D., Chung, J.W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Riviere, E.: Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In: Proc. of the EuroSys2010 Conf. Paris, France (Apr 2010)
3. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Bangalore, India (Jan 2010)
4. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proc. of the 20th Intl. Symp. on Distributed Computing. Stockholm, Sweden (Sep 2006)
5. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early Experience with a Commercial Hardware Transactional Memory Implementation. In: Proc. of the 14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. Washington, DC (Mar 2009)
6. Dice, D., Shavit, N.: TLRW: Return of the Read-Write Lock. In: Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing. Raleigh, NC (Feb 2009)
7. Diestelhorst, S., Hohmuth, M.: Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory. In: Proc. of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods. Boston, MA (Apr 2008)
8. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Salt Lake City, UT (Feb 2008)
9. Fernandes, S., Cachopo, J.: A Scalable and Efficient Commit Algorithm for the JVSTM. In: Proc. of the 5th ACM SIGPLAN Workshop on Transactional Computing. Paris, France (Apr 2010)
10. Fraser, K., Harris, T.: Concurrent Programming Without Locks. ACM Trans. on Computer Systems 25(2) (2007)
11. Herlihy, M.P., Luchangco, V., Moir, M., Scherer III, W.N.: Software Transactional Memory for Dynamic-sized Data Structures. In: Proc. of the 22nd ACM Symp. on Principles of Distributed Computing. Boston, MA (Jul 2003)
12. Herlihy, M.P., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: Proc. of the 20th Intl. Symp. on Computer Architecture. San Diego, CA (May 1993)
13. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Trans. on Prog. Languages and Systems 12(3), 463–492 (1990)
14. Hudson, R.L., Saha, B., Adl-Tabatabai, A.R., Hertzberg, B.: A Scalable Transactional Memory Allocator. In: Proc. of the 2006 Intl. Symp. on Memory Management. Ottawa, ON, Canada (Jun 2006)

15. Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a Scalable Software Transactional Memory. In: Proc. of the 4th ACM SIGPLAN Workshop on Transactional Computing. Raleigh, NC (Feb 2009)
16. Marathe, V., Moir, M.: Toward High Performance Nonblocking Software Transactional Memory . In: Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. Salt Lake City, UT (Feb 2008)
17. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable Techniques for Transparent Privatization in Software Transactional Memory. In: Proc. of the 37th Intl. Conf. on Parallel Processing. Portland, OR (Sep 2008)
18. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures. Munich, Germany (Jun 2008)
19. Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and Implementation of Transactional Constructs for C/C++. In: Proc. of the 23rd ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications. Nashville, TN, USA (Oct 2008)
20. Olszewski, M., Cutler, J., Steffan, J.G.: JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In: Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques. Brasov, Romania (Sep 2007)
21. Shavit, N., Touitou, D.: Software Transactional Memory. In: Proc. of the 14th ACM Symp. on Principles of Distributed Computing. Ottawa, ON, Canada (Aug 1995)
22. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K., Saha, B.: Enforcing Isolation and Ordering in STM. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Diego, CA (Jun 2007)
23. Spear, M.: Lightweight, Robust Adaptivity for Software Transactional Memory. In: Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures. Santorini, Greece (Jun 2010)
24. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: Proc. of the 12th Intl. Conf. On Principles Of DIstributed Systems. Luxor, Egypt (Dec 2008)
25. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: Scalable Transactions with a Single Atomic Instruction. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures. Munich, Germany (Jun 2008)
26. Spear, M.F., Shriraman, A., Dalessandro, L., Dwarkadas, S., Scott, M.L.: Nonblocking Transactions Without Indirection Using Alert-on-Update. In: Proc. of the 19th ACM Symp. on Parallelism in Algorithms and Architectures. San Diego, CA (Jun 2007)
27. Tabba, F., Wang, C., Goodman, J.R., Moir, M.: NZTM: Nonblocking Zero-Indirection Transactional Memory. In: Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing. Portland, OR (Aug 2007)
28. Wang, C., Chen, W.Y., Wu, Y., Saha, B., Adl-Tabatabai, A.R.: Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In: Proc. of the 2007 Intl. Symp. on Code Generation and Optimization. San Jose, CA (Mar 2007)
29. Yourst, M.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In: Proc. of the 2007 IEEE Intl. Symp. on Performance Analysis of Systems and Software. San Jose, CA (Apr 2007)