# Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters *

Wenjia Ruan, Yujie Liu, and Michael Spear
Lehigh University
{wer210, yul510, spear}@cse.lehigh.edu

## Abstract

Time-based transactional memories typically rely on a shared memory counter to ensure consistency. Unfortunately, such a counter can become a bottleneck. In this paper, 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*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

***General Terms*** Algorithms, Design, Performance

***Keywords*** Transactional Memory, Privatization, `rdtscp`, Counters

## 1. Introduction

Most high-performance Software Transactional Memory [19] (STM) implementations reduce the common-case overhead of validation by using timestamps. The technique, first employed in the LSA [17] and TL2 [4] 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 [7, 10, 13] can be avoided. Since 2006, virtually every single-version STM that uses ownership records has employed a global shared counter [5, 6, 12, 15, 24–26].

There are two problems with global shared clocks. First, clock-based techniques for avoiding validation are heuristic, and in the worst case, a clock-based STM might still validate the entire read set on every read, resulting in quadratic overhead. 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 of the TL2 algorithm [16] offered 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. 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., is to use the multimedia timer present in some systems in place of a shared memory clock [18]. 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 paper, 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, but the details of these counters vary among ISAs and even micro-architectures. 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 paper is organized as follows. Section 2 discusses hardware cycle counter properties in the x86 and SPARC architectures, and identifies potential pitfalls when using these counters in place of a shared memory clock. Section 3 develops STM algorithms based on the x86 `rdtscp` instruction. Section 4 considers techniques for making these algorithms privatization safe. Section 5 evaluates our algorithms on single and dual-chip x86 systems, and Section 6 concludes.

## 2. Cycle Counters

The behavior of hardware cycle counters varies among both ISAs and micro-architectures, 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 $t^p$.[1]

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

* This work was supported in part by the National Science Foundation through grants CNS-1016828 and CCF-1218530.

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

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 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 $M$, 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.[2]

On an Intel Xeon X5650 (Sandy Bridge) processor, we found that the `rdtsc` instruction was locally monotonic, but not strictly increasing. The processor also offers an `rdtscp` instruction, which is considered to be "synchronous" (it has load fence semantics, and does not complete until preceding loads complete). This variant is strictly increasing.

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 $p$, then $t_1^p \rightarrow t_2^q \Leftrightarrow v_1^p < v_2^q$. 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.

To validate our findings, we spoke with engineers at Intel and AMD. They claimed 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. Furthermore, the x86 cycle counter has a constant frequency independent of the operating frequency of the processor. This property is critical, since otherwise power management decisions could cause clock drift among cores or CPUs.

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 can bypass a preceding store. Furthermore, the `rdtscp` instruction can appear to execute after a subsequent memory instruction.

## 3. Applying `rdtscp` to STM

We now consider how the x86 cycle counter can be used to accelerate an STM implementation. We focus on existing and well-known algorithms based on ownership records (orecs).

[2] Subsequent to the conduct of this research, we learned that there is also an `stick` register with stronger properties. Exploration of its use in the algorithms presented in this paper is future work.

**Algorithm 1:** STM-Related Variables

```
GLOBAL VARIABLES
  transactions : Tx[]                  // thread metadata
  timestamp    : Timestamp             // see Algorithm 4
  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
```

**Algorithm 2:** Check-once Timestamps

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

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

11 TxWrite(addr, v)
12     writes ← writes ∪ {⟨addr, v⟩}

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

### 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, since 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 [6, 25]. 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 Single-Check Ownership Records

We begin with an analysis of STM algorithms based on "check-once" orecs [25]. 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.

**Algorithm 3:** 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()
```

**Algorithm 4:** Timestamp Implementations

```
// Timestamp implementation based on global integer counter.
  ts    : Integer    // initially 0
read()
    return ts

getNext()
    return 1 + AtomicIncrement(ts)


// Timestamp implementation based on hardware cycle counter.
read()
    return rdtscp

getNext()
    return rdtscp
```

Algorithm 1 presents the basic metadata required for all orec-based STM algorithms discussed in this paper. Algorithms 2 and 3 present a simplified framework for algorithms 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 simply accesses that location, and then checks that the orec is unlocked and contains a time earlier than the transaction start time. There is no need to check the orec 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.

Single-check orec algorithms typically use a shared memory global counter (as in Algorithm 4). 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 single-check 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 $A$ increments the timestamp (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:

1. $A$ increments the timestamp at TxCommit line 15 (**reordered**);
2. $B$ reads the timestamp at line 1 in TxBegin;
3. $A$ executes line 14 and line 16 in TxCommit (**reordered**);
4. $B$ checks orec at line 5 in TxRead.

In this case, $B$ reads the location (Line 4) before $A$ updates it (Line 14), but since $A$ 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 [8], because it will not even be detected by validating $B$.

The latest IA32/x86_64 specification [11] indicates that it is possible to prevent an `rdtsc` instruction from bypassing a preceding load by either (a) using an `LFENCE` instruction, or by using `rdtscp`. 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`).

**Algorithm 5:** Check-twice Timestamps

```
    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)
```

The solution we employ is to place an atomic fetch-and-add instruction between lines 14 and 15. This instruction adds zero to a thread-local variable, and thus has no effect. However, as an x86 atomic, it enforces ordering in that it happens *after* line 14. Since it is a read-modify-write (RMW) operation, it entails a read, and thus the subsequent `rdtscp` on line 15 must order after it. When coupled with the fact that there is a data dependence between lines 15 and 16, this fence ensures that an `rdtscp` on line 15 has the correct behavior.[3]

Let us now consider the use of `rdtscp` on line 1. In this case, the `LFENCE` semantics ensure that the read 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

1. $A$ completes validation at line 13;
2. $B$ dereferences the address at line 4 in TxRead (**reordered**);
3. $A$ completes writeback and finishes TxCommit;
4. $B$ reads timestamp at line 1 (**reordered**);
5. $B$ checks orec at line 5 in TxRead.

In this case, $B$ will read $v$, but since thread $A$ Line 16 precedes thread $B$ Line 1, thread $B$ will not abort. It is not clear that an `LFENCE` after line 1 suffices to prevent this error, though in practice we did not observe errors even in the absence of such a fence.

### 3.3 Check-Twice Ownership Records

Listing 5 presents a canonical lazy STM with check-twice orecs. This style of STM algorithm is embodied by TL2 [4], TinySTM [6], 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 single-check 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 [18].

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 return value of line 14 to be only one greater than the return value of 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 single-check orecs applies: the `rdtscp` does not occur "too early", but it seems possible that the instruction can delay "too late". The second 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. Let us now consider the case where the `rdtscp` bypasses a preceding store operation. In this case, our concern is that line 14 executing before line 13 will compromise correctness. The key difference 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 4–6 *before* thread $A$ Line 13. In this case, $B$'s 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 since thread $A$ Line 13 is implemented with the atomic `cmpxchg` instruction. Since the instruction entails both a load and a store, and since `rdtscp` has the effect of being preceded by an `LFENCE`, thread $A$ Line 14 cannot bypass thread $A$ Line 13. Note that as with single-check orecs, it appears that some ordering must be enforced between lines 1 and 4. Again, it is not clear that an `LFENCE` suffices, though in practice we did not observe errors even in the absence of an `LFENCE`.

### 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 2 line 9 and Algorithm 5 line 10). The technique is simple [18, 25]: if transaction $T$ is reading $L$ for the first time and $O_L$ is 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

[3] It is not clear whether an `MFENCE` would suffice in place of an atomic RMW. Fortunately, on modern x86 processors, the RMW operation tends to have lower latency than `MFENCE`.

locked, 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` with the sequence in Algorithm 6.

**Algorithm 6:** Timestamp Extension

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

Given the properties of `rdtscp` discussed above, 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 an `LFENCE` instruction to attempt to provide this ordering, though no errors were observed when the instruction was not used.

## 4. Privatization Safety

The current draft specification for adding TM to C++ [1] calls for privatization safety [9, 15, 22, 25]. We now turn our attention to mechanisms that make our algorithms from Section 3 privatization safe.

### 4.1 The Privatization Problem

In general, privatization safety can be thought of as two related problems [22], 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 [15]. The second approach is to use orthogonal solutions to the two problems. The approach, known as the Detlefs algorithm [14, 23], 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 7 employs check-twice orecs and timestamp extension. Since it uses timestamp extension, we can employ a validation fence [22], rather than the more coarse-grained transaction fence [22], for privatization safety. 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 $T_i$. Note that a `LFENCE` appears necessary between lines 13 and 14, to ensure that a transaction does not declare an updated start time which then occurs after validation has begun.

**Algorithm 7:** Privatization Safety

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

   TXREAD(addr)
3      if addr ∈ writes then return writes[addr]
4      while true do
5          o1 ← orecs[addr].getValue()
6          v ← *addr
7          o2 ← orecs[addr].getValue()
8          if o1 = o2 and o2 ≤ start and ¬Locked(o2) then
9              reads ← reads ∪ {addr}
10             return v
11         if Locked(o2) then continue
           // extend validity range
12         tmp ← start
13         start ← timestamp.read()
14         for each addr in reads do
15             v ← orecs[addr].getValue()
16             if v ≥ tmp then ABORT ()

   TXWRITE(addr, v)
17     writes ← writes ∪ {⟨addr, v⟩}

   TXCOMMIT()
18     if writes = ∅ then
19         start ← ∞
20         return
21     ACQUIRELOCKS ()
22     end ← timestamp.getNext()
23     VALIDATE(end)
24     WRITEBACK()
25     start ← ∞
26     RELEASELOCKS(end)
       // Quiescence
27     for each tx in transactions do
28         while tx.start ≤ end do wait
```

## 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 is not privatization safe:

- LSA – The write-through version of the LSA algorithm [6]. This is a check-twice algorithm with extensible timestamps.
- LSA-Tick – LSA, but using `rdtscp` in place of a global shared counter.
- Patient – A write-back version of LSA [24], 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, but does not have extensible timestamps [4]. 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 algorithm that does not use orecs [3].
- OrecELA – A variant of the Detlefs algorithm [12, 23], which uses check-once orecs and extensible timestamps.
- ELA-Tick-1 – A version of Algorithm 7 using check-once orecs.
- ELA-Tick-2 – A version of Algorithm 7 using check-twice orecs.

All algorithms were implemented within the RSTM framework [21], 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 5 trials. We evaluated STM algorithms on targeted microbenchmarks from the RSTM suite, and also measured their performance on the STAMP benchmarks [16]. As in prior work, we omitted Bayes and Yada from the evaluation: Bayes exhibits nondeterministic behavior, and the downloadable Yada crashes.

### 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 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. Since the data structure is pre-filled with half of the keys in the range, this results in 50% of transactions performing update operations.

On the single-chip machine, the algorithms split into four categories. The most scalable algorithms are those that do not have any bottlenecks in the STM implementation: the non-privatization-safe orec algorithms. Since the benchmark has virtually no aborts, LSA-Tick has the least overhead. This is expected for an algorithm with 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 single-check orecs, and thus requires more ordering 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. 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 of our `rdtscp`-based algorithms performs better: for small writer transactions, parallel writeback is more important. 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 inter-chip communication. Our operating system 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. 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, and thus the `rdtscp`-based privatization-safe algorithms can perform almost as well as these unsafe algorithms. Finally, writer serialization on a multi-chip system is particularly costly, resulting in OrecELA and NOrec both failing to scale.

On the opposite end of the spectrum, Figures 1(b) and 1(d) present the performance of these algorithms on a red-black tree. 80% of transactions perform lookups, with the remaining transactions split equally between insert and remove operations. Since the data structure is pre-populated with half of the keys in a 20-bit range, 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.

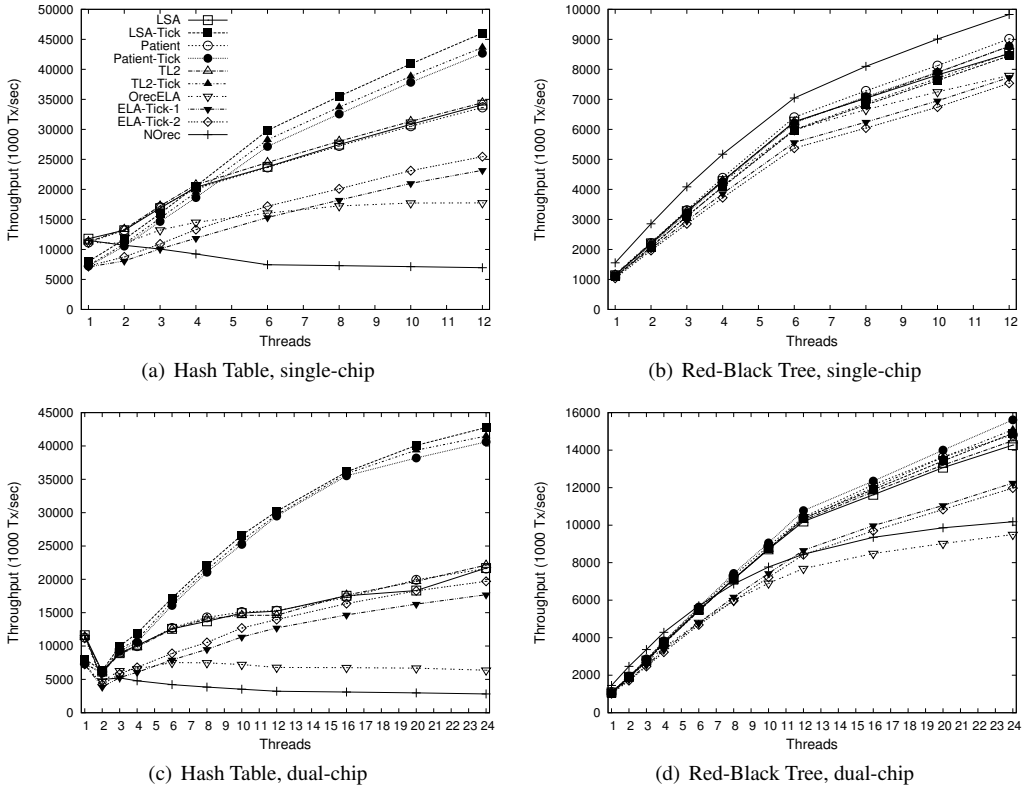
This workload nullifies the benefits of using `rdtscp`: our "Tick" algorithms require writers to validate at commit time while their counterparts need not; the cost of quiescence is higher, since committing writers must wait on long-running transactions to validate; and shared memory counters are not 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 so severe, and at high thread counts we see the privatization-safe `rdtscp` algorithms outperforming OrecELA 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 benchmark to count cycles spent in the quiescence operation, as well as cycles in quiescence spent specifically waiting for a thread to validate. For the hashtable, most of the overhead of quiescence came from cache misses, not waiting. For the tree, the overhead was mostly due to legitimate waiting, not cache misses. The implication is that for workloads with large transactions and few conflicts, quiescence will be a significant overhead.

### 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; Figures 4 and 5 present results for the dual-chip systems.

***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




(a) Hash Table, single-chip

(b) Red-Black Tree, single-chip

(c) Hash Table, dual-chip

(d) Red-Black Tree, dual-chip

Figure 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.

`rdtscp` has no noticeable effect on performance for either platform.

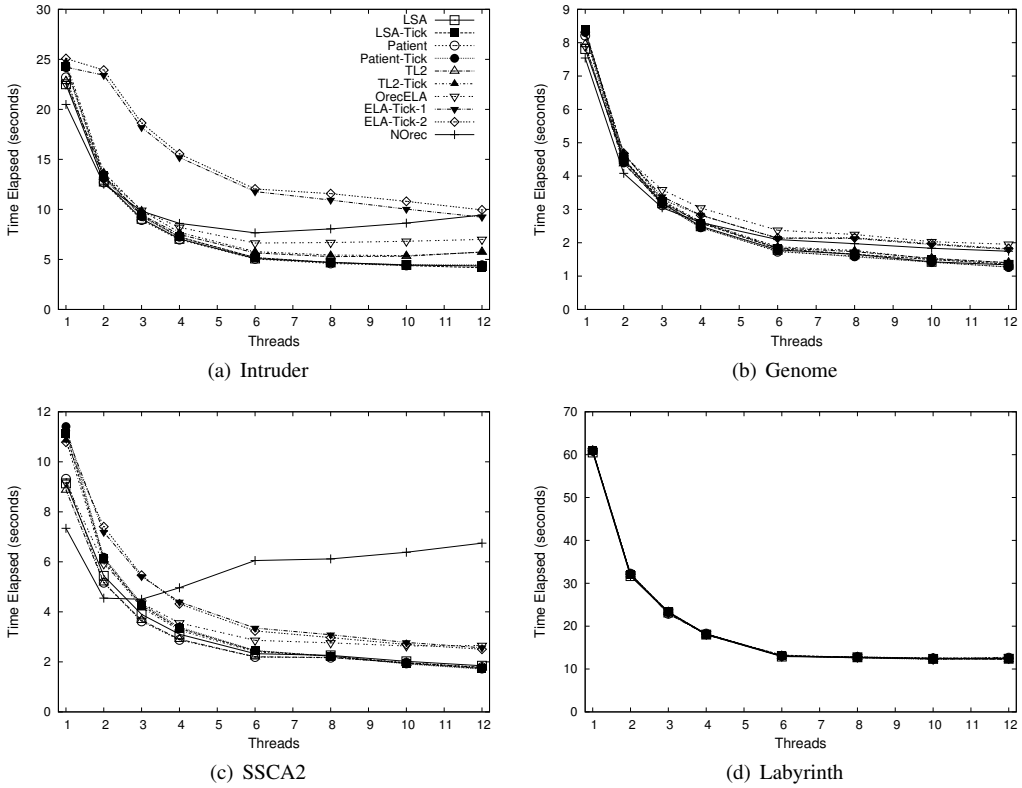
***Genome*** Genome performance is dominated by a large read-only phase, and transactions in general do not exhibit many conflicts. Consequently, on both the single and dual-chip systems, all algorithms perform at roughly the same level. The only differentiation we see is that privatization-safe algorithms 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, since this mechanism causes significant coherence traffic.

***SSCA2*** In many regards, SSCA2 is modeled by our Hashtable microbenchmark: all transactions perform writes, and transactions are frequent and 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 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. The size of these transactions serves as a buffer to minimize the overhead from shared memory bottlenecks. Furthermore, since transactions conflict rarely, 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. As a result, we see comparable performance for all unsafe 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 [2] upon which it is based, Labyrinth conflates memory speculation and control flow speculation: In Labyrinth transactions, a quadratic number of uninstrumented (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 transaction *restart* when the non-




(a) Intruder (b) Genome

(c) SSCA2 (d) Labyrinth

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

transactional 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 thus restored the Lee-TM form to the Labyrinth application for our tests. This change decouples control-flow speculation from transactional speculation on memory accesses, but 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 paper, we explored the role that x86 hardware cycle counters can play in reducing the overhead of STM. 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 performed well, while in other cases our algorithms performed on par with their non-Tick equivalents. When privatization is required, however, the use of cycle counters prevents the use of some key 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 use of an atomic add appears to provide the "write before timestamp access" ordering that our single-check orecs require, but it is less clear how timestamp accesses and subsequent memory loads are ordered. In particular, we placed `LFENCE` instructions between these accesses, but the processor specification is not clear as to whether these fences perform as we intended. Surprisingly, we did not observe any change in program behavior when these fences were *omitted*. 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 [20] viable for unmanaged languages, or at least provide an incentive for a 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.

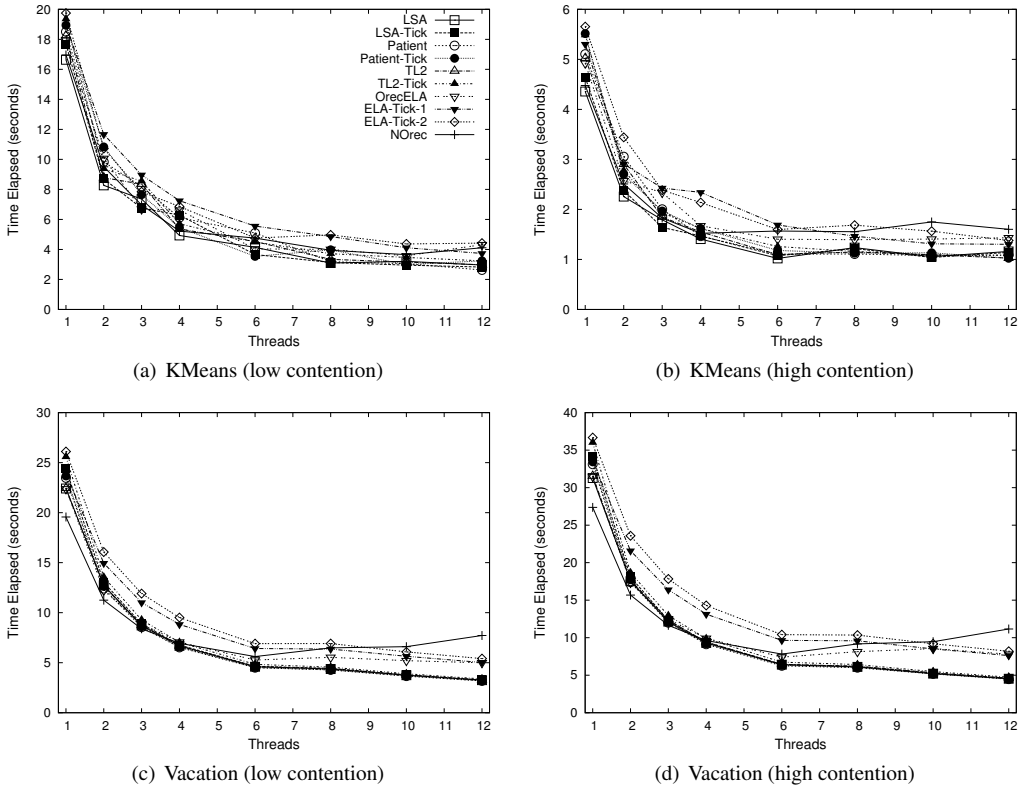
## Acknowledgments

We thank Ravi Rajwar, Stephan Diestelhorst, and Dave Dice for their advice during the conduct of this research. We are also grateful to the anonymous reviewers for their many helpful suggestions.

## References

[1] A.-R. Adl-Tabatabai and T. Shpeisman (Eds.). Draft Specification of Transactional Language Constructs for C++, Aug. 2009. Version 1.0, `http://software.intel.com/file/21569`.

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




(a) KMeans (low contention)

(b) KMeans (high contention)

(c) Vacation (low contention)

(d) Vacation (high contention)

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


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

[4] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[5] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching Transactional Memory. In *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

[6] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[7] K. Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, Sept. 2003.

[8] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[9] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010.

[10] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[11] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corp., 325462-044us edition, Aug. 2012.

[12] V. Marathe and M. Moir. Toward High Performance Nonblocking Software Transactional Memory . In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[13] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[14] V. J. Marathe, M. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[15] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

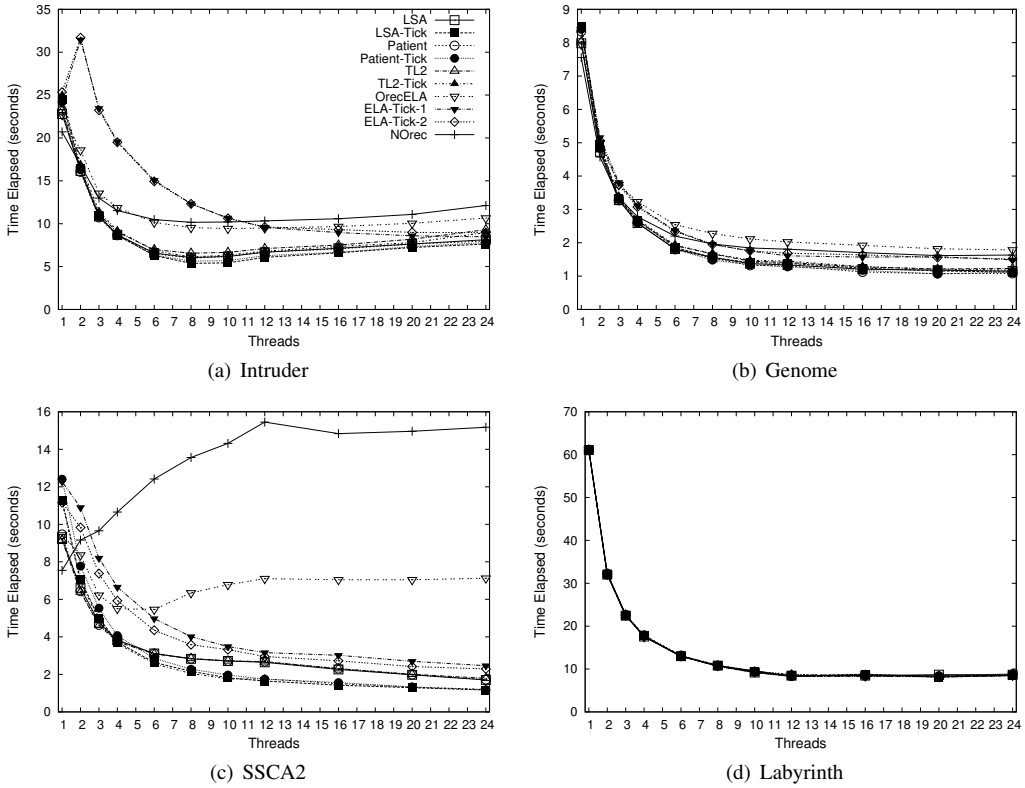
[16] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[17] T. Riegel, C. Fetzer, and P. Felber. Snapshot Isolation for Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.

[18] T. Riegel, C. Fetzer, and P. Felber. Time-Based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, California, June 2007.

[19] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[20] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 2007 ACM Con-*




(a) Intruder (b) Genome

(c) SSCA2 (d) Labyrinth

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

*ference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[21] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

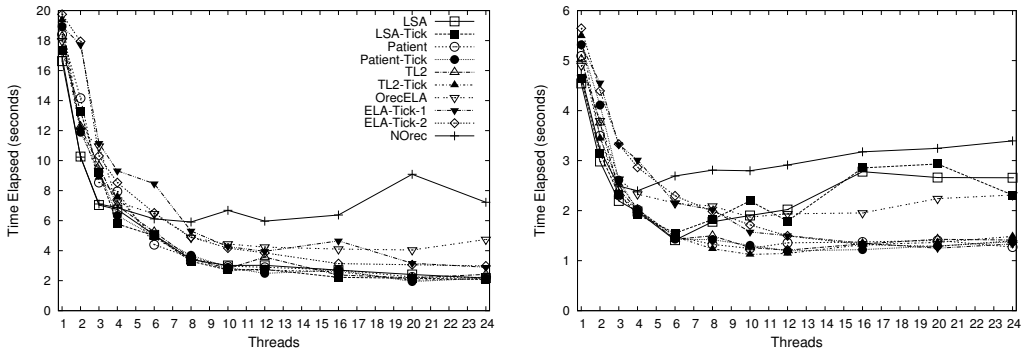
[22] M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[23] M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[24] M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[25] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

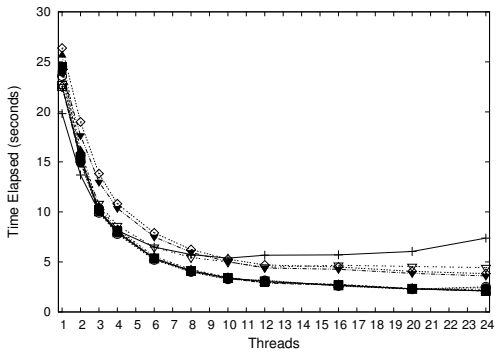
[26] R. Zhang, Z. Budimlic, and W. N. Scherer III. Commit Phase in Timestamp-based STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.



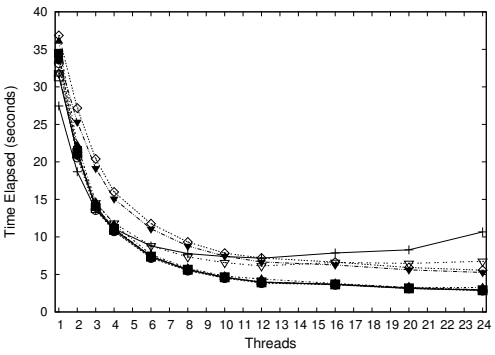

(a) KMeans (low contention)




(b) KMeans (high contention)




(c) Vacation (low contention)




(d) Vacation (high contention)

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