Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters

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

Time-based transactional memories typically rely on a shared memory counter to ensure consistency. Unfortunately, such a counter can become a bottleneck. In this article, we identify properties of hardware cycle counters that allow their use in place of a shared memory counter. We then devise algorithms that exploit the x86 cycle counter to enable bottleneck-free transactional memory runtime systems. We also consider the impact of privatization safety and hardware ordering constraints on the correctness, performance, and generality of our algorithms.

Categories and Subject Descriptors: [**Computing Methodologies**]: Shared Memory Algorithms; [**Computer Systems Organization**]—*Multicore architectures*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Transactional memory, privatization, `rdtscp`, counters

ACM Reference Format:
Ruan, W., Liu, Y., and Spear, M. 2013. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. ACM Trans. Architec. Code Optim. 10, 4, Article 40 (December 2013), 21 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555297

1. INTRODUCTION

Most high-performance Software Transactional Memory (STM) [Shavit and Touitou 1995] implementations reduce the common-case overhead of validation by using timestamps. The technique, which was first employed in the Lazy Snapshot Algorithm (LSA) [Riegel et al. 2006] and TL2 [Dice et al. 2006] algorithms, is straightforward: Every writer transaction increments a global clock during its commit phase, and writes the resulting value into every lock that it releases. All transactions read the clock when they begin, and whenever reading a new location, they check if the corresponding lock stores a clock value that is less than this start time; if so, the location can be read without validation. In this manner, the costly quadratic validation overheads of previous systems [Fraser 2003; Herlihy et al. 2003; Marathe et al. 2005, 2006] can be avoided. Since 2006, virtually every single-version STM that uses ownership records has employed a global shared counter [Dragojevic et al. 2009; Felber et al. 2008; Marathe and Moir 2008; Menon et al. 2008; Spear et al. 2009; Wang et al. 2007; Zhang et al. 2008].

There are two problems with global shared clocks. First, clock-based techniques for avoiding validation are heuristic. In the worst case, a clock-based STM might still validate the entire read set on every read, resulting in validation overhead quadratic in the

New Article, Not an Extension of a Conference Paper. A preliminary version of this work appeared at the TRANSACT 2013 workshop, which does not have archival proceedings. This work was supported in part by the National Science Foundation through grants CNS-1016828 and CCF-1218530.
Author's addresses: W. Ruan, Y. Liu, and M. Spear, Computer Science and Engineering Department, Lehigh University, Bethlehem PA 18015, USA.

DOI: http://dx.doi.org/10.1145/2555289.2555297

number of locations accessed by the transaction. Second, the use of a shared memory counter as the clock can become a scalability bottleneck. Since every writer transaction must increment the counter during its commit operation, workloads consisting of frequent small writing transactions experience considerable cache invalidation traffic as the counter moves among processors' caches.

The open-source release [Minh et al. 2008] of the TL2 algorithm [Dice et al. 2006] included heuristics for reducing the overhead of counter increments. The main observation was that that timestamp-based STM does not require a strictly increasing counter; monotonicity suffices. Thus, if a `Compare-And-Swap` (`CAS`) fails to increment a counter, then the return value of the `CAS` can be used in place of a new value. Of course, this technique is itself a heuristic, and while it lessens the impact of contention over the shared counter, scalability problems can still remain for small, frequent writer transactions. A second technique pioneered by TL2 was to skip counter increments with some probability, instead using a value one larger than the current counter value as the commit time. However, this technique is effective only if successive transactions rarely modify the same data.

An alternative to shared counters, first proposed by Riegel et al. [2007], is to use the multimedia timer present in some systems in place of a shared memory clock. Riegel's system used the real-time MMTimer built into Altix machines. This hardware timer is a read-only device, and thus concurrent accesses by multiple processors do not create contention. However, as an off-chip hardware component, the MMTimer operates at a considerably slower frequency than a processor core. Consequently, Riegel's STM needed to manually address clock skew and compensate for the clock's low frequency.

In this article, we explore whether an STM algorithm can be built upon existing in-core timing hardware, rather than an external (hardware or shared memory) clock. Modern processors expose a user-mode accessible "tick" counter, which returns the number of processor cycles which have passed since boot time. The details of these counters vary among Instruction Set Architectures (ISAs) and even microarchitectures, and we focus on the `rdtsc` family of instructions in the x86 ISA. As appropriate, we built STM systems that employed the processor tick count in place of a shared memory clock. Our primary findings are that (a) there are memory fence and ordering requirements that must be enforced when using these counters to implement an STM, and (b) the use of hardware clocks to accelerate STM is effective for STM libraries that do not offer privatization safety, but less effective for libraries that are privatization safe.

The remainder of this article is organized as follows: Section 2 describes the behavior of software clocks in STM implementations. It then discusses hardware cycle counter properties on the x86 and SPARC ISAs, and identifies potential pitfalls when using these counters in place of a shared memory clock. Section 3 extends STM algorithms so that they can employ the x86 `rdtscp` in place of software clocks. Section 4 considers techniques for making these `rdtscp`-based algorithms privatization safe. Section 5 evaluates our algorithms on single and dual-chip x86 systems, and Section 6 concludes.

2. HARDWARE AND SOFTWARE CLOCKS

Some manner of global clock object is at the heart of nearly every STM algorithm proposed during the last decade. The primary role of these clock objects is to reduce the cost of detecting conflicts among transactions. At a high level, one can assume that every location is protected by a unique versioned lock. When a writer transaction W_i commits, it is assigned a time T_i from the global clock object. In the process of committing, W_i will write T_i as the new version of each lock protecting a location that W_i modified. Every transaction R_k is assigned a start time S_k by reading the clock object before beginning, and the global clock object ensures that at the instant when a

ALGORITHM 1: A simple software global clock object

```
    ts  : AtomicInteger  // initially 0
  READ()
1 |  return ts
  GETNEXT()
2 |  return 1 + AtomicIncrement(ts)
```

committing transaction requests its commit time T_i, T_i is larger than the start time S_k of any transaction R_k that has started but not yet committed.

Given this guarantee, a running transaction R_k can identify the absence of conflicts by ensuring that when it reads any location L, the versioned lock protecting L has a version no larger than S_k. Should this comparison fail, then it is possible that a transaction W_i modified L after R_k started. STM implementations differ in their behavior at this point, with some conservatively aborting R_k and others *validating* the entire set of locations previously read by R_k to determine if R_k's execution is equivalent to one that started after W_i completed.

2.1. Software Clocks

Software implementations of a global clock object are deceptively simple. The most common implementation consists of two methods, as depicted in Listing 1.[1]

In this implementation, transactions can read the clock to attain their start time with a simple access to an atomic integer variable, and can attain a unique commit time by atomically incrementing the integer (e.g., with a `lock add` instruction on the x86). Clearly, the guarantee mentioned earlier must hold: At the time when a transaction is assigned its commit time T_i, T_i is larger than the start time S_k of every in-progress transaction. However, this implementation carries implicit memory ordering guarantees. Specifically, the read operation has acquired semantics, such that no subsequent memory operation can be ordered before it, and the getNext operation has full memory fence semantics: It cannot bypass preceding memory operations, and subsequent memory operations cannot be ordered before it. On modern implementations of the x86 ISA, these guarantees are relatively inexpensive. On processors with relaxed memory consistency, costly memory fence instructions are required to provide the necessary ordering.

2.2. Hardware Cycle Counters

The behavior of hardware cycle counters varies among both ISAs and microarchitectures, and not all cycle counters are suitable for our needs. To express the desired behaviors of hardware counters, we use the notation that p is a processor, and that v^p is the value that is returned to p when it reads its cycle counter by executing instruction t^p.

The first issue is one of *local monotonicity*. For a strictly increasing clock on processor p, $t_1^p \rightarrow t_2^p \Leftrightarrow v_1^p < v_2^p$ will always hold. For a monotonically increasing clock, the weaker property that $t_1^p \rightarrow t_2^p \Rightarrow v_1^p \leq v_2^p$ will hold.

The second issue is one of *global monotonicity*. For two processors p and q, we wish to know abstractly that $t_1^p \rightarrow t_2^q \Leftrightarrow v_1^p < v_2^q$. Unfortunately, in the absence of some event that establishes a timing relationship, we cannot "compare" the time values observed on different processors even if we know instruction t_1^p happened before t_2^q. In

[1]The properties we explore throughout this section bear strong similarity to the `std::chrono::steady_clock` object in the C++11 standard.

the opposite direction, we cannot deduce the happened before relation by comparing time. To compensate for this, we consider the following weaker scenario:

> Let p read its cycle counter as v_1^p, then let p write some value to location M, then let q read from M, and then let q read its cycle counter as v_2^q.

In this setting, we can ask the following:

—Does $t_1^p \rightarrow t_2^q \Leftrightarrow v_1^p < v_2^q$ hold if p writes an arbitrary value to M?
—Does $t_1^p \rightarrow t_2^q \Leftrightarrow v_1^p < v_2^q$ hold if p writes v_1^p to M?

On the Oracle UltraSPARC T2 processor, the `tick` register can be read to access the cycle counter. In experimental evaluation, we determined that this counter is not (even locally) monotonically increasing and thus is not suitable for our needs. More recent versions of the UltraSPARC microarchitecture also contain an `stick` register with stronger properties, which may be suitable. We leave exploration of this newer feature as future work.

On Intel "Nehalem" and later microarchitectures, the `rdtsc` instruction is locally monotonic, and for the most recent models, the instruction is *invariant* [Intel Corp. 2012, Volume 3, Chapter 17.13], meaning that the counter increments at a constant rate regardless of frequency scaling. This behavior, which is "the architectural behavior moving forward," provides a unique value on successive reads by a single core. The microarchitecture also offers an `rdtscp` instruction, which is considered to be "synchronous" (it has load fence semantics, and does not complete until preceding loads complete) [Intel Corp. 2012, Volume 2, Chapter 4.2].

We subsequently explored the global monotonicity of the x86 clocks and found that the last property held for the `rdtscp` instruction. That is, when there is a data dependence between the `rdtscp` and subsequent store by p, $t_1^p \rightarrow t_2^q \Leftrightarrow v_1^p < v_2^q$. Furthermore, the property holds on both single-socket and dual-socket multicore processors. The guarantee is provided not only among cores of the same chip, but between chips. This feat stems from (a) motherboards asserting a `RESET` signal synchronously at boot time, which synchronizes all chips' internal timestamp counters; and (b) each chip then locking the frequency of its invariant timestamp counter to the external bus frequency. Note, however, that an individual core can use `WRMSR` to set a signed offset, which is then added to the return value of `rdtsc/rdtscp` instructions. Using `WRMSR` in this manner can violate the appearance of global monotonicity.

To validate our findings, we spoke with engineers at Intel and AMD. They confirmed that:

> on modern 64-bit x86 architectures, if one core writes the result of an `rdtscp` instruction to memory, and another core reads that value prior to issuing its own `rdtscp` instruction, then the second `rdtscp` will return a value that is not smaller than the first.

This property is expected to be preserved by future x86_64 processors, but it only holds in the absence of `WRMSR` instructions. Furthermore, the *invariant* x86 cycle counter has a constant frequency independent of the operating frequency of the processor. This property is critical, as otherwise power management decisions could cause clock drift among cores or CPUs.

Given that `rdtscp` is strictly increasing, it is tempting to assume that read and getNext could both be implemented by simply executing `rdtscp`. However, it is important to understand the constraints on how `rdtsc` and `rdtscp` may be ordered within the processor. First, `rdtsc` may appear to reorder with respect to any memory operation that precedes or follows it. The `rdtscp` instruction cannot bypass a preceding load, but it can bypass a preceding store. Furthermore, the `rdtscp` instruction can appear to execute after a subsequent load or store. We now turn our attention to the consequences of

this reordering and propose extensions to STM algorithms that re-establish the strong ordering guarantees of a software global clock object when using `rdtscp`.

3. APPLYING `RDTSCP` TO STM

We now consider how the x86 cycle counter can be used to replace a shared memory global clock in an STM implementation. We focus on existing and well-known algorithms based on ownership records (orecs). The metadata and data types required for the algorithms discussed in this article are presented in Algorithm 2.

ALGORITHM 2: STM-related variables

```
GLOBAL VARIABLES
  transactions : Tx[]                    // thread metadata
  timestamp    : Global Clock Object     // see Algorithm 1
  orecs        : OwnershipRecord[]       // orec table

PER-TRANSACTION VARIABLES
  my_lock : ⟨Integer, Integer⟩  // ⟨1, thread_id⟩
  start   : Integer             // start time
  end     : Integer             // end time
  writes  : WriteSet            // pending writes by this Tx
  reads   : ReadSet             // locations read by this Tx
  locks   : LockSet             // locks held by this Tx
```

3.1. Preliminaries

In orec-based STM with timestamps (such as TL2 and TinySTM), orecs either store the identity of a lock holder or the most recent time at which the orec was unlocked. Since `rdtscp` returns a 64-bit value, we require orecs to be 64 bits wide. We also require atomic 64-bit loads. We reserve the most significant bit of the orec to indicate whether the remaining 63 bits represent a lock holder or a timestamp. This change does not have a significant impact on the risk of timestamp overflow, as a machine operating at 3GHz could operate for years without overflowing a 63-bit counter.

For simplicity in our initial discussion, we will consider algorithms with buffered update/commit-time locking, and we will not consider timestamp extension [Felber et al. 2008; Wang et al. 2007]. Both of these features can be supported without additional overhead. We will also assume a one-to-one mapping of orecs to locations in memory, as it simplifies the pseudocode.

3.2. Check-Once Ownership Records

We begin with an analysis of STM algorithms based on "check-once" orecs [Wang et al. 2007]. Though less well known than "check-twice" orecs, these algorithms offer lower per-access overhead and avoid some memory fences on processors with relaxed consistency.

Algorithms 3 and 4 present a simplified framework for STM implementations that use check-once orecs. The novelty of such algorithms stems from the ordering of accesses to the global clock relative to updates to shared memory. In the commit operation, a transaction acquires locks, validates, performs writeback, and then accesses the clock to attain a completion time. It uses this time as it releases its locks.

When a transaction begins, it accesses the clock to attain a starting time. To read a location, it reads that location, and then checks that the orec is unlocked and contains a time no newer than the transaction start time. There is no need to check the orec

ALGORITHM 3: Canonical STM algorithm with check-once ownership records

```
    TxBegin()
1   |  start ← timestamp.read()
2   |  reads ← writes ← locks ← ∅

    TxRead(addr)
3   |  if addr ∈ writes then return writes[addr]
4   |  v ← *addr
5   |  o ← orecs[addr].getValue()
6   |  if o ≤ start and ¬Locked(o) then
7   |  |  reads ← reads ∪ {addr}
8   |  |  return v
9   |  else Abort ()

    TxWrite(addr, v)
10  |  writes ← writes ∪ {⟨addr, v⟩}

    TxCommit()
11  |  if writes = ∅ then return
12  |  AcquireLocks ()
13  |  Validate(0)
14  |  WriteBack()
15  |  end ← timestamp.getNext()
16  |  ReleaseLocks(end)
```

ALGORITHM 4: Helper functions

```
    AcquireLocks()
1   |  for each addr in writes do
2   |  |  if ¬ orecs[addr].acquireIfLEQ(start) then
3   |  |  |  Abort ()
4   |  |  else locks ← locks ∪ {addr}

    ReleaseLocks(end)
5   |  for each addr in locks do
6   |  |  orecs[addr].releaseTo(end)

    WriteBack()
7   |  for each ⟨addr, v⟩ in writes do
8   |  |  *addr ← v

    Validate(end)
9   |  if end ≠ start + 1 then
10  |  |  for each addr in reads do
11  |  |  |  v ← orecs[addr].getValue()
12  |  |  |  if v ≥ start and v ≠ my_lock then Abort ()

    Abort()
13  |  for each addr in locks do
14  |  |  orecs[addr].releaseToPrevious()
15  |  restartTransaction()
```

before reading the location: Such a check is effectively subsumed by Line 1. Suppose that a read-only transaction R begins at time T and that a writing transaction W has not yet completed writeback to location L, protected by ownership record O_L. There are three possibilities:

—If W has not acquired O_L, then R can order before W.
—W has acquired but not released O_L, in which case R's check of O_L will cause it to abort.
—W completes writeback and acquires a timestamp after R starts. Thus, the time written to O_L will be after R's begin time, and R will abort. (Note that this abort may be avoided with timestamp extension.)

Since in all cases R cannot order after W while observing a value of L from before W's commit, the read is consistent with all prior reads, without a check of the orec between Lines 3 and 4.

Check-once orec algorithms typically use a shared memory global counter (as in Algorithm 1). It should be straightforward to replace the read and update of the global clock with a call to `rdtscp`. However, in the case of check-once orecs, this is not safe. Recall that an `rdtscp` can appear to execute before a preceding store operation. This creates the possibility of a location appearing to update *after* its orec is released, as Lines 14 and 15 can seem to reorder.

Such a reordering is incorrect, as it can lead to a thread observing inconsistent state. Suppose that transaction A has just completed Line 13 en route to committing a write that changes location L from value v to value v', and that transaction B is about to execute Line 1. The correctness of check-once orecs relies on the following:

—If B reads the timestamp (Line 1) before A increments the timestamp at TxCommit (Line 15), B will abort if it attempts to read L. The abort is required because the algorithm does not guarantee ordering between A's writeback (Line 14) and B's read of L (Line 4), and thus cannot guarantee that B will observe v'.
—If B reads the timestamp (Line 1) after A increments the timestamp at TxCommit (Line 15), then either (a) the step that checks the orec in B's TxRead (Line 5) happens before A releases the lock (Line 16), in which case O_L will be locked and B will conservatively abort, or (b) B will observe v' when it reads L. This follows from program order in each thread.

If Line 15 of thread A attains a timestamp before some memory update by thread A on Line 14 completes, then although it appears that A commits at time t, A's update of L from v to v' does not occur until some time $t' > t$. Thus the following order is possible:

—A increments the timestamp at TxCommit Line 15 (**reordered**);
—B reads the timestamp at Line 1 in TxBegin;
—A executes Line 14 and Line 16 in TxCommit (**reordered**); and
—B checks orec at Line 5 in TxRead.

In this case, B reads the location (Line 4) before A updates it (Line 14), but because A gets its timestamp (Line 15) before B starts (Line 1) and A releases its locks (Line 16) before B checks the orec (Line 5), B does not abort. B's continued execution with inconsistent state is not merely a violation of opacity [Guerraoui and Kapalka 2008], because it will not even be detected by validating B.

The latest IA32/x86_64 specification [Intel Corp. 2012, Volume 2, Chapter 4.2] indicates that it is possible to prevent an `rdtsc` instruction from bypassing a preceding load by either (a) preceding it with an `LFENCE` instruction, or (b) by using `rdtscp` instead of `rdtsc`. However, the specification does not give any mechanism for preventing an `rdtscp` from bypassing a preceding store. In empirical evaluation, we observed that

Line 15 can appear to execute before Line 14, even when using `rdtscp` (with its implicit `LFENCE`). Furthermore, note that an `MFENCE` instruction is insufficient in this case: Because neither `rdtscp` nor `MFENCE` reads shared memory, the sequence `store, MFENCE, rdtscp` does not guarantee that the `rdtscp` occurs after the store.

Our solution, presented in Algorithm 5, is to use an atomic read-modify-write instruction prior to the `rdtscp`. The instruction (which we refer to in pseudocode as `AtomicAdd`, and which is realized in x86 assembly code as `lock add`) adds zero to a thread-local variable, and thus has no logical effect. However, as `lock`-prefixed instruction, it enforces ordering in that its memory effects must happen *after* the stores that comprise the call on Line 14. Additionally, since it is a Read-Modify-Write (RMW) instruction, this increment involves a read, and thus the subsequent `rdtscp` on Line 16 must order after it due to the implicit `LFENCE` of `rdtscp`. When coupled with the fact that there is a data dependence between Lines 16 and 17 (the return value of `rdtscp` is used as the value written into orecs when they are released), this instruction sequence ensures that an `rdtscp` on Line 16 has the correct behavior.[2]

ALGORITHM 5: Replacement TxCommit code when using `rdtscp` withcheck-once orecs

```
    TxCommit()
        ...
15      AtomicAdd(end, 0)
16      end ← rdtscp
17      ReleaseLocks(end)
```

Let us now consider the use of `rdtscp` on Line 1. In this case, the implicit `LFENCE` ensures that getting a start time does not bypass preceding loads, which suffices for the entry to a critical section or transaction. However, it is possible for the `rdtscp` to appear to delay. Note that it cannot delay past Line 6, due to a data dependence. However, suppose that transaction A is updating location L from v to v' when transaction B begins. If thread B Line 1 occurs after thread B Line 4, then a possible ordering is:

—A completes validation at Line 13;
—B dereferences the address at Line 4 in TxRead (**reordered**);
—A completes writeback and finishes TxCommit;
—B completes its `rdtscp` at Line 1 (**reordered**); and
—B checks the orec at Line 5 in TxRead.

In this case, B will read v, but because thread A Line 17 (Algorithm 5) precedes thread B Line 1, thread B will not abort.

To guarantee that an `rdtscp` instruction in TxBegin is ordered before the transaction body, we again use a `lock` prefixed instruction to add zero to a thread-local variable. However, we now must also introduce a data dependence. Our solution, presented in Algorithm 6, stores the result of `rdtscp` and then performs a read-modify-write instruction to add zero to the result. Since there is a data dependence between the result of the `rdtscp` and the increment, the `rdtscp` must order before the increment. Additionally, since the addition is a `lock`-prefixed read-modify-write operation, on x86 processors there is a guarantee that the addition completes before any subsequent memory operations. The end result is a guarantee that Line 1 cannot reorder after any of the memory operations within a TxRead, TxWrite, or TxCommit operation.

[2]Note that a number of other instructions, including `xchg` and `lock cmpxchg`, can be used in place of `lock add`.

ALGORITHM 6: Replacement TxBegin code when using `rdtscp` with check-once orecs

```
TxBegin()
1  start ← rdtscp
2  AtomicAdd(start, 0)
3  reads ← writes ← locks ← ∅
```

3.3. Check-Twice Ownership Records

Listing 7 presents a canonical lazy STM with check-twice orecs. This style of STM algorithm is representative of TL2 [Dice et al. 2006], TinySTM [Felber et al. 2008], and most other orec-based algorithms. While ordering is required between Lines 4 and 5, and between Lines 5 and 6, which can result in more memory fences than check-once orecs, there is a useful savings at commit time: Often, validation can be avoided. When the timestamp is implemented as a shared memory counter, a transaction that successfully increments the counter from the value it observed on Line 1 is assured that no transaction changed the contents of memory during its execution, and thus validation is unnecessary. While there is no asymptotic difference in instructions (each of R reads incurs more overhead during the read operation itself, and then avoids R validation instructions at commit time), validation operations at commit time are less likely to hit in the L1 cache, and thus in the absence of memory fences, check-twice orecs with a shared memory counter can expect a slight performance advantage over check-once orecs, particularly with timestamp extension [Riegel et al. 2007].

Unfortunately, when `rdtscp` is used in place of a shared memory counter, this commit-time validation savings is lost, as it is impossible for the value of the clock that was observed at Line 14 to be only one greater than the value of the clock that was observed at Line 1. Thus, for STM algorithms with check-twice orecs, we can expect a slowdown (especially at one thread) if we replace the shared memory counter with a hardware counter.

The question remains as to whether it is correct to use `rdtscp`. Observe that there are two points at which the counter is accessed. The first is at begin time (Line 1), where the same analysis as with check-once orecs applies: The `rdtscp` does not occur "too early," but it seems possible that the instruction can delay "too late." As with check-once orecs, the solution here is to use the code sequence from Algorithm 6. This sequence ensures that the `rdtscp` in TxBegin does not delay past any shared memory operations within the transaction body.

The second point at which the counter is accessed is at commit time. There are data dependencies between the read of the counter (Line 14) and the validation (Line 15) and lock release (Line 17) operations. Thus, delay of the instruction is not possible, and the replacement of a shared memory counter with a hardware counter will not affect correctness. Furthermore, since `rdtscp` has an implicit `LFENCE`, it cannot bypass preceding load operations.

We claim that simply substituting the increment of a shared counter with an `rdtscp` instruction, as in Algorithm 8, suffices. Our lone concern is the case where the `rdtscp` bypasses a preceding store operation. In this case, we must ensure that Line 14 executing before Line 13 will not compromise correctness. The key difference relative to check-once orecs is that with check-twice orecs, Lines 4–6 alone suffice to ensure that if thread A Line 14 precedes thread B Line 1, then on an access to L, B will abort unless thread B Line 4 follows thread A Line 17. That is, B cannot safely read a location that is locked by A but may have already been updated.

Note that if thread A Line 14 were reordered before thread A Line 13, then when thread A Line 14 precedes thread B Line 1, B might be able to execute lines 6–8

ALGORITHM 7: Canonical STM algorithm with check-twice ownership records

```
    TxBegin()
1   |  start ← timestamp.read()
2   |  reads ← writes ← locks ← ∅
    TxRead(addr)
3   |  if addr ∈ writes then return writes[addr]
4   |  o1 ← orecs[addr].getValue()
5   |  v ← *addr
6   |  o2 ← orecs[addr].getValue()
7   |  if o1 = o2 and o2 ≤ start and ¬Locked(o2) then
8   |  |  reads ← reads ∪ {addr}
9   |  |  return v
10  |  else Abort()
    TxWrite(addr, v)
11  |  writes ← writes ∪ {⟨addr, v⟩}
    TxCommit()
12  |  if writes = ∅ then return
13  |  AcquireLocks ()
14  |  end ← timestamp.getNext()
15  |  Validate(end)
16  |  WriteBack()
17  |  ReleaseLocks(end)
```

ALGORITHM 8: Replacement TxCommit code when using `rdtscp` with check-twice orecs

```
    TxCommit()
    |  ...
14  |  end ← rdtscp
15  |  Validate(end)
16  |  WriteBack()
17  |  ReleaseLocks(end)
```

before thread A Line 13. In this case, B is read of L will appear to occur before A commits. However, since B believes it started after A, if B reads L again or if B reads some other location written by A *after* thread A completes Line 17, B will not detect an inconsistency. Fortunately, this problem is averted because thread A Line 13 is implemented with the atomic `cmpxchg` instruction. Since the instruction is a read-modify-write that entails both a load and a store, and since `rdtscp` has an implicit `LFENCE`, thread A Line 14 cannot bypass thread A Line 13 in Algorithm 8.

3.4. Timestamp Extension

A common practice in STM algorithms is to "extend" a transaction's start time to avoid aborts in the read function (Algorithm 3 line 9 and Algorithm 7 line 10). The technique is simple [Riegel et al. 2007; Wang et al. 2007]: If transaction T is reading location L for the first time and the orec associated with L (O_L) is unlocked but newer than $T.start$, but no location in $T.reads$ has been locked since T began, then it is safe to add L to T's read set and update $T.start$ to the value in O_L. Intuitively, all prior loads and stores performed by T would have been correct if T did not begin until after O_L was most

recently unlocked, and thus T can simply update its start time to achieve the illusion that it started later than it actually did.

Timestamp extension replaces the call to `Abort` in TxRead with the sequence in Algorithm 9.

ALGORITHM 9: Timestamp extension with a global shared memory clock

```
1 tmp ← timestamp.read()
2 VALIDATE(start)
3 start ← tmp
```

Given the properties of `rdtscp` discussed earlier, it is correct to use `rdtscp` in place of a shared memory counter only if ordering can be guaranteed between the read of the timestamp and the call to `Validate()`. As before, we use a `lock`-prefixed instruction and a data dependence to provide this ordering. The resulting timestamp extension code appears in Algorithm 10. Note that as in all previous uses of an additional `lock add` instruction, our modification of a thread-local variable is likely to result in little additional latency.

ALGORITHM 10: Timestamp extension with `rdtscp`

```
1 tmp ← rdtscp
2 AtomicAdd(tmp, 0)
3 VALIDATE(start)
4 start ← tmp
```

4. PRIVATIZATION SAFETY

In languages whose memory model demands static separation [Abadi et al. 2008], such as Haskell, Scala, and Clojure, the algorithms in Section 3 can be used directly. However, the current draft specification for adding TM to C++ [Adl-Tabatabai et al. 2012] requires implicit privatization safety [Harris et al. 2010; Menon et al. 2008; Spear et al. 2007; Wang et al. 2007] instead of static separation. We now turn our attention to mechanisms that can make our algorithms in Section 3 privatization safe.

4.1. The Privatization Problem

In general, privatization safety can be thought of as the need to prevent two problems [Spear et al. 2007], related to "doomed transactions" and "delayed cleanup." First, when a transaction T_p commits and makes some datum D private, the STM library must ensure that subsequent nontransactional accesses by T_p do not conflict with accesses performed by transactions that have not yet detected that they must abort on account of T_p's commit. Second, when T_p commits, the STM library must ensure that no transaction T_o that committed or aborted before T_p has pending cleanup (a redo or undo log) to D. The danger is that T_p's nontransactional access to D could race with that cleanup.

In general, there are two approaches to privatization safety. The first is for T_p to block during its commit phase and wait for all extant transactions to either commit or abort *and clean up*. This technique has come to be known as quiescence [Menon et al. 2008]. The second approach is to use orthogonal solutions to the two problems. The approach, known as the Detlefs algorithm [Marathe et al. 2008; Spear et al. 2008], assumes a write-back STM. In an STM with write-back, delayed cleanup can be achieved by serializing the writeback phase of all committed transactions, and doomed transactions can be detected before they do harm by requiring them to poll a global count of committed transactions on every read and to validate whenever the count changes.

4.2. Achieving Privatization Safety

Unfortunately, polling to solve the doomed transaction problem introduces the very shared memory bottleneck that our use of cycle counters seeks to avoid. Furthermore, since the cycle counter advances according to physical time, instead of upon writer transaction commits, every poll of the counter would require a validation, since every read would return a value $> start$. This would lead to quadratic validation overhead. Instead, our privatization-safe algorithms employ quiescence.

Algorithm 11 introduces a privatization-safe STM algorithm that uses `rdtscp` in place of a global shared clock. This algorithm incorporates timestamp extension and check-twice orecs. Replacing check-twice orecs with check-once orecs is trivial.

ALGORITHM 11: Privatization safe STM algorithm using check-twice orecs and `rdtscp`

```
    TxBegin()
1   |  start ← rdtscp
2   |  AtomicAdd(start, 0)
3   |  reads ← writes ← locks ← ∅

    TxRead(addr)
4   |  if addr ∈ writes then return writes[addr]
5   |  while true do
6   |  |  o1 ← orecs[addr].getValue()
7   |  |  v ← *addr
8   |  |  o2 ← orecs[addr].getValue()
9   |  |  if o1 = o2 and o2 ≤ start and ¬Locked(o2) then
10  |  |  |  reads ← reads ∪ {addr}
11  |  |  |  return v
12  |  |  if Locked(o2) then continue
    |  |  // extend validity range
13  |  |  tmp ← start
14  |  |  start ← rdtscp
15  |  |  AtomicAdd(start, 0)
16  |  |  for each addr in reads do
17  |  |  |  v ← orecs[addr].getValue()
18  |  |  |  if v ≥ tmp then Abort ()

    TxWrite(addr, v)
19  |  writes ← writes ∪ {⟨addr, v⟩}

    TxCommit()
20  |  if writes = ∅ then
21  |  |  start ← ∞
22  |  |  return
23  |  AcquireLocks ()
24  |  end ← rdtscp
25  |  Validate(end)
26  |  WriteBack()
27  |  start ← ∞
28  |  ReleaseLocks(end)
    |  // Quiescence
29  |  for each tx in transactions do
30  |  |  while tx.start ≤ end do wait
```

Since this algorithm uses timestamp extension, we can employ a validation fence [Spear et al. 2007], rather than the more coarse-grained transaction fence, for

privatization safety. To maximize the effectiveness of this technique, the timestamp extension on Lines 13 to 18 differs slightly from Algorithm 10 so that a concurrent thread that is in the process of quiescence can observe a validating thread early. In effect, whenever an in-flight transaction T_i (one that has not reached its commit point) begins a validation, any concurrent committer T_c can be sure that either T_i is doomed and will abort, or that T_i and T_c do not conflict, and T_c need not wait on T_i.

5. EVALUATION

In this section, we present performance results for several STM algorithms that use `rdtscp` in place of a shared counter. For completeness of evaluation, we consider two categories of algorithms. The first category consists of the most popular algorithms that are not privatization safe, and their `rdtscp`-enhanced versions:

—LSA: The write-through version of the LSA algorithm, also known as TinySTM-WT [Felber et al. 2008]. This is a check-twice algorithm with extensible timestamps and undo logs.
—LSA-Tick: LSA, but using `rdtscp` in place of a global shared counter.
—Patient: A redo log/commit-time locking version of LSA [Spear et al. 2009], which we augmented to use check-once orecs.
—Patient-Tick: A variant of Patient that uses `rdtscp`.
—TL2: TL2 features check-twice orecs and commit time locking with redo logs, but it does not have extensible timestamps [Dice et al. 2006]. Our version uses the "GV1" clock mechanism, which is equivalent to the shared memory counter in LSA.
—TL2-Tick: TL2, extended to use `rdtscp` in place of a global shared counter.

We also evaluate the following privatization-safe algorithms:

—NOrec: A privatization-safe, redo-log based algorithm that does not use orecs [Dalessandro et al. 2010].
—OrecELA: A variant of the Detlefs algorithm [Marathe and Moir 2008; Spear et al. 2008], which uses check-once orecs and extensible timestamps.
—OrecELA-Q: A variant of OrecELA that uses check-twice orecs and extensible timestamps, but which relies on quiescence instead of polling to ensure privatization-safety.
—ELA-Tick-1: A version of Algorithm 11 using check-once orecs.
—ELA-Tick-2: A variant of Algorithm 11 using check-twice orecs.

All algorithms were implemented within the RSTM framework [Spear 2010], in order to minimize variance due to implementation artifacts. Experiments were performed on two machines, both based on the 6-core/12-thread Intel Xeon X5650 CPU. The first machine was a single-chip configuration with 12 hardware threads, the second a two-chip configuration with 24 hardware threads. The underlying software stack included Ubuntu Linux 12.04, kernel version 3.2.0-27, and gcc 4.7.1 (–O3 optimizations). All code was compiled for 64-bit execution, and results are the average of five trials. We evaluated STM algorithms on targeted microbenchmarks from the RSTM suite, and also measured their performance on the STAMP benchmarks [Minh et al. 2008]. As in prior work, we omitted Bayes and Yada from the evaluation: Bayes exhibits nondeterministic behavior, and the released version of Yada crashes for algorithms that use redo logs.

5.1. Microbenchmark Performance

The first claim we evaluate is whether hardware cycle counters can be used to accelerate workloads with frequent small writer transactions. Even in the absence of aborts, such a

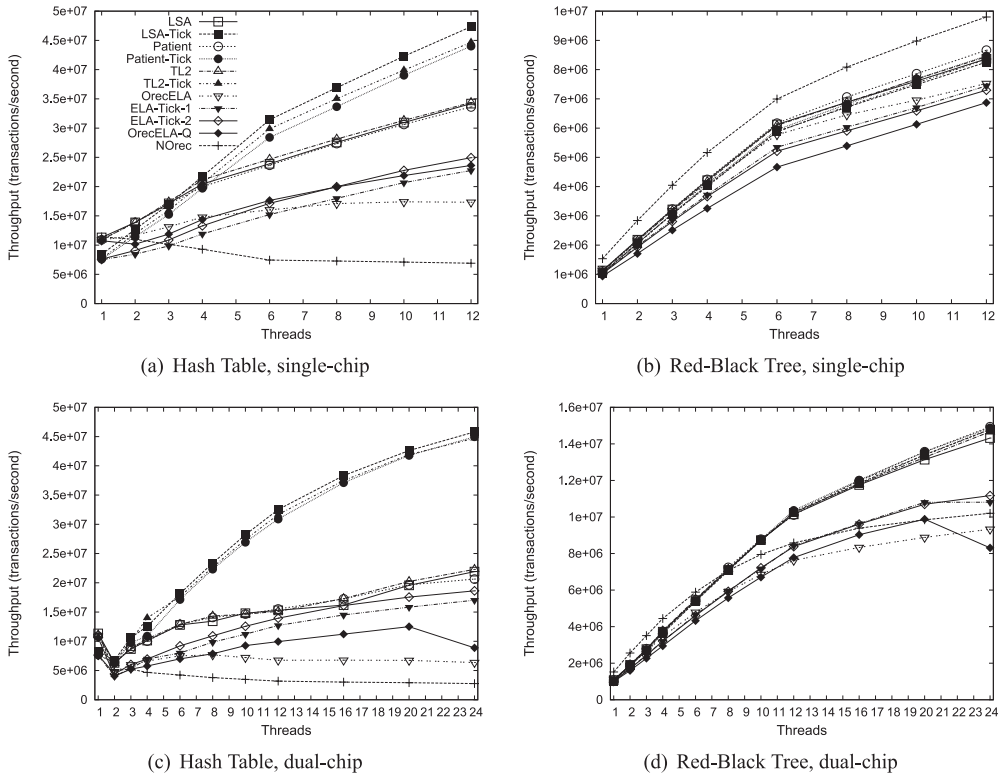


Fig. 1. Microbenchmark results. Hashtable experiments are configured with 256 buckets, 8-bit keys, and a 0% lookup ratio. Red-Black Tree experiments use 20-bit keys and an 80% lookup ratio.

workload can fail to scale adequately due to contention among transactions attempting to update the shared memory counter.

Figures 1(a) and 1(c) present the performance of our STM algorithms for a microbenchmark in which all transactions repeatedly access the same hash table. The data structure is configured with 256 buckets and linear chaining. Transactions attempt to insert and delete 8-bit keys with equal probability. The data structure was prefilled with half of the keys in the range so that 50% of transactions succeed in their insert/delete attempts; in other words, 50% of the transactions are not read-only.

On the single-chip machine, the algorithms exhibit performance that separates into four categories. The most scalable algorithms are those that do not have any bottlenecks in the STM implementation: Our `rdtscp`-based algorithms without privatization safety. Since the benchmark has virtually no aborts, undo logging has less overhead than redo logging, and LSA-Tick has the least overhead. This is expected because LSA variants use eager locking and in-place update. The slight benefit observed by TL2-Tick relative to Patient-Tick is due to the added cost of ordering in Patient-Tick (recall that Patient-Tick uses check-once orecs, and thus requires an additional `lock`-prefixed instruction in the commit operation).

The second group of algorithms are those that are not privatization safe, but which suffer from contention on the shared memory counter. Prior to this work, these would have been the best performing STM algorithms. The third group is the privatization-safe algorithms that use orecs. Here, we see that at low thread counts, serialization of writeback provides the best performance, but due to the frequency of writer

transactions, this becomes a bottleneck at high thread counts. Thus, at higher thread counts, the more heavyweight quiescence operation employed by our `rdtscp`-based algorithms performs better: For small writer transactions, it is more important to allow parallel writeback than to avoid the overhead of quiescence. The last category consists solely of NOrec. NOrec is known to perform poorly for workloads with small, frequent writer transactions.

On the dual-chip machine, we see roughly the same grouping. However, three additional trends emerge. First, all algorithms experience a slowdown at two threads, due to interchip communication. The version of the Linux operating system used in this experiment places threads as far apart as possible, and thus with two threads, any shared STM metadata must bounce between the caches of the two chips; this includes the ownership records themselves, which no longer remain local to a single chip's cache. The second new trend is that contention for shared counters is higher. This leads LSA, TL2, and Patient to perform much worse than their `rdtscp`-based counterparts. Since transactions are small, the overhead of quiescence remains manageable because no quiescence operation delays for long, and thus the `rdtscp`-based privatization-safe algorithms can perform almost as well as these unsafe algorithms. Finally, writer serialization on a multichip system is particularly costly, resulting in OrecELA and NOrec both failing to scale. Note that OrecELA-Q scales better than OrecELA because it has lower quiescence overhead and avoids the costly writer serialization of OrecELA.

On the opposite end of the spectrum, Figures 1(b) and 1(d) present the performance of these algorithms on a red-black tree. Eighty percent of transactions perform lookups, with the remaining transactions split equally between insert and remove operations. Again, the data structure is prepopulated with half of the keys in a 20-bit range so that half of all attempts to update the data structure succeed. The net effect is that 90% of transactions are read-only. Furthermore, transactions are substantially larger, consisting of more than two dozen reads, on average.

This workload nullifies the benefits of using `rdtscp`: Our "Tick" algorithms require writers to validate at commit time while their counterparts need not; the cost of quiescence is higher because committing writers must wait on long-running transactions to validate; and shared memory counters are not a significant source of contention in the first place. On the single-chip machine, we see NOrec perform best at all thread counts, and `rdtscp`-based algorithms perform a small constant factor worse than their non-`rdtscp` counterparts. The effect is less pronounced on the dual-chip system, since coherence traffic on shared counters is severe. As a result, at high thread counts, we see the privatization-safe `rdtscp` algorithms outperforming OrecELA, OrecELA-Q, and NOrec.

The performance differences between quiescence and polling/writer serialization are nuanced. To gain more insight into where quiescence overheads lie, we instrumented the microbenchmarks to count cycles spent in the quiescence operation, as well as cycles in quiescence spent specifically waiting for a thread to validate. Some results from the single-chip and dual-chip machines appear in Table I, with the "Total" column depicting the average number of cycles spent in quiescence for each transaction, and "Waiting" representing the average number of cycles within the quiescence operation that were spent waiting for any thread's start time to change. For the hash table, most of the overhead of quiescence is due to cache misses, not waiting; this is represented by the near-constant value for "Waiting" and a "Total" cost that scales roughly linearly with the number of threads: In effect, quiescence is amounting to a single cache miss per thread to read that thread's start time and conclude that no further waiting is required. In contrast, the red-black tree workload shows a significant fraction of quiescence time spent in "Waiting." This indicates that the quiescing thread experiences delays waiting for multiple threads to reach their next validation point. Since conflicts are rare in

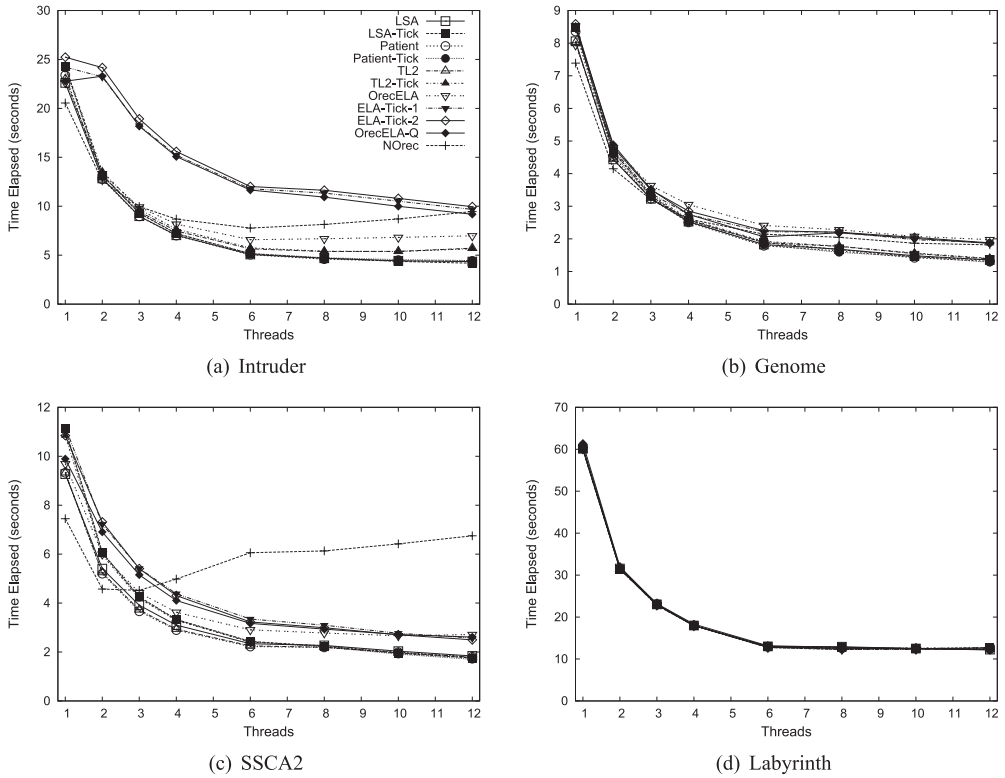
Table I. Quiescence Overhead

Single-chip system

	Red-black tree		Hash table	
Thread count	Total	Waiting	Total	Waiting
1	40	0	40	0
2	1403	1244	287	129
4	2618	2293	503	238
6	3294	2886	688	297
8	4123	3474	780	263
10	4955	4174	929	269
12	5554	4670	1063	282

Dual-chip system

	Red-black tree		Hash table	
Thread count	Total	Waiting	Total	Waiting
1	36	0	35	0
2	2318	1831	713	323
4	3706	2885	1290	642
6	4216	3265	1642	697
12	5483	4131	2506	811
18	11018	8975	3829	1111
24	18408	15800	7281	4006



(a) Intruder (b) Genome

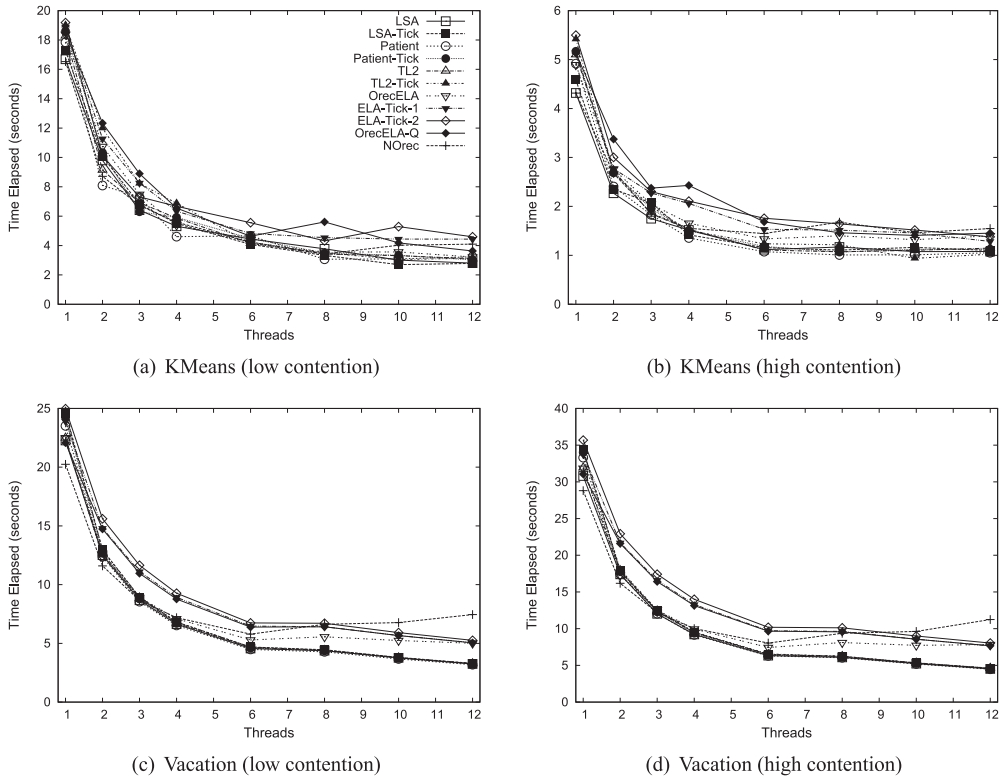
(c) SSCA2 (d) Labyrinth

Fig. 2. STAMP results on the single-chip system (1/2).

this workload, this often entailed waiting for other transactions to reach their commit point. The implication is that for workloads with large transactions and few conflicts, quiescence can be a significant overhead, comparable to the cost of writer serialization.

5.2. STAMP Performance

The variety of behaviors exhibited by the different STAMP benchmarks provide additional insight into the benefits and weaknesses of our `rdtscp`-based algorithms. Figures 2 and 3 present results for the single-chip system, and Figures 4 and 5 present results for the dual-chip system.



(a) KMeans (low contention)

(b) KMeans (high contention)

(c) Vacation (low contention)

(d) Vacation (high contention)

Fig. 3. STAMP results on the single-chip system (2/2).

Intruder. Intruder features transactions of varying lengths, with a mix of read-only and writing transactions. One characteristic exhibited within each larger transaction is that the early accesses are more likely to participate in a conflict than later accesses. In polling-based privatization-safe algorithms, this behavior is immaterial to privatization overhead, since a committing writer does not wait for in-flight transactions. However, with quiescence, a committing writer must wait. The nature of orec-based algorithms is such that a transaction will not validate and detect a conflict unless it reads a location that cannot be added to its read set. Thus, quiescence-based algorithms spend a long time waiting for transactions that ultimately abort but that never detect the need to validate. This overhead significantly degrades the performance of our privatization-safe `rdtscp` algorithms. Otherwise, the use of `rdtscp` has no noticeable effect on performance for the single- or dual-chip machine.

Genome. Genome performance is dominated by a large read-only phase, and transactions in general do not exhibit many conflicts. Consequently, on both the single and dual-chip systems, all algorithms perform at roughly the same level. The only differentiation we see is that privatization-safe algorithms do not scale as well due to their serialization/blocking at commit time. The more pronounced separation on the dual-chip system illuminates that OrecELA, with its polling and writer serialization, performs slightly worse. This is no surprise, as this mechanism causes significant coherence traffic.

SSCA2. In many regards, SSCA2's behavior is modeled by our Hashtable microbenchmark: all transactions perform writes, and transactions are frequent and

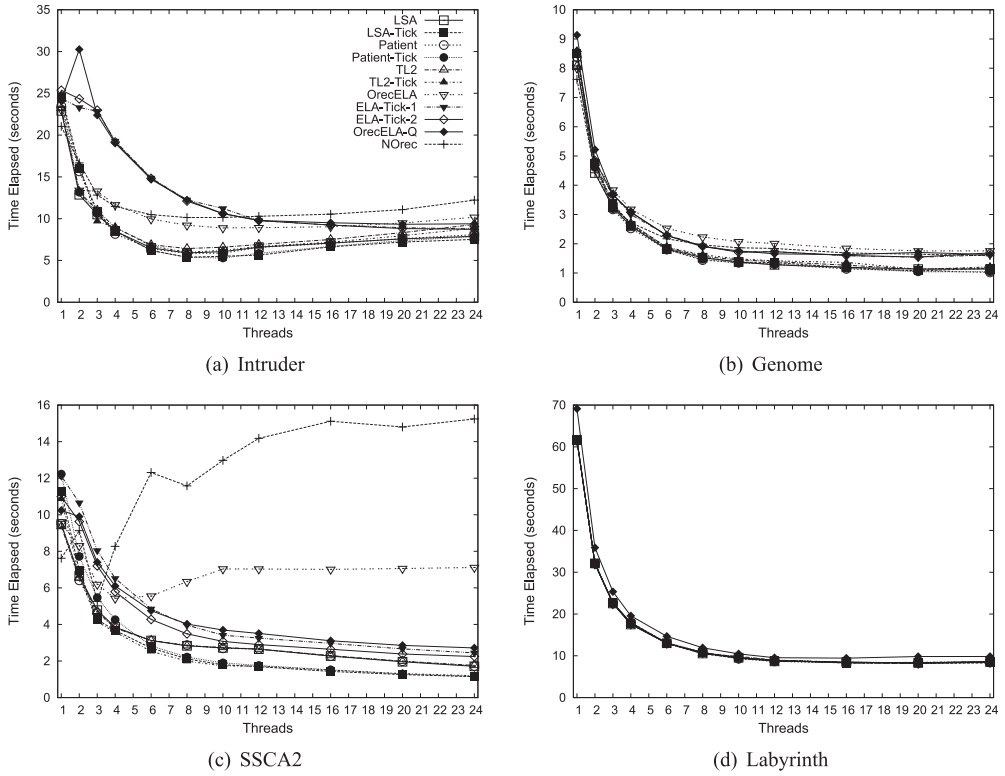
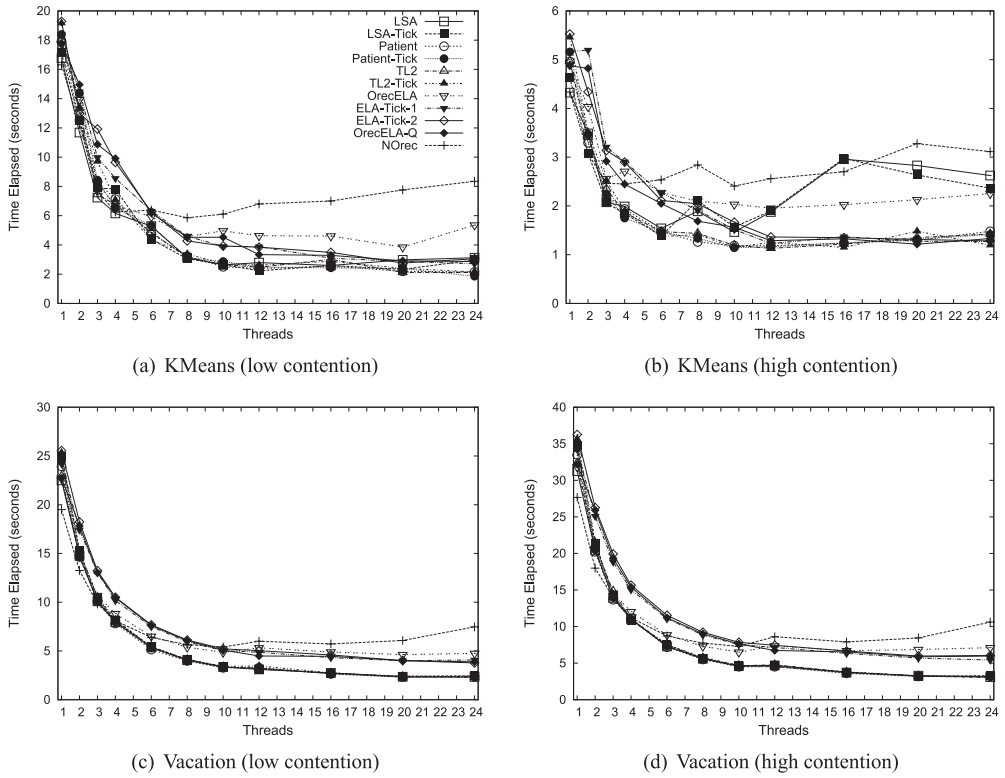


Fig. 4. STAMP results on the dual-chip system (1/2).

small. NOrec is known to perform poorly, due to serialization at commit time, and on the dual-chip system, OrecELA performs poorly as well. The only noteworthy result is to see, again, that on a dual-chip system, the coherence traffic caused by the shared counter causes a reduction in performance, and thus the use of `rdtscp` proves beneficial.

KMeans. In KMeans, transaction durations vary, particularly in the high-contention workload. As a result, the cost of quiescence can occasionally be high, resulting in a penalty on the single-chip system for our privatization safe, `rdtscp`-based algorithms. While this cost is significant on the single-chip system, the characteristics of the dual-chip system have a mitigating effect. Since quiescence entails less contention and bus traffic than writer serialization, NOrec and OrecELA degrade on the dual-chip system, leaving our `rdtscp`-based algorithms and OrecELA-Q as the best privatization-safe algorithms for this workload. A minor additional point is that under high contention, we see performance anomalies for the write-through algorithms. These variations are due to contention management; different backoff parameters would have smoothed the performance of these curves.

Vacation. Vacation is dominated by large writer transactions. As with the red-black tree microbenchmark, the size of these transactions serves as a buffer to minimize the overhead from shared memory bottlenecks. Furthermore, since transactions rarely conflict, timestamp extension does not occur often in practice. As a result, for modest to large thread counts, it is safe to expect every transaction to validate at commit time. This eliminates the main advantage of check-twice orecs. We see comparable



(a) KMeans (low contention)

(b) KMeans (high contention)

(c) Vacation (low contention)

(d) Vacation (high contention)

Fig. 5. STAMP results on the dual-chip system (2/2).

performance for all non-privatization-safe algorithms, and only slight separation between the privatization-safe algorithms. As before, the general trend is that quiescence is more expensive at lower thread counts, and writer serialization more expensive at higher thread counts.

Labyrinth. The STAMP Labyrinth application contains racy reads, which are safe in the context of the benchmark. In contrast to the original Lee-TM algorithm [Ansari et al. 2008] upon which it is based, Labyrinth conflates memory speculation and control flow speculation: In Labyrinth transactions, a quadratic number of un-instrumented (nontransactional) reads are performed within a transaction, after which a linear number of locations are accessed via instrumented reads and writes. These instrumented accesses consist of checks that ensure no intervening writes since the previous uninstrumented reads, and then transactional updates to those same locations. The programmer uses explicit *self-abort* when the nontransactional reads are shown to be inconsistent. This benchmark is thus extremely artificial: Neither a proper compiler-based TM nor a hardware TM would be able to eliminate instrumentation of the majority of accesses within a transaction. We opted to restore the Lee-TM form to the Labyrinth application for our tests. This change decouples control-flow speculation from transactional speculation on memory accesses, but it does not affect correctness. However, it results in transactions comprising a tiny fraction of overall execution time, and thus all TM implementations scale equally well.

6. CONCLUSIONS AND FUTURE WORK

In this article, we explored the role that x86 hardware cycle counters can play in eliminating bottlenecks and reducing overhead for software transactional memory algorithms. In the absence of privatization safety, our findings were positive: In workloads for which shared memory counters are known to cause scalability bottlenecks, our "Tick" algorithms outperformed the state of the art, while in other cases our algorithms performed on par with their non-Tick equivalents. When privatization safety is required, however, the use of cycle counters prevents the some valuable optimizations, such as polling to detect doomed transactions. On a single-chip system, this generally led to worse performance, though on a dual-chip system the penalty was mitigated by the ability our algorithms offer for committing writer transactions in parallel.

There are several questions that this work raises for hardware designers—chief among them is the nature of ordering between memory operations and accesses to the cycle counter. The correctness of our algorithms relied on the introduction of `lock`-prefixed operations and data dependencies between accesses to the cycle counter and subsequent memory operations. Together, these techniques provided the "write before timestamp access" ordering that our check-once orecs required, as well as the "timestamp access before read" ordering that all our algorithms required during transaction begin. If cycle counters become a more popular mechanism for synchronizing threads, it may be beneficial to offer a stronger variant of the `rdtsc` instruction, particularly one that guarantees ordering between the read of the cycle counter and *subsequent* memory operations. Another question relates to generality: Can the ARM, SPARC, and POWER architectures provide cycle counters with strong enough guarantees to support our algorithms?

In addition, the strong performance of our non–privatization-safe algorithms leads to questions about the benefit of implicit privatization safety. Perhaps the absence of bottlenecks in our algorithm will make strong isolation [Shpeisman et al. 2007] viable for unmanaged languages, or at least provide an incentive for new explorations of programming models with explicit privatization. In this regard, we are particularly excited that the use of `rdtscp` in place of accesses to a global shared counter will enable a strongly isolated system to implement individual loads and stores as mini-transactions that do not suffer from scalability bottlenecks.

ACKNOWLEDGMENTS

We thank Ravi Rajwar, Stephan Diestelhorst, and Dave Dice for their advice during the conduct of this research. We are also grateful to the TRANSACT community for their feedback on an earlier version of this manuscript.

REFERENCES

Abadi, M., Birrell, A., Harris, T., and Isard, M. 2008. Semantics of transactional Memory and Automatic Mutual Exclusion. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*.

Adl-Tabatabai, A.-R., Shpeisman, T., and Gottschlich, J. 2012. Draft specification of transactional language constructs for C++. Version 1.1. http://justingottschlich.com/tm-specification-for-c-v-1-1/.

Ansari, M., Kotselidis, C., Jarvis, K., Lujan, M., Kirkham, C., and Watson, I. 2008. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*.

Dalessandro, L., Spear, M., and Scott, M. L. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*. Bangalore, India.

Dice, D., Shalev, O., and Shavit, N. 2006. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*.

Dragojevic, A., Guerraoui, R., and Kapalka, M. 2009. Stretching transactional memory. In *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation*. Dublin, Ireland.

Felber, P., Fetzer, C. and Riegel, T. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.

Fetzer, C. and Felber, P. 2007. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*.

Fraser, K. 2003. *Practical Lock-Freedom*. Ph.D. thesis. King's College, University of Cambridge.

Guerraoui, R. and Kapalka, M. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.

Harris, T., Larus, J., and Rajwar, R. 2010. *Transactional Memory* (2nd ed.). Synthesis Lectures on Computer Architecture. Morgan & Claypool.

Herlihy, M. P., Luchangco, V., Moir, M., and Scherer III, W. N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*. Boston, MA.

Intel Corp. 2012. *Intel 64 and IA-32 Architectures Software Developer's Manual* 325462-044US Ed. Intel Corp.

Marathe, V. and Moir, M. 2008. Toward high performance nonblocking software transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*.

Marathe, V. J., Scherer III, W. N., and Scott, M. L. 2005. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*.

Marathe, V. J., Spear, M., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L. 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*.

Marathe, V. J., Spear, M., and Scott, M. L. 2008. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 37th International Conference on Parallel Processing*.

Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R., Saha, B., and Welc, A. 2008. Practical weak-atomicity semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

Riegel, T., Fetzer, C. and Felber, P. 2006. Snapshot isolation for software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.

Shavit, N. and Touitou, D. 1995. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*.

Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R. L., Moore, K., and Saha, B. 2007. Enforcing isolation and ordering in STM. In *Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*.

Spear, M. 2010. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*.

Spear, M., Dalessandro, L., Marathe, V. J., and Scott, M. L. 2008. Ordering-based semantics for aoftware transactional memory. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*.

Spear, M., Dalessandro, L., Marathe, V. J., and Scott, M. L. 2009. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*.

Spear, M., Marathe, V., Dalessandro, L., and Scott, M. 2007. Privatization techniques for software transactional memory (POSTER). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*.

Wang, C., Chen, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.-R. 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*.

Zhang, R., Budimlic, Z., and Scherer III, W. N. 2008. Commit phase in timestamp-based STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*.

Received June 14, 2013; revised October 5, 2013; accepted November 12, 2013