2015

Accelerating Transactional Memory by Exploiting Platform Specificity

Wenjia Ruan
Lehigh University

Accelerating Transactional Memory by Exploiting Platform Specificity

by

Wenjia Ruan

A Dissertation

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Computer Science

Lehigh University

August, 2015

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Wenjia Ruan
Accelerating Transactional Memory by Exploiting Platform Specificity

---
Date

Prof. Michael Spear, Dissertation Director, Chair
(Must Sign with Blue Ink)

Accepted Date

Committee Members

Prof. Hank Korth

Prof. Gang Tan

Dr. Justin Gottschlich

Acknowledgements

The life of a Ph.D student is not supposed to be easy. My student life has been fun and challenging, and sometimes even relaxing, but it never has been easy. Without the invaluable all-around guidance from my respectable, genius and extremely kind advisor, advice from my committee members, help from lab-mates and friends and unlimited support from my parents, I never would have been able to finish this dissertation.

I would like to express my deepest and most sincere gratitude to my advisor, Prof. Michael Spear, for his excellent guidance, caring, patience and complete empathy for his students. I still remember the first meeting we had, in which I expressed my interest in joining his group and frankly confessed ignorance of this research area. He started helping me develop my research with a hundred percent of patience, and that patience continued ever since. During the past few years, I knew whom I could turn to when I encountered research bottlenecks. No matter whether they were big picture confusions or very detailed debugging problems, Prof. Spear always had an open door for discussions and always provided fruitful insights during each meeting. He has not only been supportive about research, but has always been understanding of the ups and downs of his students. He has been generous with vacation time and encouraged taking time away from his lab for beneficial internships.

I also would like to thank my committee members, Prof. Gang Tan, Prof. Hank Korth and Dr. Justin Gottschlich, for sharing their precious time and insightful suggestions regarding my Ph.D proposal, general exam and dissertation. Special thanks go to Behnam Robatmili, Pablo Montesinos and Calin Cascaval from Qualcomm Research, with whom I worked with for an internship. Because of them I experienced an enlightened and joyful summer collaborating on research, and I will be fortunate enough to work with them for my first job after my Ph.D program graduation.

Infinite thanks go to my Lehigh lab-mates and friends, especially Yujie Liu, Mengtao Sun, Le Zhao and Sambhawa Priya. I received a wide spectrum of help from them, from research problems to personal life. I would not have made it through my program to today without our countless discussions together, and I feel blessed to have had them along on this journey.

Last but not the least, I would like to thank my parents, especially my mother Yunxia Chen, who always has had faith in me and always has had my back unconditionally.

Much of the work presented in this dissertation was supported in part by the National Science Foundation under grants CNS-1016828, CCF-1218530, and CAREER-1253362. Any opinions, findings, conclusions or recommendations expressed in this material are my own and those of my coauthors and do not necessarily reflect the views of the National Science Foundation.

Dedicated to my family, who gave me love, help and strength.

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Transactional Memory (TM) is one of the most promising alternatives to lock-based concurrency, but there still remain obstacles that keep TM from being utilized in the real world. Performance, in terms of high scalability and low latency, is always one of the most important keys to general purpose usage. While most of the research in this area focuses on improving a specific single TM implementation and some default platform (a certain operating system, compiler and/or processor), little has been conducted on improving performance more generally, and across platforms.

We found that by utilizing platform specificity, we could gain tremendous performance improvement and avoid unnecessary costs due to false assumptions of platform properties, on not only a single TM implementation, but many. In this dissertation, we will present our findings in four sections: 1) we discover and quantify hidden costs from inappropriate compiler instrumentations, and provide suggestions and solutions; 2) we boost a set of mainstream timestamp-based TM implementations with the x86-specific hardware cycle counter; 3) we explore compiler opportunities to reduce the transaction abort rate, by reordering read-modify-write operations — the whole technique can be applied to all TM implementations, and could be more effective with some help from compilers; and 4) we coordinate the state-of-the-art Intel Haswell TSX hardware TM with a software TM "Cohorts", and develop a safe and flexible Hybrid TM, "HyCo", to be our final performance boost in this dissertation.

The impact of our research extends beyond Transactional Memory, to broad areas

of concurrent programming. Some of our solutions and discussions, such as the synchronization between accesses of the hardware cycle counter and memory loads and stores, can be utilized to boost concurrent data structures and many timestamp-based systems and applications. Others, such as discussions of compiler instrumentation costs and reordering opportunities, provide additional insights to compiler designers. Our findings show that platform specificity must be taken into consideration to achieve peak performance.

Chapter 1

Introduction

Over twenty years ago, concurrent programming was a cutting edge area in computer science. People were excited to finally welcome their first personal computers, with the CPU frequency of a little over 50Hz, and a single core. Concurrent programming was therefore seen to be surreal, only applied to scientific or extremely high-performance computing applications. More importantly, the traditional way to write scalable concurrent programs with lock-based synchronization was widely considered to be time consuming and error prone. In 1993, while most people were excited about the latest sequential video game DOOM or the Microsoft Minesweeper, Herlihy and Moss [36] came up with the idea of the first hardware Transactional Memory (TM) to help programmers write high quality concurrent programs.

As of right now, in mid 2015, the mainstream specification for a personal device, including a variety of mobile devices, would include a CPU of at least two cores, some with hyper-threading, which means four independent threads could be exploited to cope with a single task simultaneously. So before we introduce the idea of Transactional Memory, we will first examine the necessities and pitfalls of concurrent programming.

An easy introductory example is interactions with a hash table. A hash table,

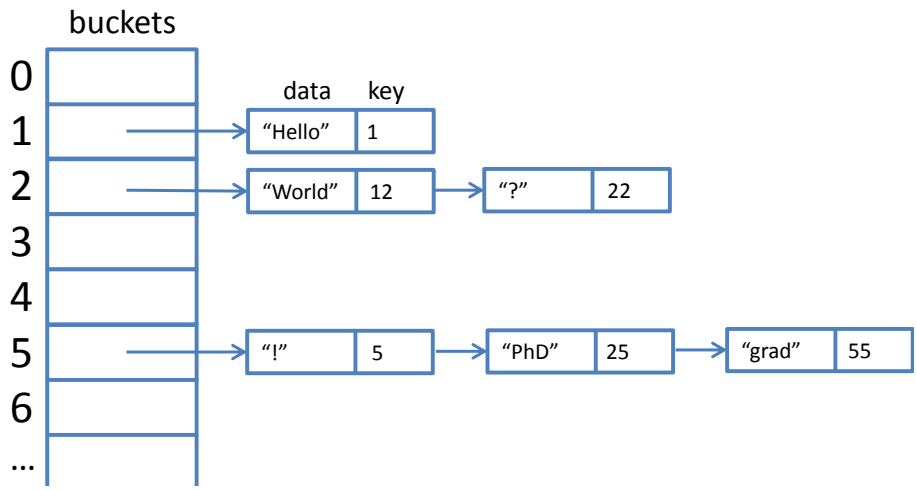


Figure 1.1: Example of a Hash Table

shown in Figure 1.1, is a data structure that stores a set of items. Each datum is paired with a "key". A hash table uses the "key" and a hash function to calculate an index into an array of buckets, and find the corresponding datum. The implementation of hash tables is usually straightforward: the buckets are an array of the heads of linked lists.

Accessing such a hash table, typically including operations of `insert`, `lookup` and `remove` of an element, is not difficult in sequential code. Now consider a situation in which a large number of `insert` operations are required at the initializing phase of an application. With multi-threading available, it would save much time to execute concurrent `insert`s, as each `insert` may touch a different bucket and therefore not interfere with others. However, that being said, it is still possible for more than one `insert` to have a *data race* on the same location: two threads may try to insert elements into the same bucket slot at the same time, and may result in an unpredicted execution ordering, such as that shown in Figure 1.2. On lines 3 and 4, threads 1 and 2 both read head as pointing to the original head (data "World"). However, thread 1 and 2 are interleaved, and in the end the `element1` is lost due to the redirection of `buckets[2]` to `element2` (line 7) and the latter pointing to the old linked list head

```
          Thread 1                |          Thread 2
            ...                   |            ...
1  int key = element1.key;        |   int key = element2.key;
2  idx = hash(key);  // gets 2    |   idx = hash(key);   // also gets 2
                                  |
3  node* head = buckets[idx];     |
4                                 |   node* head = buckets[idx];
5  buckets[idx] = &element1;      |
6  element1.next = head;          |
7                                 |   buckets[idx] = &element2;
8                                 |   element2.next = head;
```

Figure 1.2: Data race of two insertion operations on a Hash Table

(line 8). What makes things worse is that the execution ordering is nondeterministic, therefore the result is unpredictable. To fix it, we say that there is a data race on location `bucket[2]`, and from line 3 to 6 of thread 1 (or line 4 to 8 of thread 2) is a *critical section*.

A *critical section* is a piece of code that accesses shared objects that should not be concurrently accessed by more than one thread of execution without protection of some synchronization mechanism.

The traditional way to synchronize threads is to use locks to protect critical sections. There are basically two kinds of lock-based synchronization: coarse-grained and fine-grained. A coarse-grained lock is a single lock that protects many shared objects. It is the simplest way to synchronize threads. Usually the entrance of the critical section requires a lock acquisition, and the exit of the critical section releases the same lock. But as a result, coarse-grained locking lacks a performance benefit due to a reduction in concurrency: each thread is mutually exclusive to each other when they are executing in critical sections. (Note that in the scope of concurrent programming, the "performance" usually contains two criteria: *latency* and *scalability*. The former typically refers to single thread overhead, while the latter focuses on the ability of a system to improve the throughput of a certain workload by in-

creasing the thread count.) On the other hand, fine-grained locking can be good for scalability. The mechanism requires more careful design, which allows multiple locks, each guarding a single shared object shared object. However, fine-grained locking is error-prone, introducing deadlock (two or more competing threads are waiting for each other to finish), livelock (states of threads may keep changing but without forward progress), priority inversion (high priority thread is dependent upon a lock held by a low priority thread), etc. Keep in mind that it is also very difficult to debug a concurrent program that has fine-grained locking, as more locks may provide nested locking bugs. This also results in difficulty in maintening large-scale applications.

1.1 Transactional Memory

Transactional Memory (TM), proposed by Herlihy and Moss [36], is a concurrency control mechanism that provides *atomic* blocks to protect critical sections, in which a group of load and store instructions appear to execute without interruptions as if a single instruction. Unlike lock-based mechanisms, programmers writing transactional code only need to specify *what* blocks of code should be atomically executed, not *how*, and the atomicity will be achieved by the underlying TM implementation. In other words, an ideal transactional memory implementation should offer the ease of use of coarse-grained locking, and the good performance of fine-grained locking.

Although Herlihy and Moss first introduced TM in a hardware prototype, hardware support for TM did not come to mainstream processors easily. Shavit and Touitou [74] proposed a software version of TM in 1995. With the absence of hardware support for TM for many years, research on Software Transactional Memory (STM) has been well developed [21, 35, 25, 22, 71, 91, 62, 80, 84, 2, 21, 62, 89]. Since 2012 however, hardware support for TM has come to mainstream processors thanks to IBM [41, 90] and Intel [40], which encouraged the development of another

set of TM implementations that contain both hardware transactions and software transactions. They are called Hybrid TM. No matter what the implementation is, the life cycle for a transaction should start with the beginning of a critical section (atomic block), and end with either a) a successful commit of all the updates to the memory (visible to other threads) atomically; or b) an abort to discard all updates as if the transaction never happened. In a typicial implementation, transactions can run concurrently if they do not conflict with each other. A conflict happens when two concurrent transactions access the same memory location with one of them being a write. The TM system keeps track of the read and written locations (known as read/write sets) of transactions in order to detect such conflicts, and automatically enforces an ordering among conflicting transactions. This tracking of the conflicts is named "conflict detection", and is one of the major differences between software and hardware TMs.

Hardware Transactional Memory (HTM) often uses cache coherence protocols to detect conflicts when a transaction is live. For example, Intel TSX [39] has the "requestor-wins" policy, in the sense that a cache line is requested if that location is accessed by a transaction, and once a cache line is required by another hardware thread, the current transaction which holds it will abort. First-generation hardware TM systems as a result carry a number of limitations. Most significantly, these implementations are "best effort" [44], in that they do not guarantee that any transaction attempt will commit. In addition to conflicts, a transaction attempt may fail if it accesses more unique locations than the hardware can support, or if there is an interrupt (e.g., a timer interrupt) during its execution. Consequently, a TM runtime system that wishes to use hardware TM must provide a STM fall-back path. STM has more complex conflict detection mechanisms, which we will describe briefly in Section 1.1.1. Hybrid TM is introduced in Section 1.1.2.

```
1  __transaction_atomic {          TxBegin();
2                                  val = TxRead (&counter);
3    counter++;              ==>   val++;
4                                  TxWrite (&counter, val);
5  }                               TxCommit();
```

Figure 1.3: Example of compiler instrumentation

1.1.1 Software TM Framework

When programmers write transactionalized code, for instance, as shown on the left side of Figure 1.3, the compiler will translate source code to machine instructions that achieve atomic behavior via calls to a TM library, as shown on the right side of Figure 1.3. In this example, each TM function starts with an indicator of "Tx". These functions are typically exposed by every STM implementation. Each of them has its own duty:

- `TxBegin()` marks the beginning of a transaction, creates a checkpoint and initializes per-thread metadata to support tracking conflicts on reads and ensuring atomicity of writes;
- `TxRead(<T>* addr)` reads a location `addr` of type `T`, and ensures the read is consistent with all prior reads and writes performed by the transaction.
- `TxWrite(<T>* addr, <T> val)` writes to a location `addr` with value `val` and type `T`, so that the subsequent reads within the same transaction will see the update, but other transactions will not (yet) see the update.
- `TxCommit()` marks the end of a transaction, and makes writes visible to other transactions if and only if doing so will produce a result indistinguishable from an execution history in which one transaction runs at a time.

The implementation details for each function of a specific STM algorithm vary in several aspects, namely: techniques for detecting conflicts, when to announce con-

Algorithm 1: Algorithm Example: TL2 (an Orec-based, Lazy, Write-back Algorithm)

```
1  transactions : Tx[]                  //[Global] Thread metadata: storage
                                        //          all the thread_local data
   timestamp    : Integer               //[Global] Timestamp: Initially 0
   orecs        : OwnershipRecord[]     //[Global] Orec table: contains locks
                                        //          associated with memory locations
   start        : Integer               //[Thread_local] Tx starting time
   reads/writes : reads[]/writes[]      //[Thread_local] Read/write set
   my_lock      : Integer               //[Thread_local] Tx unique ID: MSB is 1

2  TxBegin()
3      start ← timestamp
4      reads ← writes ← locks ← ∅

5  TxCommit()
6      if writes = ∅ then
7          return
8      AcquireLocks ()
9      end ← AtomicInc(&timestamp, 1)
10     Validate(end)
11     WriteBack()
12     ReleaseLocks(end)

13 TxRead(addr)
14     if addr ∈ writes then
15         return writes[addr]
16     o1 ← orecs[addr].getValue()
17     v ← *addr
18     o2 ← orecs[addr].getValue()
19     if o1 = o2 and o2 ≤
       start and ¬Locked(o2) then
20         reads ← reads ∪ {addr}
21         return v
22     else Abort()

23 TxWrite(addr, v)
24     writes ← writes ∪ {⟨addr, v⟩}
```

Algorithm 2: Helper Functions for TL2

```
1  ReleaseLocks(end)
2      foreach addr in locks do
3          orecs[addr].releaseTo(end)

4  WriteBack()
5      foreach ⟨addr, v⟩ in writes do
6          *addr ← v

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

13 AcquireLocks()
14     foreach addr in writes do
15         if
           ¬ orecs[addr].acquireIfLEQ(start)
           then
16             Abort ()
17         else locks ← locks ∪ {addr}

18 Abort()
19     foreach addr in locks do
20         orecs[addr].releaseToPrevious()
21     restartTransaction()
```

flicts, and when the actual updates happen. We now present TL2 [21] in Algorithm 1 and 2 as a reference.

We first introduce the concept of ownership records (orecs). An orec can be considered as a special lock. It either stores the identity of the lock holder, or the most recent time at which this orec was unlocked. The implementation for an orec is straightforward. For instance, for a 64 bit-wide orec, the most significant bit of the orec is used to indicate whether the remaining 63 bits represent identity of a lock holder or a timestamp.

For algorithms with *orec-based* conflict detection, during each transaction, every location accessed in the memory is mapped to an orec. Usually, in each `TxRead (addr)`, a transaction `T` compares the value of orec `o` of `addr` with `T`'s last consistent time `t`. If "`o` is not locked" and " $o \leq t$", the transaction may continue, because this means location `addr` has not been updated since this transaction started. Otherwise, depending on different algorithm designs, `T` can either abort (as shown in Algorithm 1, line 22), or try to extend `t` by validating all the orecs that have been read and stored up until this point (as shown in Algorithm 16, line 14).

At the same time, TL2 shown in Algorithm 1 is a lazy lock acquisition algorithm. *Eager* and *Lazy* algorithms differ in the time when transaction attempts to acquire locks. Eager TM acquires a lock as soon as a `TxWrite()` is performed; Lazy TM usually waits until commit time to acquire locks (as in Algorithm 1, line 8). Moreover, eager and lazy mechanisms typically affect the choice of when to perform updates to the memory. Eager typically uses *write-through*, where updates are immediately performed in `TxWrite`, and an undo-log is used to save old values in case of rollback; Lazy must use *write-back* in the `TxCommit` with a write-set storing pending writes (Algorithm 1, line 11). In the latter case, each `TxRead()` must first check the write-set to to get the pending value written by the current thread (Algorithm 1, lines 14-15), and then continue with the regular procedure. Note that for eager algorithms,

immediately updating the memory does not mean the new value is visible to other transactions, as the lock associated with the location is not released until commit time.

There is another class of algorithms that use *value-based* conflict detection [18, 63]. They detect conflicts by checking the current value of `addr` and comparing it with the most recent valid value read by transaction `T`. If the two values are different, `T` must abort. This technique does not require specific locks for each location in memory, thus acquiring locks for each writing location is omitted, which results in a lighter `TxWrite` or `TxCommit` instrumentation, depending on when the lock acquisition happens. However, a locking mechanism is still required to synchronize writer transactions (i.e., at commit phase) as only one writer at a time is permitted to perform write-back.

A third class of algorithms, such as RingSTM [85] and InvalSTM [27], use Bloom filters to perform intersections between read and write sets to detect conflicts. While risking false positives, Bloom filters save memory space and have constant time ($O(1)$) complexity of element insertion and set intersection.

1.1.2 Hybrid TM

The unavoidable logging mechanisms for reads and writes in STM, together with heavily instrumented conflict detection procedures, are the main sources of latency. HTM naturally only has marginal overhead in these areas. It is therefore quite clear that with HTM coming into mainstream processors, exploiting HTM while using STM as the fall-back strategy (Hybrid TM) is a necessary step towards high performance.

The reasons that pure HTM can not be a general solution are that a) it records reads/writes in cache so it only allows limited transaction size in terms of locations that are accessed; and b) it does not guarantee forward progress. For instance, the "requestor-wins" policy can lead to livelock: competing transactions may abort each other repeatedly. Hybrid TM solves both problems by trying to execute transactions

in hardware first, and using a carefully tuned STM implementation as a back-off plan when the hardware path fails.

The coordination between HTM and STM varies among Hybrid TM implementations. For instance, HyTM [19] has a software path that adopts orec-based STM, and a hardware path that monitors associated orecs on every transactional read and write to detect conflicts. Though parallelism increases between STM and HTM, the overhead of such checking per HTM read/write is considerable and increases latency. The opposite example is PhTM [45], which typically allows hardware-only or software-only execution. The extra instrumentation per transaction is constant and only occurs at the beginning and committing phase. However scalability can be an issue as the global synchronization mechanism involves a bottleneck. A compromise between the above two implementation is Hybrid NOrec [17]. The main property of software NOrec is that each location does not have an associated orec, and the synchronization between transactions uses a small constant amount of global metadata. This opens a window for lightly instrumented hardware transactions to concurrently run with software transactions. However Intel Haswell TSX does not support non-transactional reads, which are required for high performance in Hybrid NOrec. In the absence of non-transactional reads, each time a software transaction commits, all in-flight hardware transactions will abort. Invyswell [11] presented a hybrid TM for Haswell's TSX. However it uses the lazy subscription technique introduced by Hybrid NOrec [17], which means hardware transactions do not check for conflicts with software transactions until commit time. Lazy subscription may introduce inconsistent states observed by hardware transactions, which can be benign if the transaction eventually aborts, but can also be dangerous if the unpredicted behavior caused by inconsistent state leads to an erroneous hardware transaction commit [20].

1.2 Dissertation Motivation

With hardware support, the performance of language-level atomic blocks is likely to increase substantially, making them useful for general-purpose programs. Meanwhile, the diversity of computer architecture is increasing, thus general-purpose programs or libraries are likely to have cross-platform versions, which should be adaptive and self-tuning. While most prior research was conducted to improve a single TM implementation, our focus is boosting the performance of TMs in a more general, and cross-platform way.

1.2.1 The Platform Factor

We first define *platform*, in the scope of this dissertation, as a combination of operating system, processor and complier.

Design choices discussed in Section 1.1.1 and Section 1.1.2 affect the performance, and sadly there is not a single design that works best over all platforms and workloads. Among popular algorithms, there are solutions for a general workload and platform, which include TL2 and TinySTM [25], but there are more TM algorithms that are especially good for certain kinds of workload or platform:

- **TML** [86] is similar to a sequence lock, but is a read-parallel STM that works well on read-only dominated workloads. It also can be considered as a single-orec algorithm, as there is only one lock globally synchronizing transactions. Also read barriers in TML are light weight, as there is no logging cost: a transaction `T` will simply abort in read if it detects a writer transaction commits after `T` started. But when writers are frequent, TML does not scale.

- **NOrec** [18] uses value-based conflict detection, and only allows one writer at a time to commit. NOrec performs well when writer transactions are infrequent or transactions are relatively large in terms of locations accessed and running

time, but poorly otherwise. The single global lock to coordinate writers can also be a huge problem for scaling.

- **TLRW** [22] supports visible readers by using readers-writer locks and has no validation cost. However each read needs to acquire the shared readers-writer lock in reader mode, which does not require atomic operation, but still needs a write to a shared cache line. Therefore, transactions may still contend on a cache line even if they do not conflict with each other.

- **WSTM** [30] does not use a timestamp to record the global version number, but instead, each orec has its own individual increasing versions. Thus every `TxRead` has to validate the whole read-set without exception. WSTM has quadratic validation overhead and works well on workloads dominated by frequent and tiny transactions. Also it is immune to multi-chip coherence overhead due to the lack of global timestamp.

- **PathTM** [88] allows a single transaction to have higher priority. That transaction therefore can do in-place updates, and commits faster. This helps improve the program pattern in which one transaction may become a bottleneck and/or when thread count is relatively small. Also the original algorithm does not support changing the prioritized thread, and adding this property deteriorates performance significantly.

By the brief analysis of each algorithm, we observe that a single design choice could make a significant difference, and they have nuanced impact on overall performance. However among past research, much is focused on correctness, semantics, and performance of a TM implementation on a single platform. Less is focused on how platform or different architectural characteristics would have impact on the performance of TM implementations. Based on the latter, we have several observations:

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

Figure 1.4: A transaction that modifies a highly contended variable.

- The underlying TM implementations commonly make some default assumptions applying to all platforms, thus may provide inappropriate instrumentation, which could worsen the overall performance.

- Platform-specific features could improve performance, but have not yet been exploited. For example, many TM implementations need a version number (timestamp) to keep track of in-memory updates. Currently most of the implementations use a global variable to do that job, while in fact on x86 machines, the tick counter can be a better candidate.

- Aborting a transaction is expensive. Repeated aborts due to heavy contention on certain memory locations are especially painful. An example pattern that could result in repeated aborts, but commonly exists in parallel applications is shown in Figure 1.4. This affects all platforms, and current contention managers are not enough to resolve the problem. However, we find that the compiler has the ability to discover the pattern and makes possible changes to TM instrumentation to reduce the abort rate.

- Since real-life HTM has come, and that HTM has much less overhead on logging and conflict detection mechanisms but not enough guarantees for forward progress and pathology avoidance, developing Hybrid TMs with Software Transactional Memory (STM) as backup strategy is necessary. Current proposals for Hybrid TM either lack the performance benefits, or are not safe for general pur-

pose applications. We propose and have implemented a new Hybrid TM based on Intel TSX and within the GCC TM framework, which is the most accessible platform to general users.

Note that the first three observations apply to a set of implementations, and may not even be limited to Transactional Memory. The last one is more platform-and-area-specific, but represents the current and future of transactional memory.

1.3 Contributions

Recall that there are two criteria to evaluate performance of a STM algorithm: *latency* and *scalability*. The former usually refers to single thread overhead, while the latter focuses on the ability of a STM to improve the throughput (number of transactions per unit time) of a certain workload by increasing the thread count.

As discussed previously, research directions on creating faster STM algorithms in recent years mostly focus on designing for a specific platform or workload. While this is valid and helpful, especially when we have adaptivity [80], there are system-level issues that require attention if peak performance is to be achieved. Based on the previous four observations, our research contributes to these areas that potentially affect TM (and even the broad area of concurrent programming) performance:

1. **Reduce the Cost of Instrumentation:**

   Accessing Thread Local Storage (TLS): Multi-threaded programs often require a field for thread-local metadata storage. This field is not shared with other threads, but may be accessed by the owner thread itself frequently throughout the entire execution cycle. Accessing TLS is not free and can be very expensive on certain platforms. This is often overlooked by compilers and/or programmers when considering sources of overhead. We examine the

cost of TLS on multiple platforms, and make suggestions to programmers and compiler designers.

Accessing the TM Library: In TM systems, switching the underlying TM implementation in the runtime is necessary for performance, supporting I/O, and guaranteeing progress. There are several ways to access the library, and not surprisingly, it makes a noticeable difference to choose the appropriate way to do so on different platforms. We examine multiple ways of accessing the TM library on four platforms, present the results and make suggestions.

Cost of Fences: Most TM research focuses on the x86 and SPARC processors. However for processors that have relaxed memory consistency, such as ARM and POWER, instruction ordering in `TxRead` is much more expensive to preserve. For instance, in Algorithm 1, a strict ordering in `TxRead` requires two memory fences in between line 16 & 17 and line 17 & 18, if running on processors like ARM, as two independent reads are reorder-able to those processors. We designed and implemented a new TM algorithm named "Cohorts", in which no memory fences are needed during transactional reads and writes.

These issues affect latency, and can also be applied to other cross-platform systems.

2. **Reduce the Cost of Timestamps**:

   Most timestamp-based algorithms have a bottleneck on the global timestamp counter. It is accessed frequently and often requires atomic operations. Particularly on multi-chip machines, the bottleneck can become a dominating factor that limits scalability. The x86 tick counter has much lower overhead to read, and it increments itself automatically. It is a natural timestamp. Unfortunately replacing a global counter timestamp with the tick counter is not entirely straightforward, as accessing the tick counter does not have sufficient

ordering properties. We analyzed the properties of the tick counter and proposed and implemented tick-counter based TM algorithms, which effectively improved performance. We are also glad to notice that the use of tick counter is not limited to TM implementations.

3. **Shrink the Size of Conflict Window**:

   In any TM algorithm, a transaction T starts to become vulnerable to conflicts when it accesses a shared location X. A conflict is materialized when the location X is written by another transaction before T finishes. We call this time span the *conflict window* of transaction T on location X, which starts from the time when T accesses X to the time when T commits or aborts. Clearly, shrinking the sizes of conflict windows decreases the chance of conflicts, and thus improves scalability. By applying static/dynamic analysis, we can perform instruction reordering within transactions to downsize the conflict window of certain heavily contended locations (i.e. by deferring their first access). We applied this technique to several TM algorithms under the framework of GCC TM, and the experimental results show that shrinking the size of conflict windows does reduce the abort rate and therefore improve scalability when the pattern shown in Figure 1.4 is detected. The technique does not hurt performance otherwise.

4. **Reduce the Cost of Logging**:

   Most STM implementations keep track of the memory locations accessed by a transaction in thread-local log data structures, known as the read and write sets. Logging is often necessary for achieving fine-grained conflict detection among transactions. Some STM implementations avoid the overhead of logging by using various forms of mutual exclusion, which leads to less parallelism. However HTM does not have such logging cost, as the conflict detection is

achieved via cache coherence protocols. HTM does have much lower latency, but needs a proper fall back STM to guarantee forward progress without losing scalability. Current Hybrid TMs either lack a performance benefit due to too much serialization in the software path, or are not safe because of the hazards introduced by lazy subscription. We designed and implemented a new Hybrid TM named "HyCo", which supports parallelism in the software path and does not use lazy subscription.

1.4 Organization

In the rest of this dissertation, we discuss our progress in developing systems and algorithms that improve the performance of transactionalized programs. The rest of the dissertation is organized as follows:

- In Chapter 2, we explore costs of platform-specific instrumentation, including the cost of accessing TLS, the cost of accessing a TM library, and the cost of memory fences. For each of the potential costs, we conduct stress tests on SPARC, x86 and ARM processors, paired with Solaris, Linux and Android operating systems respectively. We give suggestions and alternative options to compiler designers and/or programmers to avoid unnecessary costs due to inappropriate instrumentation. In addition, to reduce the cost of memory fences, we propose and implement a new TM algorithm, "Cohorts", which requires no memory fences on individual transactional reads.

- In Chapter 3, we explore the use of the x86 cycle counter (tick counter) as the timestamp for a set of TM implementations. We discuss the properties of the tick counter, show possible data races that could be introduced by using the tick counter without necessary memory guards, and present a correct solution. In the evaluation section of this chapter, we show that tick counter based imple-

mentations boosted performance noticeably, and introduced negligible overhead even in the worst circumstances.

- In Chapter 4, we introduce a technique that can be used to reduce the transaction abort rate, by delaying read-modify-write (RMW) operations. With static analysis, the technique can be implemented under the hood, inside each TM implementation, without requiring effort from programmers. We present several algorithms with this technique embedded, and suggest feasible optimizations. We also discuss language level semantics which could be violated by the reordering of RMWs, and present a solution to restore safety. Performance evaluation shows improvement on benchmarks that have RMW bottlenecks.

- In Chapter 5, we present a Hybrid TM algorithm, "HyCo", that uses "Cohorts" as the fall-back software slow path and Intel Haswell TSX as the fast hardware transactional path. To the best of our knowledge, HyCo is the the best performing Hybrid TM that is safe (in respect of opacity), and supports mode switching (including pure software, pure hardware, half-and-half, and irrevocable transactions) and contention managements. Detailed algorithm descriptions are given in this chapter and thorough evaluations are presented to demonstrate the performance benefit we gain from exploring collaborations between hardware and software TM.

- Finally in Chapter 6, we conclude the dissertation and discuss future works.

Chapter 2

Reducing Platform-specific Instrumentation Costs

In this chapter, we discuss and quantify platform-specific instrumentation costs and present solutions to reduce the costs. The original work was published in "On the Platform Specificity of STM Instrumentation Mechanisms" at 2013 International Symposium on Code Generation and Optimization.

2.1 Introduction

Hardware support for Transactional Memory has arrived in mainstream processors, in the form of extensions in the Intel Haswell microarchitecture [39]. As a result, the performance of language-level atomic blocks is likely to increase substantially, making them useful for general-purpose programs. An open question, however, is what form language support for TM will take. To date, the most promising proposal for unmanaged code is the draft C++ TM specification [4].

Generating code that conforms to the draft C++ TM specification is straightforward, and products from Intel, GCC, and Oracle are already available, as is an extension to LLVM. However, we observe that there are fundamental assumptions

built into each implementation, which can affect performance and maintainability. These assumptions are most dangerous for tools such as GCC and LLVM, which strive to support a diversity of architectures and operating systems. They must balance maintainability with performance: optimizing both the compiler passes and the TM library to each architecture maximizes performance, but introduces a management nightmare; doing anything less has a measurable performance overhead.

In this chapter, we explore the relationship between the compiler, TM runtime library, and performance, by assessing the following dimensions: the cost of thread-local storage (TLS), the manner in which instrumentation is reached, platform preference for certain algorithms, and the requirements that hardware TM and/or *relaxed* transactions [75] might place upon the compiler. While we focus primarily on constant overheads, the findings are significant since these constants are incurred not only on transaction boundaries, but also on every load or store of shared memory within every transaction. A single bad choice can cause a 10% slowdown or worse.

In each case, we identify the most effective solution for each of our target platforms: IA32 CPUs running Linux, SPARC Niagara2 CPUs running Solaris, and Tegra-3 ARM CPUs running Android. As appropriate, we also consider differences between Linux and Mac OSX. Our results include the following:

1. TLS introduces a significant overhead on all but IA32/Linux; on other platforms the transactional compiler should add manual management of per-thread metadata.

2. Fine-grained support for mode-switching within the TM library causes a noticeable slowdown on ARM, necessitating less flexible mechanisms.

3. On ARM, small core counts and the high cost of memory fences both favor a new software TM (STM) algorithm, which we call Cohorts. Cohorts is the first STM algorithm to require only a constant number of memory fences per

transaction.

In Section 2.2, we discuss platform-specific thread-local storage (TLS) costs and alternatives. Section 2.3 focuses on how instrumentation is reached, and shows that different platforms sit at different points on the flexibility/overhead spectrum. Section 2.4 examines how the platform affects the choice of algorithm, and in turn determines what compiler optimizations and transformations are desirable. This section also briefly discusses the requirements that hardware TM and/or irrevocable relaxed transactions place upon the compiler. Section 2.5 summarizes the chapter.

2.2 Per-Thread Metadata Access Costs

STM implementations use per-thread metadata (e.g., "transaction descriptors") to store the set of locations read within a transaction and whatever values are needed to roll back a transaction's writes if it cannot commit. Descriptors must be accessed at transaction begin, at commit, and upon every load and store to shared memory.

There are two possibilities for how a transactional library finds threads' descriptors. The simplest is the approach currently used by the GCC compiler: in every STM library function, the first operation is to access thread-local storage (TLS). While there exists a POSIX library for this purpose (`pthread_getspecific()`), most platforms support a language-level construct for indicating thread-local storage (e.g., the `__thread` modifier). The language-based approach is considered safer and more convenient to program with, since it enforces type checking on TLS variables and unifies the syntax between TLS and normal memory accesses. It is also more amenable to compiler optimization, and expected to be fast.

The second alternative is for the compiler to explicitly manage descriptors. This is the approach taken by the Oracle TM compiler. The function that starts transactions (`TxBegin()`) returns a reference to the thread's descriptor, and then, on every

subsequent shared memory access, the descriptor is passed to the TM library as an additional parameter. Furthermore, when the compiler generates transactional versions of functions that can be called from a transactional context, it changes their signatures to add an extra parameter for the descriptor. Managing these new function signatures invisibly can be a challenge, particularly when dealing with function pointers that are called transactionally and nontransactionally.

Clearly the TLS option is simpler for the compiler. However, on some platforms it is expensive, while on others its cost is negligible. To demonstrate the difference, we modified the open-source RSTM library [80] to allow either convention. We then used a stress-test microbenchmark, in which all transactions repeatedly execute an equal mix of insert and remove operations on a red-black tree storing 8-bit values. There is no work between transactions, and thus the cost of different mechanisms for accessing TLS is exaggerated. We call experiments that use TLS on every library call as "TLS"; "Param" refers to the case where descriptors are managed by compiler-inserted instrumentation. To minimize the variance in "TLS", we structured all STM routines (`TxBegin()`, `TxCommit()`, `TxRead()` and `TxWrite()`) so that the transaction descriptor is accessed exactly once per routine.

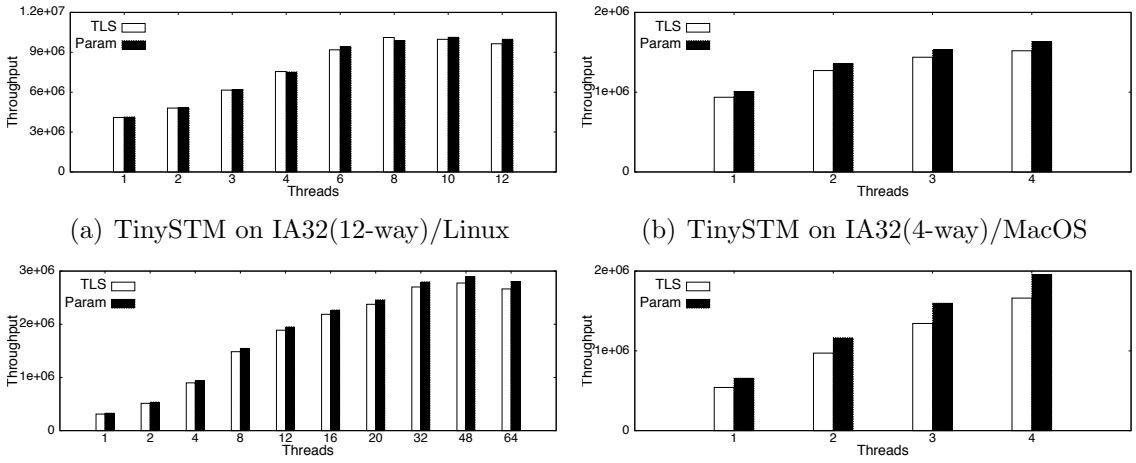
We evaluate the following platforms. For each, we show the best-performing STM algorithm: TinySTM [25] for IA32, TL2 [21] on SPARC, and NOrec [18] on ARM.

1. IA32/Linux: 2.6GHz Intel Xeon X5650 with 6 cores/12 threads, 6GB RAM, Ubuntu Linux 12.04, kernel version 3.2.0-27, GCC 4.7.1.

2. SPARC: 1.165 GHz Sun SPARC Niagara T2000 with 8 cores/64 threads, 32GB RAM, Solaris 10, GCC 4.7.1.

3. ARM: NVIDIA 1.4GHz Tegra3 with 4 cores, 1GB RAM, Android Linux 4.0.3, GCC 4.4.1.

4. IA32/MacOS: 2.66GHz Intel Core i7 M620 with 4 cores, 4GB RAM, MacOS



(a) TinySTM on IA32(12-way)/Linux

(b) TinySTM on IA32(4-way)/MacOS

(c) TL2 on SPARC(64-way)/Solaris

(d) NOrec on ARM(4-way)/Linux

Figure 2.1: The cost of thread-local storage vs. additional function parameters

10.7.2, GCC 4.2.1.

2.2.1 Results

The results of our stress test appear in Figure 2.1. On the IA32/Linux platform, no difference is seen between the approaches, while on IA32/MacOS, we see a noticeable improvement for Param. On the SPARC platform, Param constantly provides about 5% improvement at all thread levels, and on ARM, Param outperforms the default TLS implementation by up to 15%.

On the MacOS platform, there is no OS-level support for **__thread**, and the STM library must call the pthread library to access TLS. When the compiler inserts code to manage descriptors explicitly, all but one pthread call can be eliminated for each transaction.

The remaining differences stem from the platform-specific TLS ABI. On IA32/Linux, the thread local data region is at a constant offset in the binary. Thus, a TLS access can be translated to a simple register-to-register move, in which the TLS address is computed directly from the segment register (%gs). For example, in the following code the TLS pointer is copied to %eax in only one instruction.

```
mov    %gs:0xfffffff0,%eax
```

On SPARC, accessing TLS involves more instructions than on IA32. The initial assembly code consists of 12 instructions, but during linking several constants are known, resulting in a three-instruction sequence:

```
sethi   %hi(0),  %g1
xor     %g1, -4, %g1
ld      [ %g7 + %g1 ], %o0
```

The first two instructions load a 32-bit constant into %g1. The final instruction is specific to the TLS model on SPARC, and loads the TLS offset. Since the Niagara2 pipeline is single-issue in-order, these three instructions are likely to have a higher relative cost than on IA32. At the same time, passing a parameter on SPARC is usually cheap, since we organized the library so that %o0 could hold the descriptor pointer for all function calls within a region of code.

Lastly, on ARM, TLS requires 7 instructions, including a branch to invoke the system ABI:

```
    ldr     r3, [pc, #12]
    push    {lr}
    bl      __aeabi_read_tp
    ldr     r0, [r3, r0]
    andeq   r0, r0, r4, lsl r0

__aeabi_read_tp:
    mvn     r0, #61440
    sub     pc, r0, #31
```

The combination of a relatively simple pipeline with an extra branch results in noticeable overhead, especially since the alternative is extremely cheap. In our microbenchmark, under the Param configuration the compiler essentially reserves one of the 16

general-purpose registers solely for storing the descriptor. Relative to this negligible cost, 7 instructions and a branch are significant.

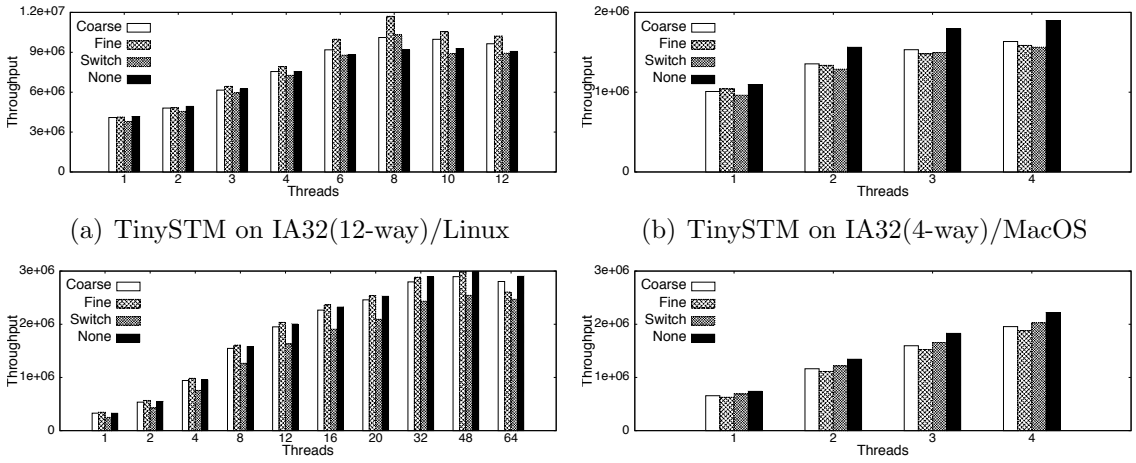
2.3 The Cost of Accessing the TM Library

Under most circumstances, TM libraries are required to provide different instrumentation for different individual transactions. Clearly this has been true in the various adaptive TM systems [62, 45, 80, 67]. However, the requirement is not merely one needed for switching among algorithms. At least a limited form of adaptivity appears to be fundamental.

Consider the draft C++ TM specification [4], which includes *relaxed* transactions [75]. A relaxed transaction is expected to transition between a concurrent mode and a serial mode if the transaction attempts an operation that cannot be rolled back.[1] While this support may be provided by a branch on every load/store of shared memory, it nonetheless represents a transition between two modes of transactional behavior. Hardware TM will almost certainly require some additional adaptivity support [15, 45], at which point full adaptivity support [67, 92] is at least worth considering.

There are three common approaches for supporting adaptivity within a TM implementation. The first, as mentioned above, is to use conditionals (typically a `switch` statement) within the library's transactional read and write functions to globally coordinate all transactions' behavior [67]. The second is to use per-thread function pointers, such that each thread can make fine-grained and nuanced decisions about how to perform its transactional operations [62, 80]. A third option is to use global function pointers, which coordinate all transactions' behavior, but do not incur additional branching overhead in the common case. Of these options, per-thread function

[1] A simple understanding of relaxed transactions is that they allow the programmer to specify that the transaction cannot self-abort, so that irrevocable operations [94] can then be permitted.



(a) TinySTM on IA32(12-way)/Linux

(b) TinySTM on IA32(4-way)/MacOS

(c) TL2 on SPARC(64-way)/Solaris

(d) NOrec on ARM(4-way)/Linux

Figure 2.2: The cost of mechanisms for reaching (adaptive) instrumentation

pointers have been evaluated on the IA32 platform, and shown to have good performance. We are not aware of any other published evaluation of the cost of different mechanisms for accessing an STM library.

In Figure 2.2, we contrast the performance of these options by repeating the red-black tree microbenchmark from Section 2.2. As before, a high incidence of transactions amplifies the impact of differences in the instrumentation mechanisms. We compare four manners in which transactional instrumentation can be achieved:

1. Fine: This curve corresponds to a configuration with per-thread function pointers [62, 80]. "Fine" affords maximum flexibility, since individual transactions can vary their behavior without branching. However, it incurs TLS overhead to locate function pointers.

2. Coarse: This curve represents global function pointers. On architectures with inexpensive indirect branches, this mechanism should be fast. It also avoids TLS overhead and prevents branching in the common case where adaptivity is not occurring.

3. Switch: In this configuration, there is no TLS or indirect call overhead, but every

read and write must perform a branch to select the right instrumentation. We used a `switch` statement to choose among several dozen options, and verified that the compiler produced a dense branch table. Note that dynamic branch prediction is expected to succeed in most cases.

4. None: Here, adaptivity is not supported, but overhead should be minimal. Reaching instrumentation does not require TLS, indirection, or branching. This curve should serve as a best-case.

Note that each platform is configured optimally with respect to the previous section: on the Linux/IA32 system, we access metadata via TLS, and otherwise we access it via an extra parameter passed to each function.

2.3.1 Results

As expected, on IA32/Linux fine-grained instrumentation is optimal. Since indirect calls and TLS are inexpensive, this mechanism can even outperform the expected best case of no adaptivity: fine-grained mode switching within a transaction can avoid enough branches to recover more performance than it costs (e.g., by skipping write-set lookups when reading in a transaction that has not yet performed a write [80]). On MacOS, where TLS is expensive, "Coarse" is best: the indirection of function pointers does not incur a noticeable cost, but the avoidance of TLS overhead is preferred. Note, however, that an unrealistic "adaptivity-free" implementation ("None") gives the best performance.

On SPARC, "Fine" again gives the best performance. In this case, we configured the library to pass the descriptor as a parameter, but the function pointers themselves are still accessed via TLS. The key difference here is that while TLS is more expensive than on IA32, the additional registers on SPARC make it easier for the compiler to cache most of the computation for locating function pointers. Thus

fine-grained adaptivity does not introduce much overhead. At the same time, the simple pipeline of the SPARC CPU greatly benefits from the reduction in branches that follows from fine-grained adaptivity. This is borne out both in comparison to coarse adaptivity, which has additional branching within each read, and relative to switch-based adaptivity.

On ARM the best adaptive STM performance comes from the "Switch" mechanism. The high overhead relative to an adaptivity-free option confirms the cost of TLS, which is unavoidable for fine-grained adaptivity. However, the pipeline implementation of ARM makes the switch statement ralatively more efficient than an indirect branch.

2.4 Architectural Impacts on Algorithm Selection and Optimization

Dozens of STM algorithms have been proposed over the last decade, to include those primarily evaluated on older UltraSPARC CPUs [21, 35], various IA32 platforms [25, 22, 71, 91, 62, 80], IBM POWER [84], and SPARC Niagara [22]. A few algorithms employ custom OS features [2, 21], and several rely on specific architectural properties (such as low-overhead memory fences in the TLRW byte-lock mechanism [22], an atomic-or primitive in the Unified STM algorithm [62], or more generally the fact that compare-and-swap (CAS) is cheap on modern IA32 chips [80]). Some trade high latency for fewer bottlenecks and improved scalability. Others are best when the core count is low.

2.4.1 Platform Impact on Optimization

Many STM algorithms benefit from custom compiler optimizations. While the desire for a library-based implementation has reduced interest in fully inlined STM algo-

rithms, such as the original McRT STM [71], still (a) some algorithms with commit-time locking can exploit relaxed consistency checks in `TxRead` [84], and (b) algorithms with in-place update can benefit from compilers that match certain access patterns (read after read, read after write, write after write, and write after read) to custom STM library calls [62].

To illustrate the benefit of a wide interface, consider the case of a write after read in the Unified STM system [62]: If a read of location L dominates a write to L, then the write lock for L can be acquired during the read (making it a "read for write") and the write can be downgraded to a write-after-write. This decreases metadata logging and lowers overhead.

This optimization offers no value to algorithms that use commit-time locking. Furthermore, while early locking is profitable on IA32 [62, 22, 32, 25], the picture may differ on other platforms. If eager locking is too costly, then an STM library will favor a commit-time locking algorithm, and any analysis or optimization for eager locking will go unused. The opposite is true for the optimizations proposed in [84], which have low value for early locking STM algorithms.

To assess whether either set of optimizations is universal, Figure 2.3 presents an evaluation on ARM, SPARC, and IA32 using the STAMP benchmark suite [56]. We consider the algorithms listed in Table 2.1. In all cases, we use the RSTM implementations of the STM algorithms to eliminate artificial implement differences (e.g., all algorithms use the same code to acquire a lock, or to handle memory management). For each benchmark, we present the harmonic mean speedup compared to a single-threaded execution using the Mutex algorithm. We do not consider the MacOS platform, since the algorithms we evaluate do not exploit OS-specific features. We omit two STAMP benchmarks: yada is known to contain bugs, and bayes exhibits wildly nondeterministic behavior. Speedup was computed from the average of 5 trials. With the exception of TLRW on intruder on SPARC, variance among trials was

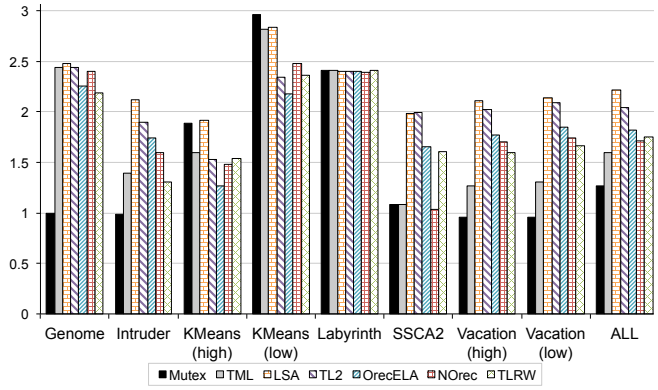
Algorithm	Description
Mutex	The standard STM baseline: all transactions are protected by a single lock. There is no concurrency among transactions, but latency is minimal.
TML	All transactions are protected by a sequence lock. There is only concurrency among reader transactions, but latency is low. This algorithm reveals the overheads of speculation and instrumentation [18].
LSA	The TinySTM write-through algorithm. This algorithm acquires locks eagerly, modifies memory locations before commit time, and maintains undo logs [25]. LSA uses a table of versioned locks ("orecs") to detect conflicts among transactions. The valid version number (timestamp) for each transaction is extended in each `TxRead` if read-set validation is passed.
TL2	TL2 uses commit-time locking and redo logs, does not extend timestamps, and otherwise closely resembles LSA [21].
OrecELA	This algorithm expands upon LSA and TL2 to offer stronger language-level semantics ("ELA" semantics in the taxonomy of Menon et al. [55]). In practical terms, OrecELA is "privatization safe".
NOrec	NOrec does not maintain per-location metadata, but rather tracks conflicts via values. Writers cannot commit in parallel, but per-access instrumentation is typically lower than orec-based algorithms [18].
TLRW	TLRW uses an array of reader/writer bytelocks. Transactions lock every location they read, but do not validate reads at commit time [22].

Table 2.1: Representative STM algorithms.

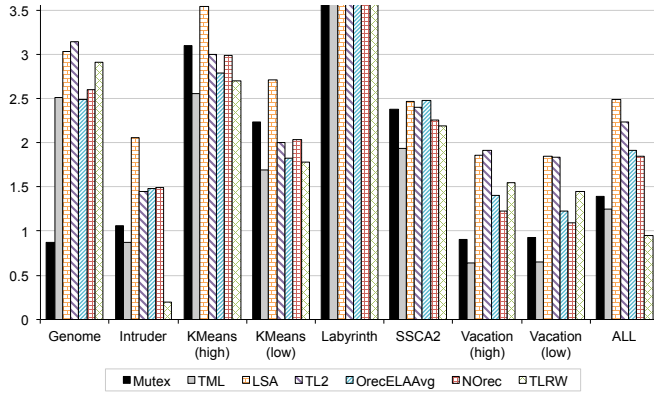
low. Since this lone exception always performed poorly, its higher variance does not affect our conclusions.

Summary of Results

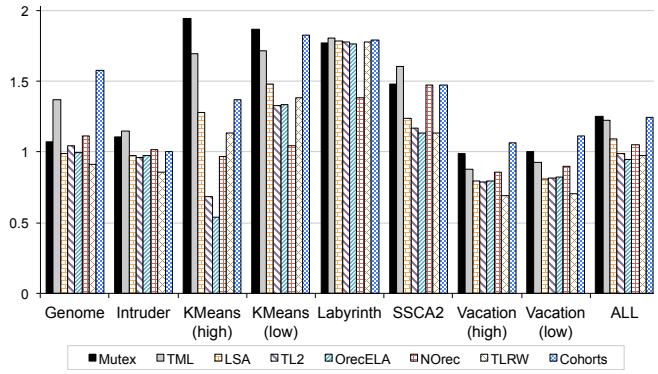
There are two dimensions on which algorithms can be partitioned. First, the LSA, TML, and TLRW algorithms are the only ones in which locks are acquired before commit time. Thus they are the only algorithms that could potentially benefit from Ni's optimizations [62]. Second, TML, OrecELA, and NOrec are the only algorithms that comply with the C++ memory model: TLRW and LSA both allow for a racy program to observe out-of-thin-air reads, due to modifications to memory performed by transactions that might abort [55]. Furthermore, LSA and TL2 are not privatization-safe, which can result in races that appear to be violations of the ordering between transactions and nontransactional code. Thus while LSA, TL2, and TLRW can be used in programs that are proven to not suffer from these problems, they cannot be



(a) Platform: IA32/Linux (12 threads)



(b) Platform: SPARC/Solaris, (64 threads)



(c) Platform: ARM/Linux (4 threads)

Figure 2.3: STAMP speedups vs. single-threaded Mutex

considered truly general-purpose algorithms.

On the IA32 platform, we see that most STM algorithms scale well, with LSA offering the best performance overall, and OrecELA giving the best performance with stronger semantics (though its advantage over NOrec is largely a consequence

of NOrec's poor performance on the frequent small writing transactions of SSCA2). The same is generally true on SPARC, though TL2 and LSA are much closer.

We did not measure either of the extended interfaces to the TM library discussed earlier. Ni's optimizations [62] would offer an improvement to LSA, TML and TLRW, while Spear's optimizations [84] would benefit NOrec, OrecELA, TML, and TL2. Since no single algorithm stands out on all platforms, the most practical approach for a compiler on these platforms is to provide both sets of optimizations. This increases the burden on library designers, who potentially must offer 5 versions of the read function (regular, after-write, before-write, after-read, and relaxed) and 2 versions of the write function (regular and after-write) for each primitive data type (char, short, int, long, float, double).

2.4.2 Tackling the Cost of Fences

On ARM, none of the algorithms from Table 2.1 consistently outperforms Mutex. The latency of individual transactions is simply too high, in large part due to memory fences. Specifically, on LSA and TL2, every load of shared memory requires two fences, to order lock checks that occur before and after the actual load from memory. On NOrec, TML, and OrecELA, a fence is required after the load, before a check of a lock or global counter, but not before the load. On all platforms, TLRW requires a fence on every load and store.

Since ARM is not expected to offer an abundance of cores (e.g., more than 8) within the foreseeable future, we designed a new STM algorithm that reduces memory fence costs by increasing blocking at the begin and commit points of transactions. We call the resulting algorithm "Cohorts", as transactions dynamically form sets that attempt to commit together. To the best of our knowledge, Cohorts is the first STM algorithm designed specifically for processors with relaxed memory consistency. It requires some transactions to block at their start and end points, but eliminates all

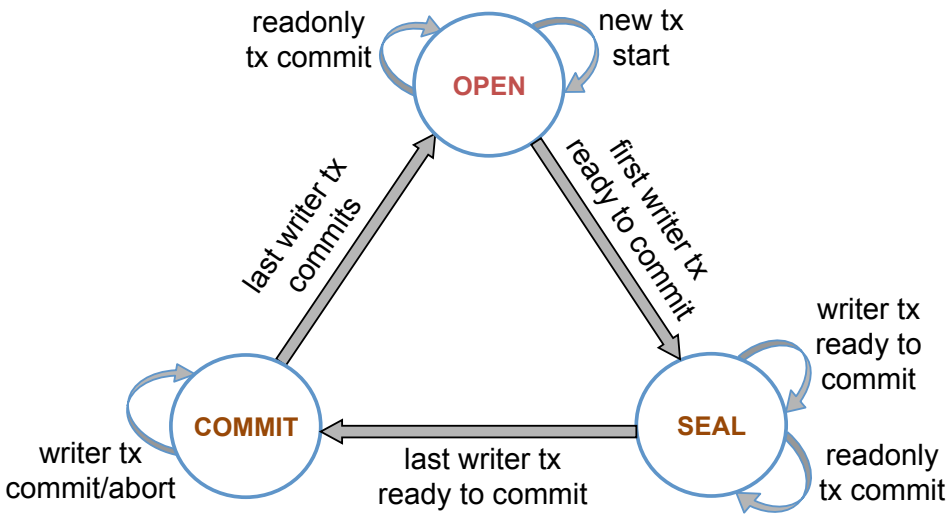


Figure 2.4: State Transitions of a Cohort

but a constant number of fences at transaction boundaries.

Cohorts outperforms all other algorithms on ARM (Figure 2.3c). In additional experiments, we found that when the incidence of transactions is high (e.g., in data structure microbenchmarks), the benefit of Cohorts increases, due to its lower latency. However, Cohorts performance is consistently poor on IA32 and SPARC. From a compilation perspective, Cohorts admits exciting but unique optimization opportunities, which risk complicating the TM compiler/library interface even further since they are specific to a single algorithm that only performs well on a single platform.

The Cohorts Algorithm

Cohorts employs a state machine to control which transactions can commit, and when (Figure 2.4). Initially, there are no transactions running (the OPEN state). As threads begin transactions, the system remains in this state, and if a transaction is read-only, it can commit directly from the OPEN state. However, once a writer is ready to commit, the system transitions to the SEAL state. In effect, the current group of running transactions has become a cohort, in which no transaction can commit writes while another is still running, and which no new transactions can join.

Read-only transactions can continue to commit immediately, but all writers block at their commit point. Eventually, all threads either are not running transactions, or are ready to commit writes. At this point, the cohort moves to the COMMIT state, and transactions validate and commit, one at a time.

Algorithm 3: Cohorts Algorithm

```
1  constant: START, FINISH
2  Global Variables:
3    tail          : Boolean*   // initially nil
     transactions  : Tx[]       // array of transactions
4  Per-transaction Variables:
     spin      : Boolean   // entry in commit queue
     status    : Integer   // transaction status indicator
5    turbo     : Boolean   // go turbo indicator
     writes    : WriteSet  // write set
     reads     : ReadSet   // read set

1  TxBegin()
2    while true do
3      if tail = nil then
4        status ← START; MemoryBarrier
5        if tail = nil then break
6        status ← FINISH
7    turbo ← false; spin ← true
8    writes ← reads ← ∅

9  TxRead(addr)
10   if turbo then
11     WriteBack ()
12     return *addr
13   if addr ∈ writes then return writes[addr]
14   v ← *addr
15   reads ← reads ∪ {⟨addr, v⟩}
16   return v

17 TxWrite(addr, v)
18   if turbo then
19     WriteBack ()
20     *addr ← v
21   else writes ← writes ∪ {⟨addr, v⟩}
```

Algorithms 3 and 4 present the Cohorts algorithm. The state machine is imple-

Algorithm 4: Cohorts Algorithm continued.

```
22 TxCommit()
       // commit a read-only transaction
       // or a (notified) turbo transaction
       // that has performed write back
23     if writes = ∅ then
24         status ← FINISH
25         return

       // commit a read-write transaction:
       // enqueue and wait
26     pred ← AtomicSwap(tail, &spin)
27     status ← FINISH; MemoryBarrier
       // notify the last transaction
       // to go turbo
28     if TxLeft() = 1 then
29         foreach tx in transactions do
30             tx.turbo ← true
31     if pred = nil then
           // first writer waits until
           // everyone reaches commit
32         while TxLeft() ≠ 0 do wait
33     else
           // otherwise wait until signaled
34         while (*pred) do wait
35     if ¬Validate() then
36         FinishCommit()
37         Abort
38     WriteBack()
39     FinishCommit()

40 WriteBack()
       // do write back if buffer is not empty
41     if writes ≠ ∅ then
42         foreach ⟨addr, v⟩ in writes do
43             *addr ← v
44         writes ← ∅

45 Validate()
       // value-based validation
46     foreach ⟨addr, v⟩ in reads do
47         if *addr ≠ v then
48             return false
49     return true

50 FinishCommit()
51     spin ← false
52     if tail = &spin then
53         tail ← nil

54 TxLeft()
       // number of tx left in a cohort
55     counter ← 0
56     foreach tx in transactions do
57         if tx.status = START then
58             counter++
59     return counter
```

mented by two fields: a per-transaction variable *status* (`START` indicates the transaction has entered the cohort but has not yet reached the commit point, and `FINISH` indicates the thread is not in a transaction, blocked in TxBegin(), or ready to commit); and a queue (similar to a CLH queue lock [46]) of writer transactions that are ready to commit. The OPEN state corresponds to the situation when *tail* is null; COMMIT corresponds to the situation in which each transaction's *status* is `FINISH` but *tail* is not null; and SEAL corresponds to a non-null *tail* with some transaction's *status* set to `START`.

During execution, transactions do not check for conflicts. Writes are buffered, and reads simply check the buffer, then record the values they load from memory in the case that the buffer check fails. As in NOrec [18], commit-time validation does not check a lock table, but instead compares actual values in memory. This leads to trivial read (Algorithm 3 lines 13 to 16) and write (line 21) instrumentation and also obviates memory fences while reading and writing.

At the point of transition from SEAL to COMMIT, any arbitrary contention management policy [73] can be used to maximize fairness or prevent starvation (note that livelock is impossible). An appealing alternative, though, is to detect when a sealed cohort has exactly one transaction whose state is still `START`. At that point, the transaction can transition to an irrevocable mode ("turbo mode"), in which it directly accesses memory, without any metadata accesses. In this manner, the duration during which `FINISH` transactions block can be minimized.

Cohorts offers ELA semantics. While a detailed proof is beyond the scope of this chapter, the argument is straightforward. There are two criteria. First, a transaction cannot make data private and then use that data outside of transactions while another (destined-to-abort) transaction has an outstanding reference to that data. Second, a transaction cannot make data private and then use that data unless it is sure that no other transaction is still finalizing its writes to the data (e.g., during its commit phase). In the former case, since writer transaction can commit while another is running, the problem is avoided. In the latter case, writer transactions do not commit in parallel, preventing the problem.

Compiler Optimizations

In Figure 2.3, Cohorts exhibits strong performance on all but the KMeans test, and in that case, Mutex performs well. The problem is quite simple: there is enough nontransactional work that transactions rarely overlap in time. To handle this case,

we extended Cohorts to support programmer-supplied hints. The hint indicates that after a transaction performs some number of loads and stores (ls), it should check if it is the only transaction. If so, it can seal the cohort early and transition to the low-overhead mode. Adding a hint of $ls = 2$ increased Cohorts performance to within 90% of Mutex throughput for KMeans.

A second optimization for Cohorts deals with read-only transactions. Once any transaction begins, no changes to shared memory are possible until all transactions are ready to commit. If a compiler can statically detect that a transaction does not modify shared memory, that transaction does not require *any* instrumentation: logging address/value pairs is not required, since read-only transactions never validate before committing, and write buffer lookups are unnecessary since the write buffer is always empty. While STAMP does not have read-only transactions, our discussion of HTM sheds some insight into the benefit that this optimization provides.

Generality

Although Cohorts outperforms all other algorithms on ARM, Cohorts has poor performance on IA32 and SPARC machines. This comes as no surprise, since Cohorts introduces serialization to avoid expensive memory fences, but fences are not required on IA32 and SPARC. In addition, whenever writer transactions are abundant, Cohorts simply cannot scale to high core counts. As a result, we recommend its use only for small, single-chip multicore machines with relaxed memory consistency. Today this category includes the majority of mobile devices. Whether it remains an important market or not is uncertain, particularly if these platforms add hardware TM support.

2.4.3 The Impact of Hardware-Assisted STM Libraries

Early work on hardware accelerated STM [72, 57, 78] assumed that hardware would provide new instructions to simplify the instrumentation of a fundamentally software-based TM library. The limit point for such instrumentation is captured by the Hybrid or Best Effort approach [15, 68, 17, 45], where the hardware supports native execution of small transactions, and falls back to STM for large transactions or when conflicts are too common.

A critical question is whether simply upgrading a shared library will suffice to reap the benefit of such hardware. The alternative is that these hybrid systems will require transactional code to be cloned and optimized according to several different algorithm-specific strategies. To some degree, this occurs already in the Intel TM compiler, which produces two versions of code: one that makes calls to the TM library on every load/store of shared memory, and one for when assumes the transaction is running in a "Serial Irrevocable" mode [94, 67]. In this mode, loads and stores are not instrumented. Multiple cloning also occurs in Christie's TM stack [15] and the Oracle TM compiler, where the peculiarities of the Rock processor's speculation mechanism require the compiler to produce several versions of a transaction, to include one in which pipeline speculation is constrained via explicit fences.

Recalling our discussion of the current proposal for TM support in C++, the consequences of this problem can already be seen. Early versions of the GNU C++ compiler did not generate multiple transactional clones of a function. Thus, for example, a relaxed transaction that required irrevocability still performed a function call on every load and store. Using the RSTM framework, we now demonstrate the performance cost of this approach. It follows that any custom hardware designed to accelerate TM, up to and including Best-Effort TM, would suffer a similar fate if the compiler did not generate a custom code path for it.

Figure 2.5 presents the performance of the RBTree microbenchmark from Sec-

tion 2.2, using the Mutex runtime (all transactions are protected a single lock). The curve marked "NoInline" uses the adaptive RSTM interface, resulting in the read and write functions depicted below:

```
1 T TxRead (T * addr)
2     return *addr;
3 void TxWrite (T * addr, T val)
4     *addr ← val;
```

Note that in this case, using TLS to access descriptors is optimal: it results in fewer parameters to the function, and avoids a TLS lookup since the descriptor is not used by the instrumentation. The "Inline" variant uses a different clone of the transaction body, which does not require function calls, but instead performs the load or store directly.

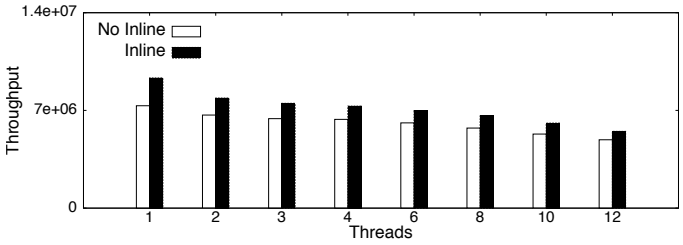
As can be seen in the figure, the lack of clones makes the code significantly slower. In analyzing the assembly code, we found that the reason differs on each platform:

1. On IA32, the fact that register `eax` is both the parameter and return value of the read function impedes instruction scheduling, especially when multiple reads are performed in succession. For "Inline", the operands to successive loads can be kept in different registers.

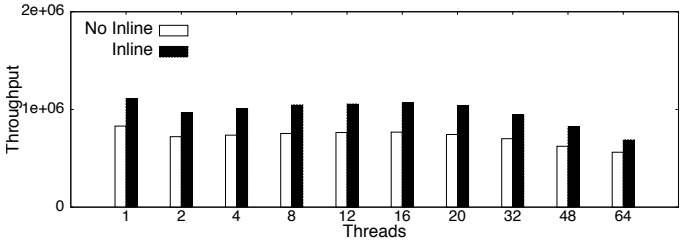
2. On SPARC, function calls are expensive, due to the overhead of register window maintenance.

3. On ARM, parameters and return values are passed in separate registers, but there is still an instruction scheduling cost and overhead due to function calls.

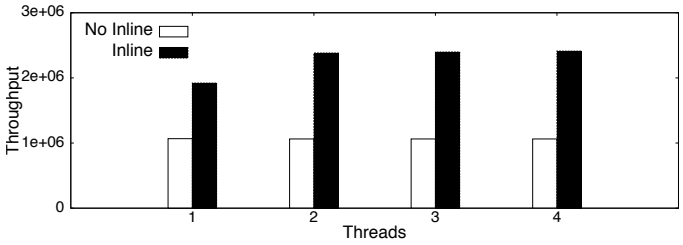
To illustrate the impact on instruction scheduling, the read-only search phase of the RBTree insert function is a while loop that traverses a tree to find a key. On IA32, it grows from 31 instructions to 46, solely on account of the differences in



(a) Platform: IA32/Linux, Algorithm: Mutex



(b) Platform: SPARC/Solaris, Algorithm: Mutex



(c) Platform: ARM/Linux, Algorithm: Mutex

Figure 2.5: Inlined and non-inlined versions of the Mutex algorithm on different platforms. Differences between IA32/Linux and IA32/MacOS are negligible.

parameter movement, register allocation, and instruction scheduling that arise from the increased incidence of function calls. A similar penalty should be expected for any hardware accelerated TM system. Even achieving reasonable performance will require compiler support, in the form of multiple code clones.

2.5 Summary

In this chapter, we showed that there are significant hidden costs that arise from how the compiler interacts with a TM library. Differences in OS and CPU affect the cost of accessing TLS, resulting in some platforms benefiting greatly from the compiler manually managing TM metadata by changing function signatures and passing

references to metadata among functions. Varying costs for both TLS and indirect branches dictate how efficiently a library can adapt among its various internal modes of operation. Furthermore, architectural characteristics can strongly tip the scales in favor of certain algorithms, such that on each platform, a different set of algorithm-specific analyses and optimizations should be employed. Upcoming hardware TM support is likely to make the situation even more complicated.

These themes are not specific to TM. As parallelism becomes more pervasive, and as architectural diversity continues to increase, more libraries will become adaptive and self-tuning. This, in turn, will increase dependence on TLS and adaptive instrumentation, which we have shown to have platform-dependent implementation overheads.

Responsibility for reducing overheads can often be assigned to many parties. Until chip manufacturers reduce the overhead of memory fences, new algorithms like Cohorts will be needed. Similarly, until OS designers reduce TLS overheads, compilers may need more complex transformations to reduce the frequency of TLS access. Ultimately, custom per-algorithm instrumentation may necessitate dynamic recompilation in order to minimize latency for complex parallel software that runs on a diversity of platforms.

Chapter 3

Boosting Timestamp-based TM by Exploiting Hardware Cycle Counters

In this chapter, we utilize the x86 hardware cycle counter to improve performance of timestamp based TM implementations. The original work was published in "Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters" at ACM Transactions on Architecture and Code Optimization, Volume 10 Issue 4, December 2013 Article NO. 40.

3.1 Introduction

Most high-performance STM implementations reduce the common-case overhead of validation by using timestamps. The technique, which was first employed in the LSA [65] and TL2 [21] 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 [26, 35, 49, 50] can be avoided. Since 2006, virtually every single-version STM that uses ownership records has employed a global shared counter [23, 25, 48, 55, 82, 91, 98].

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 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 [56] of the TL2 algorithm [21] included heuristics for reducing the overhead of counter increments. The main observation was 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., is to use the multimedia timer present in some systems in place of a shared memory clock [66]. 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 chapter, 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 micro-architectures, 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 chapter is organized as follows. Section 3.2 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 3.3 extends STM algorithms so that they can employ the x86 `rdtscp` instruction in place of software clocks. Section 3.4 considers techniques for making these `rdtscp`-based algorithms privatization safe. Section 3.5 evaluates our algorithms on single and dual-chip x86 systems, and Section 3.6 summarizes.

3.2 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 committing transaction receives 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.

3.2.1 Software Clocks

Software implementations of a global clock object are deceptively simple. The most common implementation consists of two methods, as depicted in Algorithm 5.

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 above must hold: at the time when

Algorithm 5: A simple software global clock object

```
    ts  : AtomicInteger  // initially 0
1   read()
2   |  return ts
3   getNext()
4   |  return 1 + AtomicInc(ts)
```

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 acquire 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, memory fence instructions are required to provide the necessary ordering.

3.2.2 Hardware 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. Symbol $\rightarrow$' refers to a happens-before ordering.

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 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. More recent versions of the UltraSPARC micro-architecture 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* [38, 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 an 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) [38, 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, 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. 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 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, since 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 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

Algorithm 6: STM-related variables

Global Variables

transactions	: `Tx[]`	// thread metadata
timestamp	: `Global Clock Object`	// see Algorithm 5
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

re-establish the strong ordering guarantees of a software global clock object when using `rdtscp`.

3.3 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 chapter are presented as Algorithm 6.

3.3.1 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 [25, 91]. 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.

3.3.2 Check-Once Ownership Records

We begin with an analysis of STM algorithms based on "check-once" orecs [91]. 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 7 and 8 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 before reading the location: such a check is effectively subsumed by Line 2. 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

Algorithm 7: Canonical STM algorithm with check-once ownership records

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

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

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

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

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 5 and 6.

Check-once orec algorithms typically use a shared memory global counter (as in Algorithm 5). 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 18 and 19 can seem to reorder.

Algorithm 8: Helper functions

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

6  ReleaseLocks()
7      foreach addr in locks do
8          orecs[addr].releaseTo(end)

9  WriteBack()
10     foreach ⟨addr, v⟩ in writes do
11         *addr ← v

12 Validate()
13     if end ≠ start + 1 then
14         foreach addr in reads do
15             v ← orecs[addr].getValue()
16             if v ≥ start and v ≠ my_lock then Abort ()

17 Abort()
18     foreach addr in locks do
19         orecs[addr].releaseToPrevious()
20     restartTransaction()
```

Such a reordering is incorrect, as it can lead to a thread observing inconsistent state. Suppose that transaction A has just completed Line 17 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 2. The correctness of check-once orecs relies on the following:

- If B reads the timestamp (Line 2) before A increments the timestamp at Tx-Commit (Line 19), 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 18) and B's read of L (Line 6), and thus cannot guarantee that B will observe v'.

- If B reads the timestamp (Line 2) after A increments the timestamp at TxCommit (Line 19), then either (a) the step that checks the orec in B's TxRead (Line 7) happens before A releases the lock (Line 20), 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 19 of thread A attains a timestamp before some memory update by thread A on Line 18 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 19 (**reordered**);
- B reads the timestamp at Line 2 in TxBegin;
- A executes Line 18 and Line 20 in TxCommit (**reordered**);
- B checks orec at Line 7 in TxRead.

In this case, B reads the location (Line 6) before A updates it (Line 18), but since A gets its timestamp (Line 19) before B starts (Line 2) and A releases its locks (Line 20) before B checks the orec (Line 7), B does not abort. B's continued execution with inconsistent state is not merely a violation of opacity [29], because it will not even be detected by validating B.

The latest IA32/x86_64 specification [38, 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 19 can appear to execute before Line 18, even when using `rdtscp` (with its implicit `LFENCE`). Furthermore, note that an `MFENCE` instruction is insufficient in this

case: since neither `rdtscp` nor `MFENCE` reads shared memory, the sequence `store, MFENCE, rdtscp` does not guarantee that the `rdtscp` occurs after the store.

Algorithm 9: Replacement TxCommit code when using `rdtscp` with check-once orecs

```
    TxCommit()
        ...
19      AtomicAdd(end, 0)
20      end ← rdtscp
21      ReleaseLocks(end)
```

Our solution, presented in Algorithm 9, 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 a `lock`-prefixed instruction, it enforces ordering in that its memory effects must happen *after* the stores that comprise the call on Line 18. Additionally, since it is a read-modify-write (RMW) instruction, this increment involves a read, and thus the subsequent `rdtscp` on Line 20 must order after it due to the implicit `LFENCE` of `rdtscp`. When coupled with the fact that there is a data dependence between Lines 20 and 21 (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 20 has the correct behavior.[1]

Let us now consider the use of `rdtscp` on Line 2. 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 8, 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 2 occurs after thread B Line 6, then a possible ordering is:

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

- A completes validation at Line 17;
- B dereferences the address at Line 6 in TxRead (**reordered**);
- A completes writeback and finishes TxCommit;
- B completes its `rdtscp` at Line 2 (**reordered**);
- B checks the orec at Line 7 in TxRead.

In this case, B will read v, but since thread A Line 21 (Algorithm 9) precedes thread B Line 2, thread B will not abort.

Algorithm 10: Replacement TxBegin code when using `rdtscp` with check-once orecs

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

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 10, 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 2 cannot reorder after any of the memory operations within a TxRead, TxWrite, or TxCommit operation.

3.3.3 Check-Twice Ownership Records

Listing 11 presents a canonical lazy STM with check-twice orecs. This style of STM algorithm is representative of TL2 [21], TinySTM [25], and most other orec-based algorithms. While ordering is required between Lines 6 and 7, and between Lines 7 and 8, 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 2 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 [66].

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 18 to be only one greater than the value of the clock that was observed at Line 2. 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 2), 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 10. This sequence ensures that the `rdtscp` in TxBegin does not delay past any shared

Algorithm 11: Canonical STM algorithm with check-twice ownership records

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

4  TxRead(addr)
5      if addr ∈ writes then return writes[addr]
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     else Abort()

13 TxWrite(addr, v)
14     writes ← writes ∪ {⟨addr, v⟩}

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

memory operations within the transaction body.

Algorithm 12: Replacement TxCommit code when using `rdtscp` with check-twice orecs

```
   TxCommit()
       ...
18     end ← rdtscp
19     Validate(end)
20     WriteBack()
21     ReleaseLocks(end)
```

The second point at which the counter is accessed is at commit time. There are data dependencies between the read of the counter (Line 18) and the validation (Line 19) and lock release (Line 21) 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 12, suffices. Our lone concern is the case where the `rdtscp` bypasses a preceding store operation. In this case, we must ensure that Line 18 executing before Line 17 will not compromise correctness. The key difference relative to check-once orecs is that with check-twice orecs, Lines 6–8 alone suffice to ensure that if thread A Line 18 precedes thread B Line 2, then on an access to L, B will abort unless thread B Line 6 follows thread A Line 21. 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 18 were reordered before thread A Line 17, then when thread A Line 18 precedes thread B Line 2, B might be able to execute lines 8–10 *before* thread A Line 17. 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 21, B will not detect an inconsistency. Fortunately, this problem is averted since thread A Line 17 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 18 cannot bypass thread A Line 17 in Algorithm 12.

3.3.4 Timestamp Extension

A common practice in STM algorithms is to "extend" a transaction's start time to avoid aborts in the read function (Algorithm 7 line 11 and Algorithm 11 line 12). The technique is simple [66, 91]: 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 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 13.

Algorithm 13: Timestamp extension with a global shared memory clock

```
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 a `lock`-prefixed instruction and a data dependence to provide this ordering. The resulting timestamp extension code appears in Algorithm 14. 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 14: Timestamp extension with `rdtscp`

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

3.4 Privatization Safety

In languages whose memory model demands static separation [1], such as Haskell, Scala, and Clojure, the algorithms from Section 3.3 can be used directly. However, the current draft specification for adding TM to C++ [3] requires implicit privatization

safety [31, 55, 83, 91] instead of static separation. We now turn our attention to mechanisms that can make our algorithms from Section 3.3 privatization safe.

3.4.1 The Privatization Problem

In general, privatization safety can be thought of as the need to prevent two problems [83], 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 [55]. The second approach is to use orthogonal solutions to the two problems. The approach, known as the Detlefs algorithm [51, 81], 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.

3.4.2 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

Algorithm 15: Privatization safe STM algorithm using check-twice orecs and `rdtscp`

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

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

22 TxWrite(addr, v)
23     writes ← writes ∪ {⟨addr, v⟩}

24 TxCommit()
25     if writes = ∅ then
26         start ← ∞
27         return
28     AcquireLocks ()
29     end ← rdtscp
30     Validate(end)
31     WriteBack()
32     start ← ∞
33     ReleaseLocks(end)
       // Quiescence
34     foreach tx in transactions do
35         while tx.start ≤ end do
36             wait
```

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 15 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.

Since this algorithm uses timestamp extension, we can employ a validation fence [83], rather than the more coarse-grained transaction fence, for privatization safety. To maximize the effectiveness of this technique, the timestamp extension on Lines 16 to 21 differs slightly from Algorithm 14, 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.

3.5 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 [25]. 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 [82], 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 does not have extensible timestamps [21]. 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 [18].

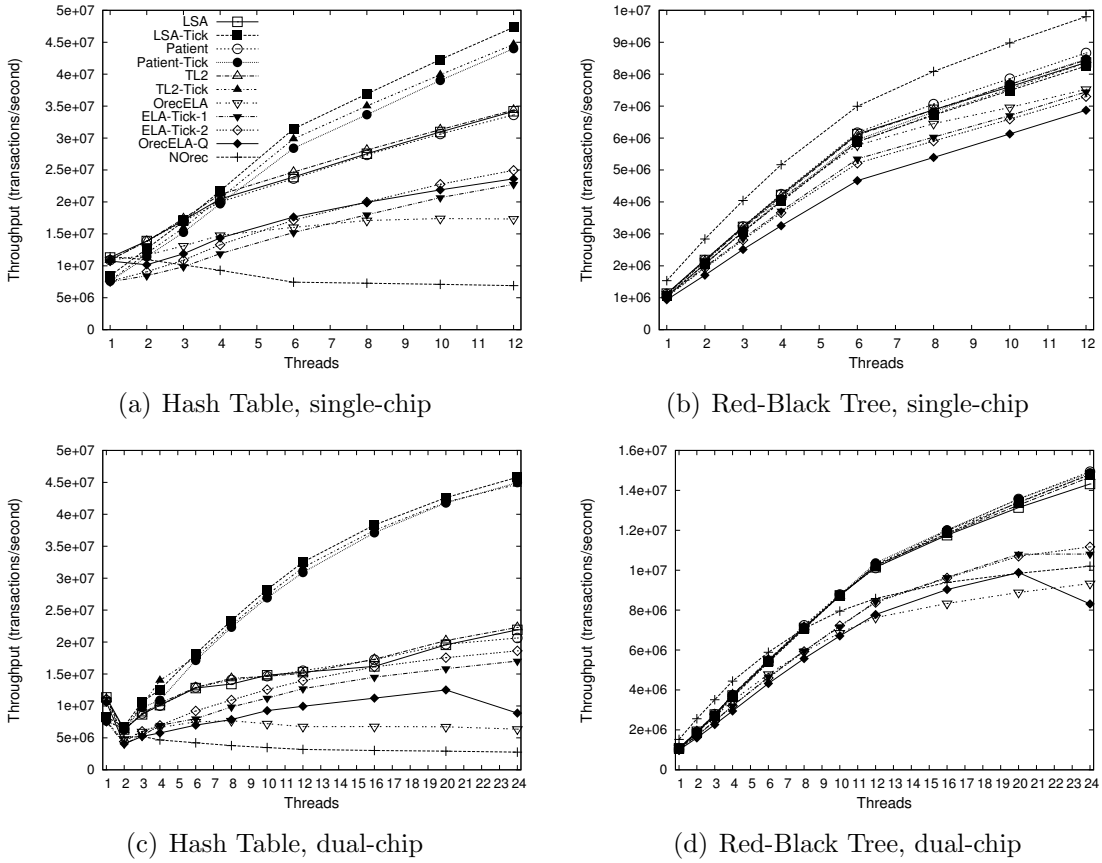
- OrecELA – A variant of the Detlefs algorithm [48, 81], which uses check-once orecs and extensible timestamps.
- OrecELA-Q – A variant of OrecELA that uses check-twice orecs and extensible timestamps, but which relies on quiescence instead of polling to ensure privatization-safety.
- ELA-Tick-1 – A version of Algorithm 15 using check-once orecs.
- ELA-Tick-2 – A variant of Algorithm 15 using check-twice orecs.

All algorithms were implemented within the RSTM framework [80], 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 [56]. 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.

3.5.1 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 3.1(a) and 3.1(c) present the performance of our STM algorithms for a microbenchmark in which all transactions repeatedly access the same hash table.



(a) Hash Table, single-chip

(b) Red-Black Tree, single-chip

(c) Hash Table, dual-chip

(d) Red-Black Tree, dual-chip

Figure 3.1: 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.

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 pre-filled 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, since 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 inter-chip 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, since 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 multi-chip system is particularly costly, resulting in OrecELA and NOrec both failing to scale. Note that OrecELA-Q scales better

than OrecELA, since it has lower quiescence overhead and avoids the costly writer serialization of OrecELA.

On the opposite end of the spectrum, Figures 3.1(b) and 3.1(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. Again, the data structure is pre-populated 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, 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 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 3.1, 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

Table 3.1: Quiescence overhead

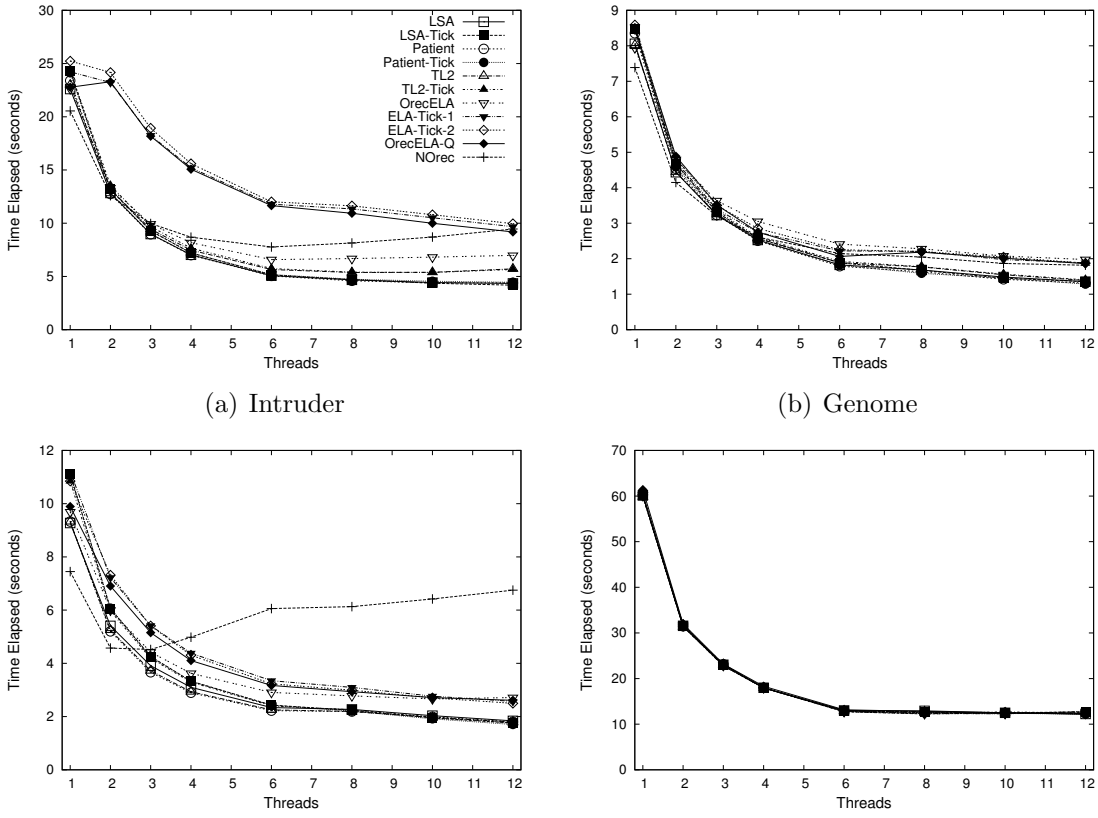
Single-chip system

	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

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 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.



(a) Intruder

(b) Genome

(c) SSCA2

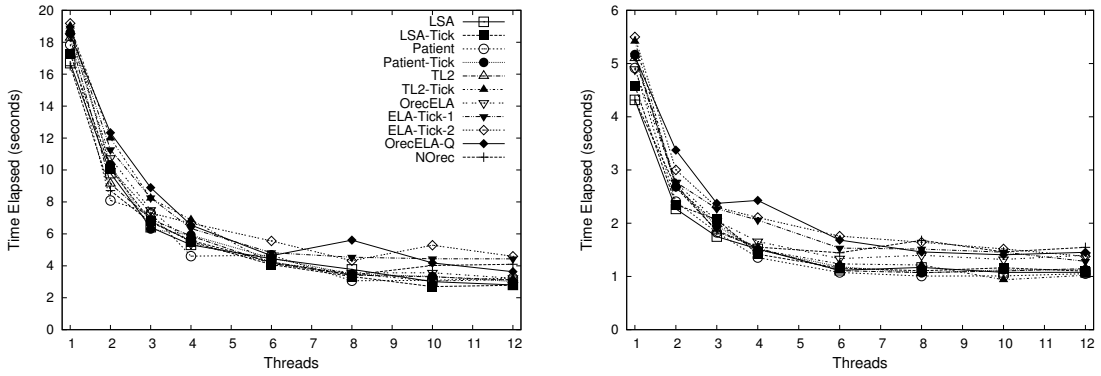
(d) Labyrinth

Figure 3.2: STAMP results on the single-chip system (1/2).

3.5.2 STAMP Performance

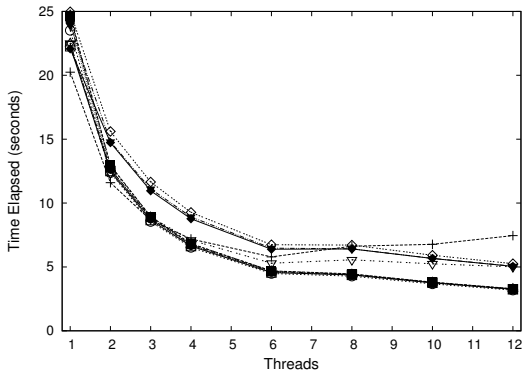
The variety of behaviors exhibited by the different STAMP benchmarks provide additional insight into the benefits and weaknesses of our `rdtscp`-based algorithms. Figures 3.2 and 3.3 present results for the single-chip system; Figures 3.4 and 3.5 present results for the dual-chip system.

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 trans-

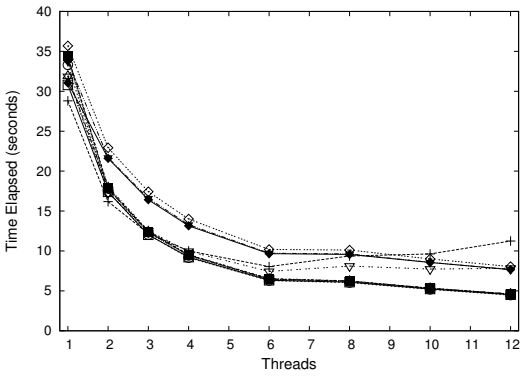


(a) KMeans (low contention)

(b) KMeans (high contention)



(c) Vacation (low contention)



(d) Vacation (high contention)

Figure 3.3: STAMP results on the single-chip system (2/2).

actions. 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, since this mechanism causes significant coherence traffic.

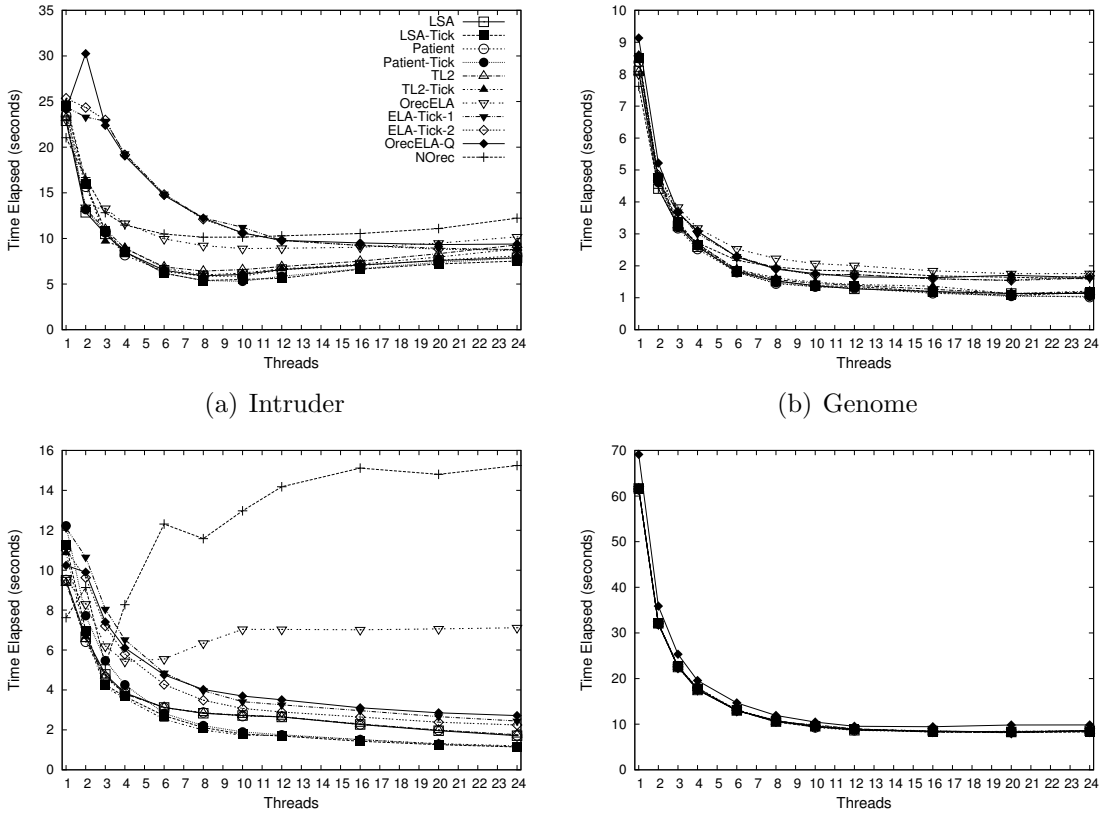
SSCA2 In many regards, SSCA2's behavior is modeled by our Hashtable microbenchmark: all transactions perform writes, and transactions are frequent and small. NOrec is known to perform poorly, due to serialization at commit time, and on the dual-chip system, OrecELA performs poorly as well. The only noteworthy 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; "carefully tuned" 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 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 [6] upon which it is based, Labyrinth conflates memory speculation and control flow speculation: In Labyrinth transactions, a quadratic number of un-instrumented (non-transactional) 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 un-instrumented 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 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.



(a) Intruder

(b) Genome

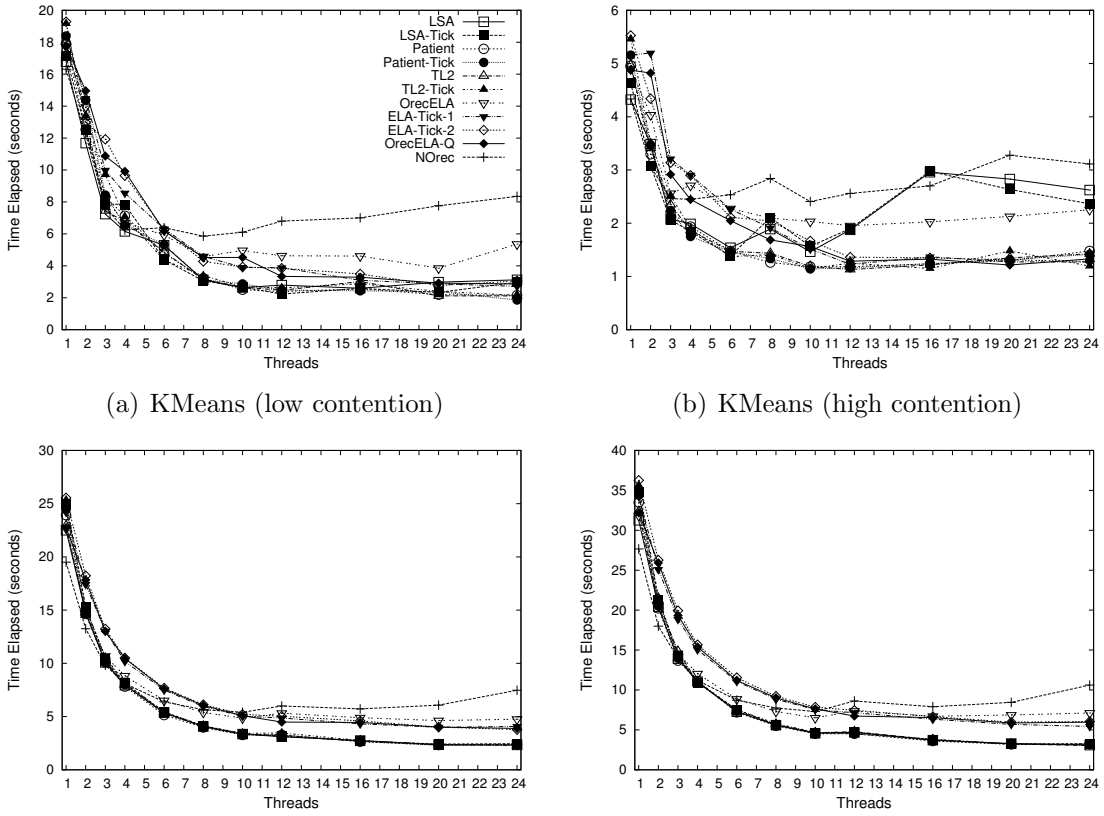
(c) SSCA2

(d) Labyrinth

Figure 3.4: STAMP results on the dual-chip system (1/2).

3.6 Summary

In this chapter, 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



(a) KMeans (low contention)

(b) KMeans (high contention)

(c) Vacation (low contention)

(d) Vacation (high contention)

Figure 3.5: STAMP results on the dual-chip system (2/2).

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 [76] 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.

Chapter 4

Reducing the Abort Rate by Delaying Read-Modify-Writes

In this chapter, we present a technique to reduce the transaction abort rate. The original work was published in "Transactional Read-Modify-Write Without Aborts" at ACM Transactions on Architecture and Code Optimization, Volume 11 Issue 4, January 2015 Article NO. 63.

4.1 Introduction

As described in Section 1.1 of Chapter 1, language-level transactions are often described as providing atomicity, in the sense that the operations that comprise the transaction appear to happen "all at once", as an indivisible operation without any intervening memory operations from concurrent transactions. While Hardware Transactional Memory is largely able to provide this illusion for small transactions, Software Transactional Memory typically cannot, due to the high overheads that arise [55, 28]. Instead, STM implementations fall back to semantics based on multiple logical locks, in which transactions appear to acquire locks covering their read and/or write sets dynamically during the course of their execution.

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

Figure 4.1: A transaction that modifies a highly contended variable. If line 3 could be delayed until commit time, concurrent invocations of this transaction would not abort due to accesses to `stats`.

The unfortunate consequence is that compilers must be conservative when reordering memory accesses that occur within a transaction. In particular, under the most popular "Encounter-time Lock Atomicity" (ELA) semantics proposed by Menon et al. [55], a transaction appears to acquire a lock for each location it reads at the time each location is first read. In practical implementations of STM [21, 25], writes are treated as implicit reads, and also appear to acquire locks at the time the location is first accessed, even though ELA permits writes to appear to delay lock acquisition until as late as the commit point of the transaction.

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

To illustrate this point, consider the code in Figure 4.1. In every invocation, considerable work is done by the first function, after which a statistics counter is incremented. The increment consists of a shared memory read, local computation, and a shared memory write. Subsequently, another expensive computation is performed.

During the second computation, the transaction is vulnerable to aborts on account of concurrent accesses to the counter.

Suppose two transactions T_1 and T_2 execute this code in lock-step, but with different values of x and y. It is possible for their only conflict to be on accesses to `stats`. If the underlying TM is eager (e.g., best-effort HTM [14, 38, 16] or STM that acquires locks upon first write access [25, 22, 91]), one transaction will lock `stats`, the other will attempt to read `stats`, and the resulting conflict will cause at least one transaction to be aborted. If the underlying TM is lazy [57, 21, 18], then both transactions will continue to their commit point, at which time the conflicting accesses to stats will be detected and at least one transaction will abort. In both cases, a significant amount of computation will be devoted to a transaction that does not complete, resulting in wasted work.

If it were known that `stats` was never accessed by either function, the increment could be moved to the end of the transaction, avoiding conflicts in many cases [52]. More aggressively, since the result of the increment is also never used within the transaction, but the increment still must be performed atomically with the rest of the transaction, it is even be possible to delay the increment until some point *within the commit operation*. Doing so would eliminate all possibility of aborts due to `stats` accesses.

The aborts that result from shared counters arise in real code, as demonstrated by efforts to transactionalize memcached [64, 70]. Worse yet, in memcached the relevant counters are, themselves, incremented in nested function calls that occur only on some branches. When the prospect of nested transactions is also considered, we must conclude that manually refactoring code to place hot increment operations at the end of transactions is not possible.

In this chapter, we propose an algorithm for dynamically reordering increment and other simple read-modify-write operations within a transaction. Our mechanism

employs run-time tracking to ensure correctness even when the target of a deferred operation is read or written by other instructions within the transaction. We present the algorithm and tracking mechanisms in Section 4.2, and then in Section 4.3 present candidate implementations, which use annotations or live-out analysis to instrument operations so they can be delayed. Section 4.4 discusses the surprising consequences that our algorithm can have on the publication idiom [55] and proposes solutions. We evaluate our algorithms in Section 4.5, discuss related work in Section 4.6, and then summarize in Section 4.7.

4.2 An Algorithm for Delaying RMWs

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

4.2.1 STM Background

STM implementations typically expose an API with four operations: `TxBegin` creates a checkpoint and initializes per-thread metadata to support tracking conflicts on reads and ensuring serializability of writes; `TxRead <T>` reads a location of type `T` and ensures the read is consistent with all prior reads and writes performed by the transaction (this property, called opacity [29], provides a basis for asserting the correctness of STM implementations); `TxWrite <T>` updates the value of a location such that subsequent reads within the same transaction will see the update, but other transactions will not (yet) see the update. `TxCommit` makes writes visible to other transactions only if doing so will produce a result indistinguishable from an execution history in which one transaction runs at a time. Typically, `TxCommit` ensures that

all to-be-written locations are locked by the transaction, verifies that no concurrent transaction updated any location read by the transaction, then finalizes writes and releases locks.

For simplicity of presentation, we assume throughout this section that transactions operate only on locations of a single type. We also assume that the language is type safe, such that for any two operations on locations L_1 and L_2, both of size S, $L_1 = L_2$, or $|L_1 - L_2| \geq S$. Note that these assumptions do not apply to the C and C++ languages, requiring a more complicated implementation in Section 4.5.

The basic framework for an STM algorithm is presented as Table 4.1 and Algorithms 16–17. If we assume that `TxRMW` is never called, then the `rmws` set will be empty, and the resulting algorithm resembles TL2 [21] or the commit-time-locking variant of TinySTM [25]. Note that this algorithm delays all writes until commit time, buffering them in a per-thread log during transaction execution. The correctness of the algorithm hinges on three properties. First, on Algorithm 16 line 6, every read first checks if there was a previous write to the same location by the current transaction. This check ensures processor consistency: an in-flight transaction observes every modification that it intends to make to main memory. Second, the `TxCommit` function ensures that transactions appear to commit atomically. This property is provided via the use of versioned locks (ownership records, or orecs) and a global timestamp. Briefly, locks can only be acquired if their version number is no greater than the time at which the transaction was last known to be valid, and all reads are verified after all locks are acquired, to ensure serializability. Third, transactions discard speculative state and restart whenever they observe inconsistencies. The use of incremented values of timestamps as the version numbers is the key to providing this property efficiently.

Table 4.1: STM-Related Variables

Global Variables

timestamp	`Integer`	timestamp
orecs	`OwnershipRecord[]`	orec table, the MSB of each orec is set to 1 if orec is acquired

Per-transaction Variables

my_lock	⟨`Integer`, `Integer`⟩	⟨1, *thread_id*⟩
start	`Integer`	start time
end	`Integer`	end time
writes	`WriteSet`	pending writes
reads	`ReadSet`	locations read
locks	`LockSet`	locks held
rmws	`RMWSet`	pending RMWs
rdflag	`Boolean`	see Algorithm 18
rdaddrs	`AddressSet`	see Algorithm 19

4.2.2 Problem: Aborts on Read-Modify-Write

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

A Read-Modify-Write (RMW) operation takes a function f and a location X as parameters. It applies function f to the value x stored at location X, and writes back the new value $f(x)$ to X. We assume the function f is pure. Using the existing TM

Algorithm 16: A lazy STM algorithm with support for delayed RMWs. Underlined code represents additions relative to a traditional lazy STM algorithm.

```
1  TxBegin()
2      start ← timestamp
3      reads, writes, locks, rmws ← ∅    // clear read-modify-write set
4  TxRead(addr)
       // if there's a previous RMW on addr, promote it
5      if addr ∈ rmws then return Promote(addr)
6      if addr ∈ writes then return writes[addr]
7      while true do
8          o1 ← orecs[addr]
9          v ← *addr
10         o2 ← orecs[addr]
11         if o1 = o2 ∧ o2 ≤ start then
12             reads ← reads ∪ {⟨addr, o1⟩}
13             return v
14         else if ¬o2.Locked() then Extend()
15         else Abort()
16 TxWrite(addr, v)
17     if addr ∈ rmws then
18         rmws ← rmws \ {⟨addr, −⟩}  // forget this RMW, since it will be overwritten
19         TxRead(addr)               // but perform a read to preserve consistency
20     writes ← writes ∪ {⟨addr, v⟩}
21 TxRMW(addr, f)
22     rmws ← rmws ∪ {⟨ addr, f ⟩}    // on an RMW, log the address and operation
23 TxCommit()
       // immediate return for read-only transactions
24     if writes = ∅ ∧ rmws = ∅ then return
25     AcquireLocks ()
26     end ← AtomicInc(&timestamp, 1)
27     if start ≠ end − 1 then ValidateBoth() // validate both read set and rmws set
28     WriteBack()
29     Replay()            // replay RMWs
30     ReleaseLocks(end)
```

interface, an RMW operation in a transaction is instrumented as follows:

```
RMW(X, f) :

  x <- TxRead(X);

  TxWrite(X, f(x));
```

For a given transaction T, executing an RMW operation on location X makes the transaction vulnerable to conflicts on X. In most implementations, the transaction

Algorithm 17: Helper Functions.

```
1  AcquireLocks()
2      foreach addr in writes do
3          o ← orecs[addr]
4          if o.Locked() ∨ o > start then
5              Abort()
6          if ¬CAS(&orecs[addr], o, my_lock) then
7              Abort()
8          locks ← locks ∪ {addr}
9      foreach m in rmws do
10         m.orec ← orecs[m.addr]
11         o ← m.orec
12         if o = my_lock then continue
13         if o.Locked() then Abort()
14         if ¬CAS(&orecs[m.addr], o, my_lock)
           then
15             Abort()
16         locks ← locks ∪ {addr}
17 WriteBack()
18     foreach ⟨addr, v⟩ in writes do
19         *addr ← v
20 ValidateBoth()
21     foreach ⟨addr, v⟩ in reads do
22         foreach m in rmws do
23             if m.addr = addr ∧ m.orec ≠ v
               then
24                 Abort()
25         o ← orecs[addr]
26         if o ≠ v ∧ o ≠ my_lock then Abort()
27 ValidateRead()
28     foreach ⟨addr, v⟩ in reads do
29         o ← orecs[addr]
30         if o ≠ v ∧ o ≠ my_lock then Abort()
31 Promote(addr)
32     if addr ∈ writes then
33         v ← writes[addr]
34     else
35         while true do
36             o1 ← orecs[addr]
37             v ← *addr
38             o2 ← orecs[addr]
39             if o1 = o2 ∧ o2 ≤ start then
40                 reads ← reads ∪ {⟨addr, o1⟩}
41                 break
42             else if ¬o2.Locked() then
43                 Extend()
44             else Abort()
45     foreach m in rmws do
46         if addr = m.addr then
47             v ← m.f(v)
48     rmws ← rmws \ {⟨addr, −⟩}
49     writes ← writes ∪ {⟨addr, v⟩}
50     return v
51 ReleaseLocks(end)
52     foreach addr in locks do
53         orecs[addr].releaseTo(end)
54 Replay()
55     foreach m in rmws do
56         *m.addr ← m.f(*m.addr)
57 Extend()
58     t ← timestamp; ValidateRead()
59     start ← t
60 Abort()
61     foreach addr in locks do
62         orecs[addr].releaseToPrevious()
63     rollback
```

aborts if either (1) X is already acquired when T performs the `TxRead`, or (2) any other transaction writes to X and commits before T.

Let us now return to our example in Figure 4.1. Suppose that in an execution trace, the function calls by concurrent transactions T_1 and T_2 both observe only unlocked orecs with versions equal to 0, and that these sets of orecs observed by the transactions have an empty intersection. We will also assume that the orec related to

`stats` has an initial value of 0. Suppose T_1 commits after T_2 has read `stats`: it will lock `stats`' orec, update the global timestamp to N, and then updates `stats`' orec version to N, where N is larger than T_2's start time. As a result, T_2 must abort. The abort will occur either when T_2 tries to lock `stats`' orec within `TxCommit`, and sees that N is greater than the time at which it was last valid (either T_2's start time or the time of its last validation), or else on account of a validation within the `Extend` function, which would also observe a "too new" orec value of N.

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

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

4.2.3 The Basic Algorithm

Our basic algorithm to support transactional RMW operations is presented in Algorithm 16. When instrumenting source code with transactional constructs, candidate RMW operations are replaced with calls to `TxRMW`, which allocates an entry for the RMW operation and appends it to the RMW log (`rmws`). The `TxRead`, `TxWrite`, and

`TxCommit` functions are extended to interact with the log. The design is guided by the assumption that the number of RMW operations tends to be small, in comparison to read and write operations. Note that it would be possible to promote a `TxRMW` to a `TxWrite` if it is known that the location was previously read or written, but at the cost of searching the read and write sets.

On a `TxRead` to location L, a lookup is performed in `rmws` (line 5). If delayed RMW operations on L are found, they are immediately transformed into their corresponding `TxRead` and `TxWrite` operations, via the `Promote` function. This ensures that the return value of the `TxRead` is consistent with an execution in which all delayed RMWs occurred before the `TxRead`.

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

Lastly, to ensure that an RMW performed on L observes prior writes to L by the transaction, we order `Replay` after `WriteBack` in the `TxCommit` function.

4.2.4 Correctness

For the time being, we focus on correctness in the absence of the publication idiom [55]. The algorithms above, and discussion of correctness below, are applicable to languages that enforce static separation [1], as well as programs that allow for privatization of shared data. Section 4.4 discusses algorithmic extensions needed for publication safety.

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

- Opacity: if transaction T_1 performs a read of X and then a delayed operation on X, then T_1 must abort if a concurrent transaction modifies X before T_1 commits.

- Atomic commit: a delayed operation must not appear to happen either *before* or *after* the remainder of the transaction, but *atomically* with the rest of the transaction.

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

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

Second, we must ensure that the RMWs happen only at a time when all reads performed by the transaction are known to be valid. In the absence of delayed RMWs, this is achieved in `TxCommit` by acquiring all locks, then validating. If the validation succeeds, then all reads were valid at a time when all locks were held. However,

there is a subtlety here: in lines 2–8 of `AcquireLocks`, locations are locked only if their version number is no greater than the transaction start time. This allows a simplification in `ValidateRead` (line 30): if the location being validated is owned by the validating transaction, it is known that the read was valid. Unfortunately, we cannot place the same constraint on locks acquired for RMW operations, or else much of the benefit of delayed RMWs would be lost. Instead we add lines 9–16 to `AcquireLocks`, so that RMW locations can be locked as long as the lock is currently unheld, even if the version number is "too new". We must then guard against the situation where an RMW was performed to location L after L was read, but there was an intervening modification to L by another transaction. This ordering is detected by storing the version at the time an RMW location was locked (`AcquireLocks` line 10). Then, during read set validation, we check if a location being validated was locked for RMW, via `ValidateBoth` lines 22–24. If so, we verify that the version number when lock was acquired was no greater than the version of the lock at the time of the read.

4.3 Implementation

The naïve implementation of `TxRMW` in Algorithm 16 suffers from high asymptotic complexity: since the `rmws` log stores an ordered list of delayed RMWs, if there are M RMW operations and R reads, there can be $O(M)$ overhead on each of R calls to line 5 of `TxRead`, and another $O(R \times M)$ overhead due to lines 22–24 of `ValidateBoth`. Furthermore, if there are W calls to `TxWrite`, there can be up to $O(W \times M)$ overhead to eliminate RMWs to locations that are overwritten (lines 17–19). We now present mechanisms for decreasing these overheads. We also discuss alternative mechanisms for guiding the compiler to produce calls to `TxRMW`.

4.3.1 `TxRMW` via Live-Out Analysis

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

As a compiler-based approach is likely to produce a large number of delayed operations, and hence a large value of M, it is beneficial to reduce the asymptotic cost of checking when a read and RMW are performed to the same location. To reduce the cost of line 5, it is possible to maintain a hash table storing all addresses involved in RMWs by the current transaction, so that each execution of line 5 has constant overhead. Alternatively, a technique proposed for TL2 [21] can be used, where a small Bloom filter [9] approximates the locations in `rmws`, and can be consulted before performing a lookup.

To reduce overhead at commit time, we recommend a technique pioneered by the Amino CBB STM algorithm [37]. In this technique, each orec has a second field that stores the previous version. When a transaction acquires an orec, it stores the old version number in the second field. At commit time, Amino acquires any unlocked orec, even if the orec version number is "too new". Then, during validation, any read whose orec is locked by the current transaction checks the second field to determine

if the lock was acquired at a time when the read was consistent. While Amino CBB STM used this technique to minimize aborts for transactions that wrote to locations they did not read, we observe that it also eliminates $R \times M$ overhead at commit time for our algorithm.

4.3.2 `TxRMW` via Programmer Annotation

We expect the overhead of maintaining a hash table to be high, and the imprecision of a Bloom filter to also be high. We are also concerned that a compiler might prove too aggressive in its application of delayed `TxRMW` calls. Additionally, a programmer may have better insight into the frequency with which certain locations are accessed after an RMW, particularly with profiler feedback, and thus might wish to use `TxRMW` for some updates to variables that are subsequently accessed only on an uncommon code path.

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

Algorithm 18 refines Algorithm 16 to distinguish between accesses to annotated variables (which call `TxRmwRead`, `TxRmwWrite`, and `TxRMW`) and accesses to non-annotated variables (which call `TxRead` and `TxWrite`). These changes require a `rdaddrs` log to store the addresses of annotated variables that are read.

The annotations ensure that only `TxRmwRead` promotes delayed RMW operations (line 20-21), and only `TxRmwWrite` needs to discard pending RMWs (lines 26-28). These changes reduce two sources of overhead by eliminating lookups for common-case reads and writes. The third source of overhead in the naïve algorithm relates to RMW operations that follow reads to the same location, and manifests as extra overhead during commit-time validation. To avoid this cost, we leverage the expectation that

Algorithm 18: An algorithm for delayed RMWs that assumes the variables involved in delayed RMWs are annotated.

```
1  TxBegin()
2      start ← timestamp
3      reads, writes, locks ← ∅
4      rmws, rdaddrs ← ∅
5  TxRead(addr)
6      if addr ∈ writes then
7          return writes[addr]
8      while true do
9          o1 ← orecs[addr]
10         v ← *addr
11         o2 ← orecs[addr]
12         if o1 = o2 ∧ o2 ≤ start then
13             reads ← reads ∪ {⟨addr, o1⟩}
14             return v
15         else if ¬o2.Locked() then
16             Extend()
17         else Abort()
18 TxRmwRead(addr)
19     rdaddrs ← rdaddrs ∪ {addr}
20     if addr ∈ rmws then
21         return Promote(addr)
22     return TxRead(addr)
23 TxWrite(addr, v)
24     writes ← writes ∪ {⟨addr, v⟩}
25 TxRmwWrite(addr, v)
26     if addr ∈ rmws then
27         rmws ← rmws \ {⟨addr, −⟩}
28         TxRead(addr)
29     TxWrite(addr, v)
30 TxRMW(addr, f)
31     if addr ∈ rdaddrs then
32         v ← TxRead(addr)
33         v ← f(v)
34         TxWrite(addr, v)
35     else rmws ← rmws ∪ {⟨ addr, f ⟩}
36 TxCommit()
37     if writes = rmws = ∅ then return
38     AcquireLocks ()
39     end ← AtomicInc(&timestamp, 1)
40     if start ≠ end − 1 then
41         ValidateRead()
42     WriteBack()
43     Replay()
44     ReleaseLocks(end)
```

`rdaddrs` is small, and perform a lookup in `rdaddrs` during every `TxRMW`. When there is a match, we do not delay the RMW, but instead perform it immediately. As before, RMWs that follow writes do not require a special case, since `Replay` follows `WriteBack`.

We assume that annotated RMW operations are infrequent, since (a) a high number of contention hotspots would suggest that the program cannot scale, and (b) unlike live-out analysis, annotations do not silently convert many operations into delayed operations. This being the case, overheads should be significantly lower using annotations. With W writes and R reads of non-annotated variables, and W' writes, R' reads, and M' RMWs of annotated variables, we can expect the overhead of supporting delayed RMWs to drop to $O(R' \times M')$ in `TxRmwRead`, and $O(W' \times M')$ in `TxRmwWrite`. `TxCommit` no longer has any overhead due to delayed RMWs, but

`TxRMW` incurs $O(R' \times M')$ overhead. Note that this last quantity could be avoided with Amino-style orecs, but not doing so avoids adding additional instructions to commit-time validation. Since locks are held during this validation, and since both R' and M' should be small, we expect $O(R' \times M')$ overhead to be insignificant in practice.

4.3.3 Optimized Programmer Annotations

A final optimization appears in Algorithm 19. In this algorithm, we again expect the programmer to annotate the declaration of highly contended variables for which delaying RMWs may be profitable. However, we now assume that highly contended variables are rarely read or written in the same transactions as those that access annotated variables with RMW operations.

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

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

4.4 Impact on Semantics

Menon et al. [55] proposed several levels of transactional semantics, which require varying amounts of serialization at the boundaries of transactions, and which place

Algorithm 19: Aggressive optimizations for the common case: if any annotated location is read or written, delayed RMWs are disabled for the transaction.

```
1  TxBegin()
2      start ← timestamp
3      reads, writes, locks, rmws ← ∅
4      rdflag ← false
5  TxRead(addr)
6      if addr ∈ writes then
7          return writes[addr]
8      while true do
9          o1 ← orecs[addr]
10         v ← *addr
11         o2 ← orecs[addr]
12         if o1 = o2 ∧ o2 ≤ start then
13             reads ← reads ∪ {⟨addr, o1⟩}
14             return v
15         else if ¬o2.Locked() then
16             Extend()
17         else Abort()
18 TxRmwRead(addr)
19     if ¬ rdflag then
20         rdflag ← true
21         for each m in rmws do
22             Promote (m.addr)
23     return TxRead(addr)
24 TxWrite(addr, v)
25     writes ← writes ∪ {⟨addr, v⟩}
26 TxRmwWrite(addr, v)
27     if ¬ rdflag then
28         rdflag ← true
29         for each m in rmws do
30             Promote (m.addr)
31     TxWrite(addr, v)
32 TxRMW(addr, f)
33     if rdflag then
34         v ← TxRead(addr)
35         v ← f(v)
36         TxWrite(addr, v)
37     else rmws ← rmws ∪ {⟨ addr, f ⟩}
38 TxCommit()
39     if writes = rmws = ∅ then return
40     AcquireLocks ()
41     end ← AtomicInc(&timestamp, 1)
42     if start ≠ end − 1 then
43         ValidateRead()
44     WriteBack()
45     if ¬ rdflag then Replay()
46     ReleaseLocks(end)
```

varying restrictions on how programmers can transition data between a state in which they are accessed via transactions, and a state in which they are accessed nontransactionally. At the most basic level, these semantics propose different levels of support for the publication idiom: when a thread initializes private data and then uses a transaction to mark that data as visible to other threads, the underlying STM must ensure that all threads agree that the initialization happens before the transaction; otherwise, a thread might see that the data is marked as safe to access, but then observe the uninitialized data.

The different levels of semantics differ in terms of which racy publication idioms are allowed in Java programs. However, the two least restrictive levels, "Asymmetric Lock Atomicity" (ALA) and "Encounter-time Lock Atomicity" (ELA), are both applicable

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

Thread 1:		Thread 2:	
1		1	`transaction {`
2		2	`  tmp = data;`
3	`data = 1;`	3	
4	`transaction {`	4	
5	`  ready = true;`	5	
6	`}`	6	
		7	`  if (ready)`
		8	`    val = tmp;`
		9	`}`

Can val == 42?

Figure 4.2: Basic publication example (reproduced from Figure 1 of Menon et al. [2008]). The vertical ordering of instructions is meant to imply the execution order on a sequentially consistent machine.

to C++, where racy code is erroneous. In the context of C++, ALA and ELA differ in whether the compiler can reorder reads of a datum that might be concurrently initialized outside of a transaction.[1]

The canonical example appears in Figure 4.2, where ALA ensures that the race accessing `data` is benign, and does not produce the erroneous output `val == 42`. Note that when all transactions are protected by a single global lock, the race is also benign, as 42 can never be used by Thread 2. Under ELA semantics, neither the programmer nor the compiler is permitted to transform the code `if (ready) val = data;` into the code run by Thread 2.

Now consider an extension in which the "ready" flag is a counter, where zero indicates that `data` is not initialized, and any other value is the number of transactions that have used `data` in a successful transaction. This code appears in Figure 4.3. While admittedly contrived, we hope the reader agrees that this code is not unrealistic. For example, a naïve transactionalization of legacy code that includes `auto_ptr` could

[1]Note that since Menon's work focused on the Java language, there were additional constraints on the TM implementation, which cannot be entirely separated from publication-related issues. For example, in a racy program, Java forbids out-of-thin-air reads [47], and thus even at the weakest semantics levels, Java appears to require STM algorithms to use redo logs. In C++, an STM may use undo logs, since the behavior of racy programs is undefined. Throughout this section, we ignore Java-specific issues related to undo logs. The algorithms presented herein, if applied to an STM algorithm with redo logs, would be valid for Java.

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

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

Can val == 42?

Figure 4.3: Publication violation example with delayed RMWs.

result in this code.

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

If the RMW on line 2 is not delayed, the example in Figure 4.3 is correct for both ELA and ALA, and `val` will never equal 42. With delayed RMWs, the example breaks under ELA semantics: the read of `ready` by Thread 2 will result in a call to `TxPromote`, and effectively cause the read and write of `ready` to occur *after* Thread 2's transaction commits, instead of before `data` is read. Thus an STM implementation that only provides ELA semantics cannot naïvely delay RMWs without risking publication violations. Note that this finding extends the work of Menon et al., which showed that under ELA, dependent reads cannot be reordered above the reads that establish a control-flow dependency. Here, dynamic reordering of accesses that are not dependent are similarly unsafe, for both C++ and Java. With ALA or stronger semantics, delaying the RMW remains safe.

For STM algorithms that use ownership records and provide ELA publication

safety, such as those presented in Section 4.2, we can resolve this problem so that delayed RMWs are safe. We extend the `rmws` data structure to store an additional field. Then, in `TxRMW`, we begin by reading the orec that corresponds to the address parameter, and storing it along with the description of the delayed RMW, in this extra field.

If the delayed RMW is not performed until `Replay`, in the `TxCommit` function, then the added field is ignored. However, if the RMW is performed earlier, on account of a call to `Promote`, then we use the field. In `Promote`, we replace line 36 with $o_1 \leftarrow z$, where z is the value stored in the field, and we remove lines 42–43.

These changes restore publication safety by guaranteeing that a delayed RMW can only be promoted if the promotion is indistinguishable from a `TxRead` and `TxWrite` succeeding at the time the RMW was initially requested. By requiring `Promote`'s read (lines 35–44) to fail unless the orec is unlocked and storing the same value as stored in `rmws`, the read no longer appears to delay until the time `Promote` was requested. Note that the write performed by `Promote` can still be delayed until commit time.

4.5 Evaluation

In this section, we evaluate the performance of our mechanism for delaying RMW operations within transactions. We consider microbenchmarks, the STAMP benchmark suite [56], and a transactional version of the memcached application [70]. STM experiments were performed on a dual-chip 2.67GHz Intel Xeon 5650 system with 12 GB of RAM and 12 cores / 24 threads. HTM experiments used a single-chip 3.40GHz Intel Core i7-4770 with 4 cores / 8 threads. Both machines run Ubuntu Linux 13.04, kernel 3.8.0-21, and a pre-release 4.9 GCC compiler with –O3 and –m64 flags. Results are the average of 5 trials, which led to uniformly low variance.

4.5.1 Systems Evaluated

The default GCC STM implementation uses an eager STM, in which locks are acquired on first write access by a transaction, and updates are performed immediately, with an undo log for recovering from aborts. A critical feature of this implementation is that it is correct for arbitrary C code: well-known pitfalls [97] related to unaligned accesses, unsafe casting, and overlapping accesses to variables of different sizes are all handled correctly. We added a new commit-time locking algorithm to GCC, in order to assess the effect of our delayed RMWs on both eager and lazy STM. Our implementation stores buffered writes in a tree, incurring $O(log(n))$ overhead on each write-set lookup. To the best of our knowledge, our implementation is the only correct lazy STM for GCC; it also satisfies all of Menon's requirements for a Java STM implementation. Throughout this section, experiments using STM algorithms derived from the default GCC algorithm are labeled "Eager", and those derived from our commit-time locking algorithm are labeled "Lazy". Both algorithms provide ELA semantics by default.

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

For the Eager and Lazy STM systems, we compare performance against three implementations of our delayed RMW mechanism. Experiments labeled "naïve" use Algorithm 16 and do not take advantage of annotations to avoid lookups in the `rmws` log on every read and write. Experiments labeled "Annotated" correspond to the use of Algorithm 18, where we can statically distinguish between reads and writes to locations that might have delayed RMWs, and those that do not. Finally,

experiments labeled "Flag" indicate the use of Algorithm 19, which optimizes for the case where delayed RMWs tend to be to locations that are not otherwise accessed by the transaction. Unless otherwise noted, experiments do not include the extensions from Section 4.4.

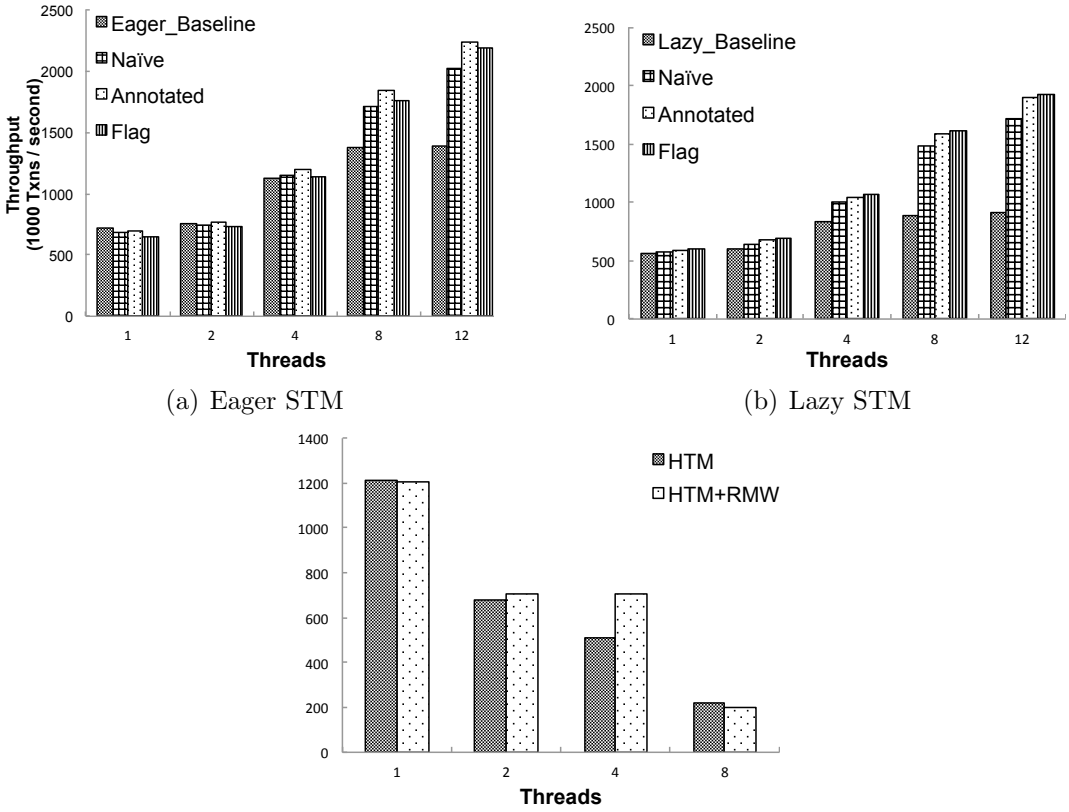
We also added support for RMW operations to the GCC HTM runtime. For these tests, we use a variant of the "Annotated" algorithm: Regular reads and writes to shared memory from within a transaction do not incur function call overhead, but RMW operations, and reads and writes to annotated variables, are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

4.5.2 Microbenchmark Performance

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

We consider two microbenchmarks based on the red-black tree implementation from the RSTM library [50]. This tree implementation generally offers good scalability with no internal bottlenecks, but provides only insert, remove, and lookup operations; there are no methods for iteration or statistics (e.g., tree size). We configured the tree experiments to perform an equal mix of insert, lookup, and remove operations, using 20-bit keys, and pre-filled the tree with 2^{19} entries. Charts present the average of five 5-second trials.

Our first variant of the tree adds a vector of counters. On every lookup (whether



(a) Eager STM

(b) Lazy STM

(c) HTM

Figure 4.4: Red-Black Tree experiments augmented with a global vector of counters to monitor the height at which searches terminate.

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

Figure 4.4 presents the throughput for the vector-of-counters tree. The Eager baseline scales poorly; the Lazy baseline is even worse, due to wasted work performed by transactions that conflict on a counter, but do not detect the conflict until commit time. When our mechanism is used to defer RMWs on the counters, throughput

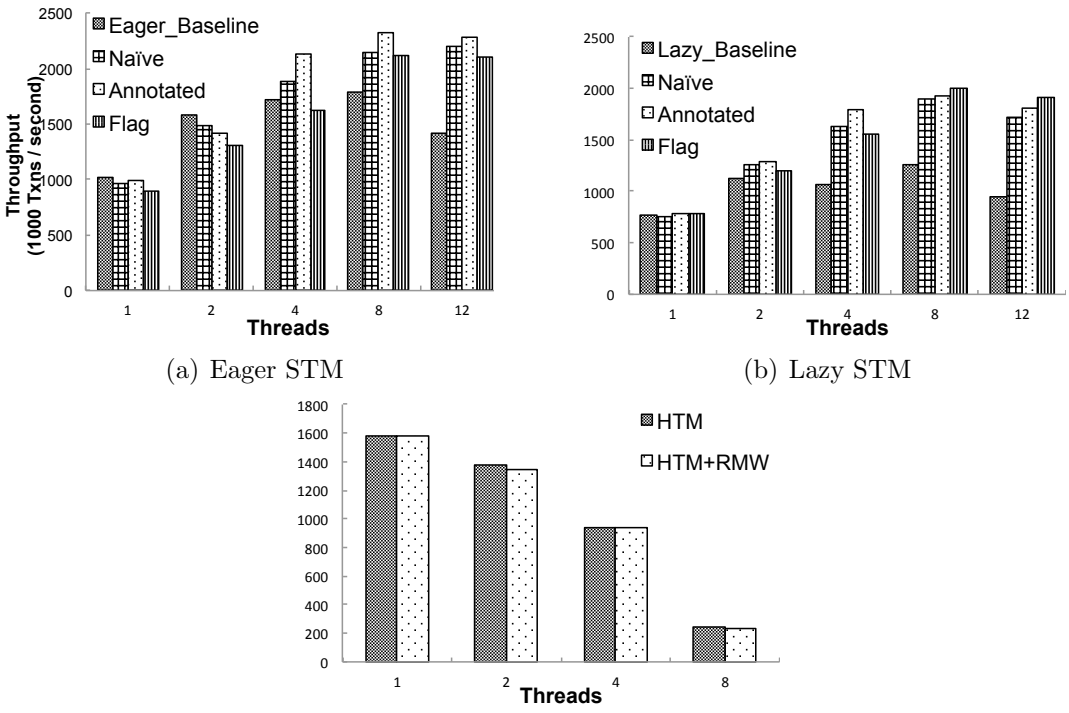
increases dramatically, and the performance difference between eager and lazy vanishes. As expected, the variations on our mechanism that lower asymptotic overhead lead to the best performance, but even the naïve implementation offers significant improvement.

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

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

Figure 4.5 presents the throughput of the tree with a global count field. Since lookup transactions do not access the counter, their presence enables some scalability for the baseline algorithm. However, the counter becomes a bottleneck beyond 2 threads, dampening scalability significantly. While some amount of dampened scalability is unavoidable, since the cache lines holding the counter and its associated ownership record must move between cores, the difference between our algorithms and the baseline shows that some of the dampened scalability is due to aborts, and that those aborts can be prevented by delaying RMWs to hot locations.

As in the previous experiment, we see that the benchmark scalability characteristics are very different for HTM than STM. With 20-bit keys, the average transaction



(a) Eager STM

(b) Lazy STM

(c) HTM

Figure 4.5: Red-Black Tree experiments augmented with a global counter to monitor the number of elements in the tree.

accesses 20 unique locations, and the tree itself contains approximately a half million objects. Since accesses are random, and the TLB only holds 64 entries, many transactions experience a TLB miss. These misses cause HTM transactions to abort, then serialize, then retry, resulting in poor scaling on the HTM machine.

4.5.3 STAMP Performance

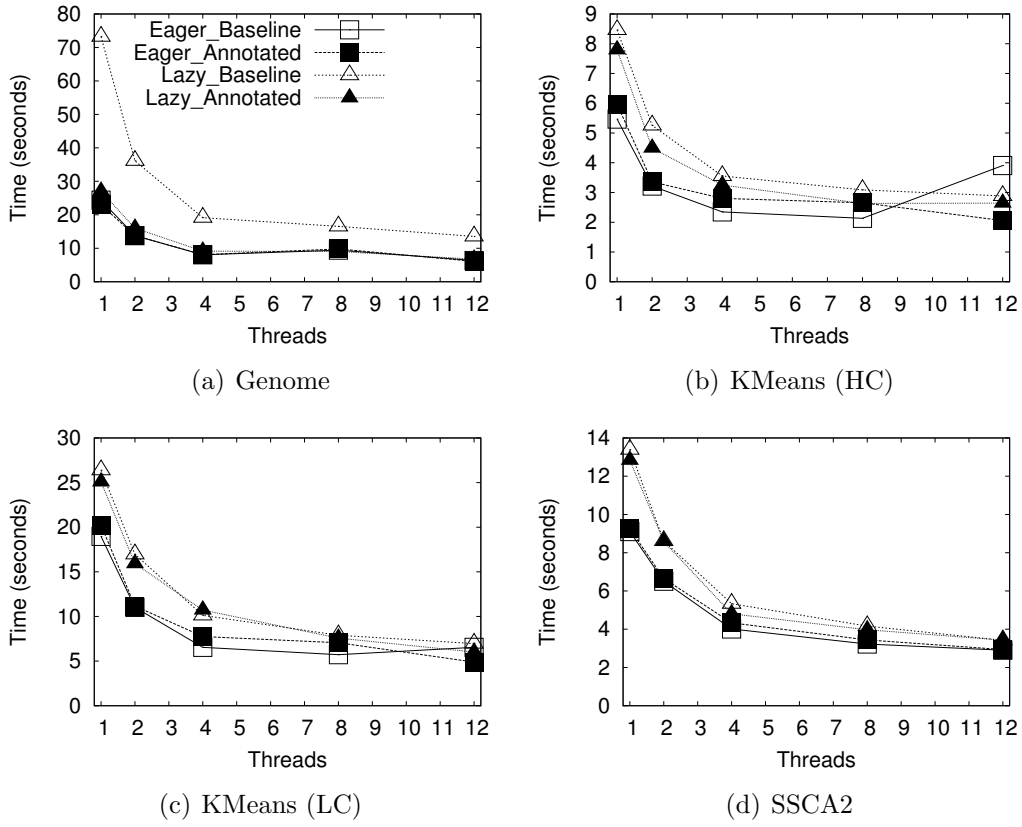
Next, we ran experiments on the STAMP benchmark suite [56]. Table 5.1 describes the frequency of RMW operations in STAMP. We discuss RMWs within the benchmark code separately from RMWs in libraries used by each data structure. Recall that our mechanism works best when transactional RMWs are followed by non-RMW work.

It is clear that the STAMP benchmark suite does not afford many opportunities

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

Table 4.2: Frequency of RMW operations in STAMP benchmarks.

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



(a) Genome (b) KMeans (HC)

(c) KMeans (LC) (d) SSCA2

Figure 4.6: STAMP results on the STM machine [1/2]. HC and LC refer to high- and low-contention command-line configurations.

Note that STAMP uses a library interface to interact with the TM runtime, rather than adhering to the Draft C++ TM Specification. To employ our extensions to GCC, we had to substantially modify the benchmarks, not only to annotate RMW operations, but also to make STAMP compatible with GCC's TM implementation. In particular, we had to remove or replace unsafe function calls and devise alternatives to the non-transactional reads that STAMP sometimes performs from within an atomic transaction. Thus these results do not always correspond directly to prior published work. More details were recently published by Ruan et al. [69].

Figure 4.6 and 4.7 present results for STAMP on the STM system. The "Flag" and "Annotated" benchmarks perform indistinguishably, so we present only the more general "Annotated" algorithm. We see a slight improvement for delayed RMWs in

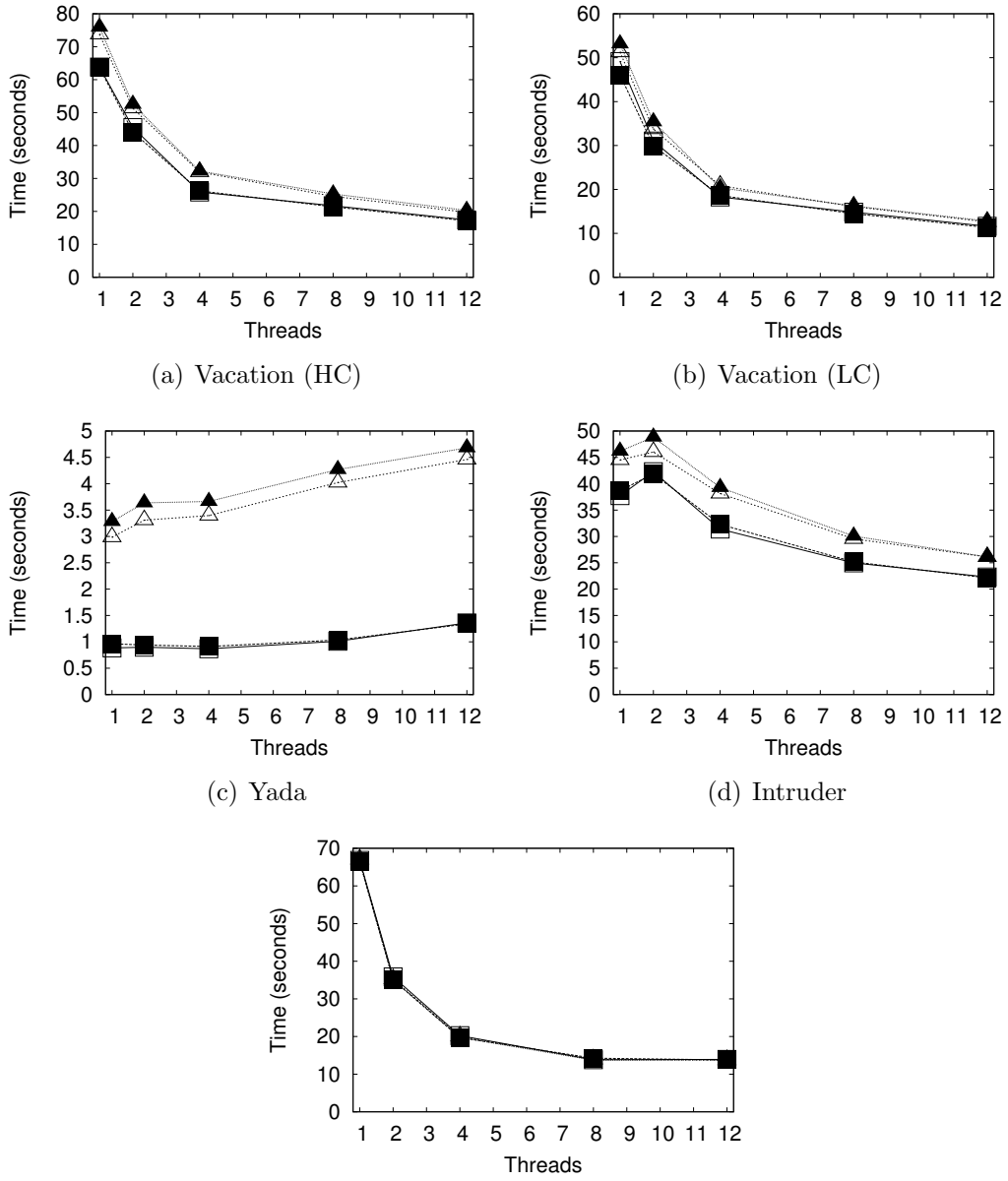


Figure 4.7: STAMP results on the STM machine [2/2]. HC and LC refer to high- and low-contention command-line configurations.

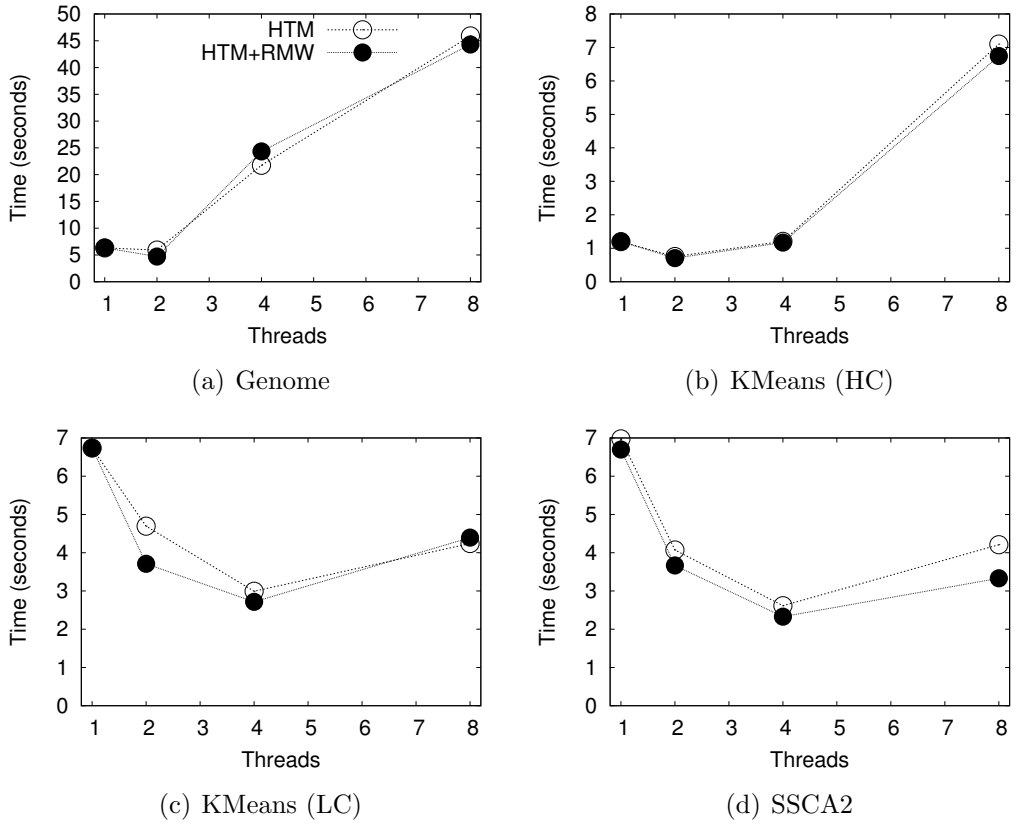
most cases, for both Eager and Lazy STM. KMeans is an outlier: its transactions consist entirely of RMWs, and when contention is low, delayed RMWs can add overhead without avoiding aborts. This is especially true for a lazy STM, where aborts are an order of magnitude less frequent to begin with. Results for Bayes are not

trustworthy, as Bayes exhibits high variability from one run to the next even in the absence of our mechanism, therefore we do not include it in the charts.

Figure 4.8 and 4.9 repeat these experiments on the HTM machine. As with the microbenchmark experiments, we see that HTM trends are not always the same as STM trends. Most noticeably, since our SMT machine has 4 cores and 8 hardware threads, experiments with more than 4 software threads rarely scale: the effective write capacity of HTM transactions is halved when the L1 cache is shared. Furthermore, many transactions serialize even at one thread. These serializations can be due to many factors, including TLB misses (as in Section 4.5.2) and overflowing the capacity of the cache [95]. Naturally, once transactions serialize, delaying RMWs to the end of a transaction offers no benefit.

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

Taken as a whole, we observe that since STAMP transactions do not exhibit the pattern first described in Figure 4.1, the benefit of delaying RMWs is small. While we do not incur noticeable overhead for delaying RMWs, the contention hotspots they are designed to avoid are rare in STAMP, and thus our mechanism cannot substantially improve STAMP performance. We expect the impact to be greater on production systems, where software is more likely to use standard libraries and to employ statistics counters.



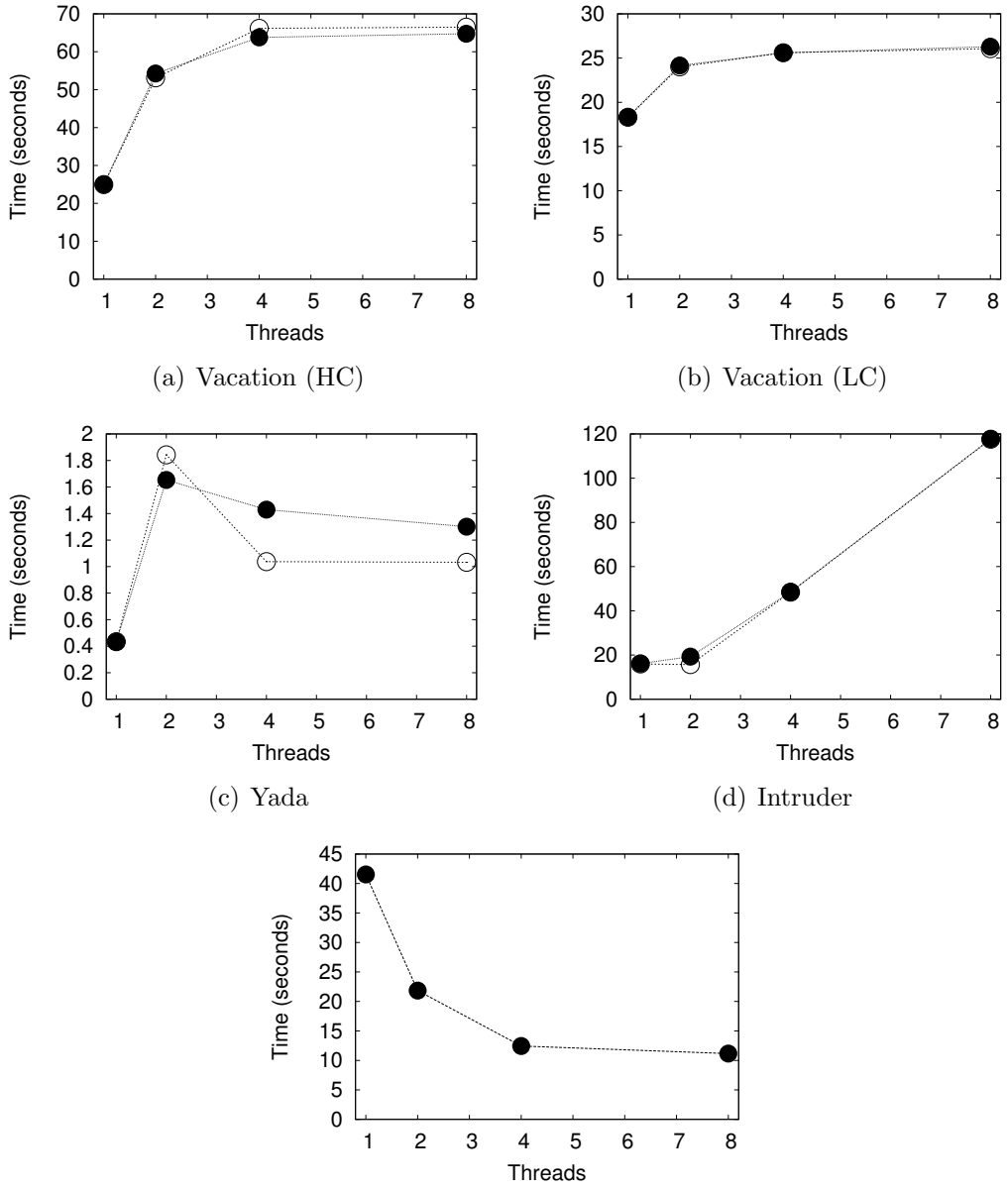
(a) Genome (b) KMeans (HC)

(c) KMeans (LC) (d) SSCA2

Figure 4.8: STAMP results on the HTM machine [1/2]. HC and LC refer to high- and low-contention command-line configurations.

4.5.4 Memcached Performance

Lastly, we look at a real-world application that we expect to possess the attributes lacking from STAMP. We evaluate memcached, following the experiment configuration of Ruan et.al [70]. We use memslap to produce a workload, and run memcached and memslap on the same machine, so as to limit the effect of the network. The configuration results in a number of operations proportional to the number of threads: flat curves indicate perfect scaling, higher values represent slowdown. Note that this configuration results in SMT effects beyond 2 threads on the HTM machine. Consequently, we report only STM results. Performance appears in Figure 4.10. Note, too, that we only instrumented memcached statistics counters, as opposed to all RMWs in memcached. This results in a program that matches the pattern from Figure 4.1.



(a) Vacation (HC)

(b) Vacation (LC)

(c) Yada

(d) Intruder

(e) Labyrinth

Figure 4.9: STAMP results on the HTM machine [2/2]. HC and LC refer to high- and low-contention command-line configurations.

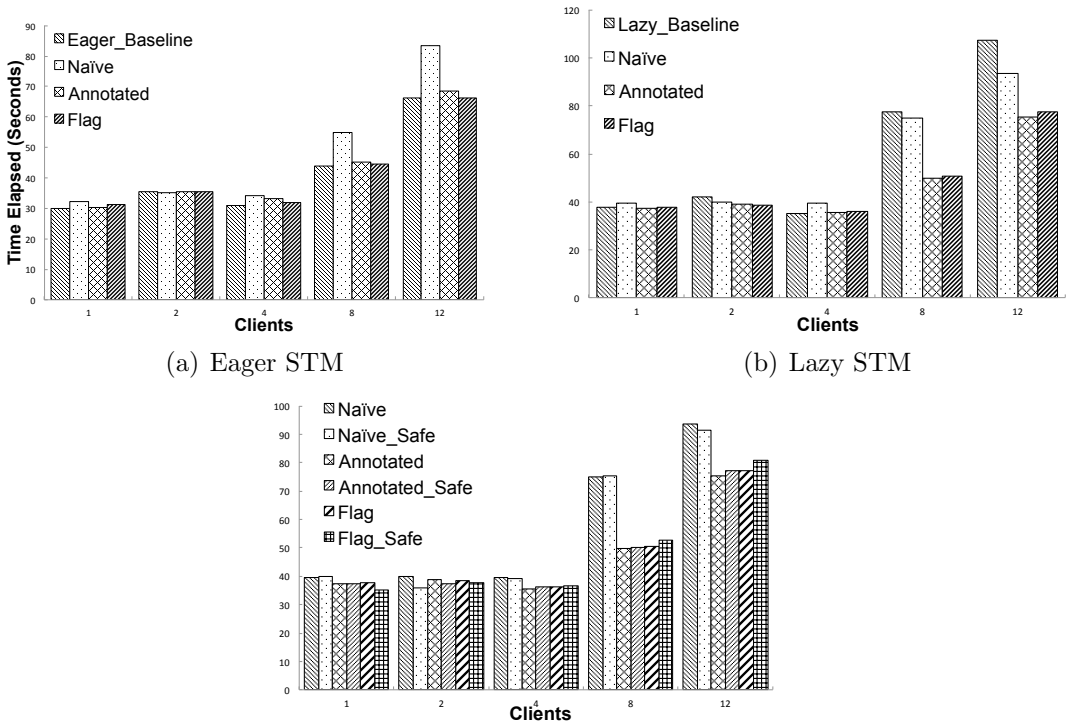
Eager STM Figure 4.10(a) compares the performance of the baseline eager GCC algorithm to three variants using our different delayed RMW algorithms. At low thread counts, we do not observe a significant difference in performance. Starting at 4 threads, the performance of the naïve algorithm becomes noticeably worse than

the others. To gain more insight, we instrumented GCC to report abort rates. At 12 threads, the baseline GCC algorithm reached as high as 20 aborts per commit. We then elided all accesses to statistics counters, to assess the ideal performance (note that the values of statistics counters rarely affect program behavior in memcached). When the counters were removed, we observed neither a change in the abort rate, nor a change in overall running time.

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

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

In a similar manner, the results in Figure 4.10(b) show the consequence of lazi-



(a) Eager STM

(b) Lazy STM

(c) Lazy STM + Publication Safety

Figure 4.10: Memcached performance on a 2-chip, 12-core system.

ness: performance degrades significantly at high thread counts due to wasted work. When two transactions have conflicting accesses, both continue executing up until one commits, at which point the other becomes invalid. Again there is no livelock, but now the problem is that aborts are too infrequent. Indeed, aborts are an order of magnitude less common with the lazy algorithm than with the eager algorithm. However, now speculation is continuing past the point where it could be known that the speculation will not be profitable.

Since transactions speculate past their first read/write conflict, more transactions reach their accesses of statistics counters. This, in turn, allows us to observe the impact of delayed RMW operations. In Figure 4.10(b), even the naïve algorithm improves performance, and algorithms that use annotations do even better, reaching more than 20% improvement at 12 threads. Furthermore, the improvement correlates directly with a decrease in abort rate, which drops from 1.8 aborts per commit without

delayed RMWs to 1.5 aborts per commit at 8 threads. Unlike eager, here aborts typically happen late in the transaction's execution, and every abort prevented is effectively another transaction committed.

Publication Safety Up until this point, the implementations we tested did not include the modifications proposed in Section 4.4, and thus would not be correct if memcached were to use RMW operations in conjunction with variables used for publication of previously private data. However, conducting the evaluation in this manner allows us to measure the best-case performance of delayed RMWs. We now look at the expected case behavior, where support for ELA publication safety is added.

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

Figure 4.10(c) repeats the experiments from Figure 4.10(b), but adds additional bars to show the cost when publication safety is turned on. In memcached, statistics counters are never used for publication, and in fact are never read by the same transactions that update them via RMWs. Consequently, modifying the algorithm to provide safe publication should not affect program behavior. We observe only a slight increase in execution time, due to the increased logging overhead and its associated cache pollution. This cost is negligible in almost all cases.

4.6 Related Work

Contention Management: Contention management (CM) [73, 96, 24, 8, 7] is the most popular approach to resolving conflicts among transactions. While the mechanisms vary greatly, at a fundamental level, CM aims to influence the scheduling of transactions to prevent conflicts. Simple nonblocking contention managers often incorporate backoff, such that when transactions T_A and T_B conflict, one will abort, wait briefly, and restart. Usually this perturbation of the schedule suffices to prevent the conflict from manifesting again. Blocking approaches may instead explicitly deschedule one transaction (e.g., T_B) until the other (T_A) commits. At the extreme point, T_B may be re-scheduled to run on the same processor as T_A, to ensure that no conflict occurs and locality is maximized.

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

Conflict Detection: Similarly to CM, some TM implementations can vary their conflict detection mechanism, such that some transactions use encounter-time locking, and others use commit-time locking [62]. Some implementations vary the locking mechanism on a per-variable basis [79], and others change the global choice of TM implementation based on the presence of high abort rates [67, 45, 92]. Similar mechanisms have even been proposed for hardware TM [77]. In general, these approaches aim to prevent pathology: upon repeated aborts due to conflicts, a trans-

action becomes pessimistic, locking locations eagerly in order to prevent concurrent transactions from interfering. However, as shown in our microbenchmarks, highly contended variables will still result in conflicts regardless of the conflict detection strategy employed by the TM. Thus, as with CM, our recommendation is that these approaches be viewed as complementary to our work. When transactions experience pathology, or when false sharing is causing conflicts over ownership records, then increasing pessimism or changing algorithms can improve performance.

In a related manner, Zyulkyarov et al. [99] developed tools for identifying contention in transactional programs, and then used this information to rewrite code to decrease contention. We believe that the same analysis could guide the annotation of variables involved in RMW-based conflicts, as an alternative to ad-hoc programmer techniques or the live-out analysis described in Section 4.3.

Nesting: Both open and closed nesting [60, 61, 59] can improve performance in the face of highly contended variables. For example, when a hot counter is incremented in a closed nested transaction that comprises the tail of a long-running parent, then conflicts on the counter may require only the nested child transaction to restart. While limited to the case where the hot counter is incremented at the end of the transaction, this approach still avoids much of the cost of aborts due to hot variables.

Similarly, if the hot counter was incremented via an open-nested transaction, then most conflicts on the counter simply would not manifest. With open nesting, the increment would occur immediately, and become visible to concurrent threads. If the parent transaction subsequently aborted, then some programmer-specified compensating action would undo the increment. While the infrastructure for supporting delayed operations resembles the infrastructure for registering undo actions, there is a fundamental difference: our delayed operations do not break atomicity, and thus can be invisible to the programmer. In contrast, open nesting often requires the

programmer to employ ad-hoc abstract locking [34] to prevent concurrent transactions from reading an incorrect counter value (e.g., if the counter is used to detect an empty collection). Additionally, our mechanism supports accesses to the counter after a delayed RMW, which can be difficult with open nesting [61, 5].

An aggressive approach to closed nesting that preserves atomicity is to use Abstract Nested Transactions (ANTs) [33]. ANTs are similar to closed nested transactions, except that when they complete, they are not merged into the parent transaction. Should the parent later detect a conflict, then if the conflict is localized to an access performed within the ANT, the ANT can often be rolled back and re-executed without requiring the parent transaction to abort. This, of course, succeeds only when the parent does not read locations modified by its ANTs. Our work can be thought of as (a) demonstrating the value of ANTs, (b) providing a practical implementation of small ANTs for unmanaged languages, and (c) introducing the run-time mechanisms necessary for resolving parent accesses to values modified by its ANTs. Additionally, our work identifies and resolves questions related to semantics that had not yet been identified when ANTs were proposed.

Non-Atomic Updates: As a last resort, operations on hot counters could, in some cases, be performed outside of transactions. For example, the Atomos language [13] and the TM proposed by Ni et al. [62, 13] both allow transactions to register "onCommit" functions whose execution is delayed until after the transaction commits. These functions do not execute within the context of the transaction, and their accesses to shared data must be manually synchronized. Depending on the implementation, they may be able to use transactions themselves. Clearly such an approach is not appropriate for the general case, where a transaction might read a hot variable after incrementing it. However, when precise counts are not required by the application logic, and when these hot variables are not used in other ways by the

parent transaction, deferring their update until after commit, via simple "onCommit" routines, may be a viable alternative to the mechanisms proposed in this chapter.

4.7 Summary

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

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

Chapter 5

Exploring Collaborations between Software and Hardware Transactions

In this chapter, we present a new hybrid TM that uses Intel Haswell TSX as hardware fast path and STM "Cohorts" as software slow path. The original work "Hybrid Transactional Memory Revisited" was accepted as a regular paper to appear at the 29th International Symposium on Distributed Computing, October 2015.

5.1 Introduction

Despite the efforts we put into improving STM performance in the previous chapters, the dominating latency for most of the scalable STM implementations are logging and heavily instrumented conflict detection mechanisms, which HTM naturally does not introduce. The recent addition of hardware TM support to IBM [40, 89] and Intel [39] processors brings the field of concurrent programming much closer to a state in which programmers can eschew locks in favor of transactions.

However, first-generation hardware TM systems carry a number of limitations.

Most significantly, these implementations are "best effort" [44], in that they do not guarantee that any transaction attempt will commit. In particular, a transaction attempt may fail if it accesses more unique locations than the hardware can support, or if there is an interrupt (e.g., a timer interrupt) during its execution. Consequently, a TM runtime that wishes to use hardware TM must provide a software fall-back path. This fall-back path also provides a means of circumventing the hard-coded conflict resolution strategy ("requester wins" [10]) that the hardware enforces, so as to allow the run-time system to improve the chance that a long-running transaction does not starve.

Broadly speaking, TM runtime systems that combine the use of hardware TM with a software fall-back path are called hybrid TM [58]. Existing hybrid TM proposals can be categorized as follows:

- Low-Scalability Fall-back: Lev's PhaseTM [45] was among the earliest hybrid TMs. While it envisioned a variety of different ways to compose hardware and software transactions, it required that all transactions used the same technique at the same time (i.e., all use hardware, or all use a software TM algorithm). Of the many approaches, the combination of hardware TM with a single-lock fall-back was perhaps the most straightforward [15], and has subsequently been improved, e.g., by Calciu et al. [12].

- Scalability Through Non-Transactional Actions: The systems by Dalessandro et al. [17] and Riegel et al. [68] both assumed that the underlying hardware TM would allow non-transactional operations within a transactional context (for reading and writing, respectively).

- Hybrid TM-Specific Hardware: proposals by Minh et al. [57], Shriraman et al. [78], and Saha et al. [72] assumed that the hardware TM would provide a wide API so that a hybrid run-time system could use parts of the hardware

(e.g., tracking cache invalidations of specific lines) to accelerate software TM. The hardware for these systems is not currently available, nor does it appear in any product roadmaps.

- Reduced Hardware Capacity: Systems by Kumar et al. [43], Damron et al. [19], and Riegel et al. [68] required all hardware transactions to access the per-location metadata used by the software TM fall-back. This approach can improve the concurrency between hardware and software transactions, but it effectively halves the capacity of the hardware, and is largely viewed as impractical.

- Unsafe Hybrid TM: The Invyswell system [11] reduces the safety of hybrid transactions by sacrificing opacity [29]. The resulting system cannot guarantee correctness in the face of certain patterns [20], but can scale well on existing systems.

- Behavior-Specific Hybrid TM: Reduced Hardware NOrec [53] ensures opacity and is compatible with existing hardware TM. However, its performance relies upon transactions following a specific pattern, in which there is a large read-only prefix before the transaction's first attempted write. While appropriate for data structures, this may not be a suitable approach for realistic applications.

From an architectural perspective, we believe it unlikely that vendors will extend future micro-architectures with hybrid TM features or add non-transactional actions. However, it is likely that future hardware TM may overcome its existing capacity constraints (e.g., by expanding the capacity/associativity of private caches, or by moving conflict tracking structures higher in the cache hierarchy). Thus we believe that the most important qualities of a hybrid TM are to provide a safe programming model, to minimize the use of hardware capacity for tracking metadata, and to emphasize fairness and progress for transactions that fall back to software.

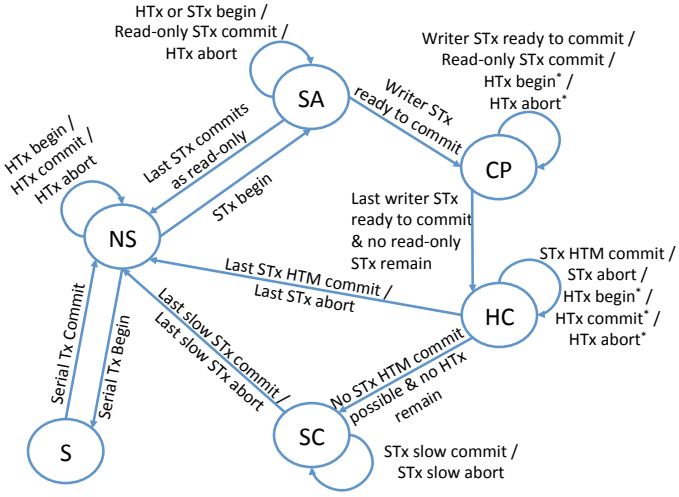
To provide these properties, we introduce the Hybrid Cohorts (HyCo) algorithm. Based on the Cohorts algorithm we discussed in Chapter 2, HyCo uses a state machine to manage the behavior of transactions. By guaranteeing the immutability of memory during any software transaction's execution, and employing hardware TM as broadly as possible, HyCo minimizes instrumentation for all transactions, and eliminates many of the bottlenecks of the original Cohorts algorithm, without sacrificing safety.

The remainder of this paper is organized as follows. In Section 5.2, we discuss the overall approach of the Hybrid Cohorts algorithm, with a focus on the state machine that governs transaction behavior. Section 5.3 presents the pseudocode for one implementation of the state machine, which aims to limit the impact on transactions that use hardware TM resources throughout their execution. In Section 5.4, we present the results of performance experiments. Section 5.5 concludes and discusses some future research directions.

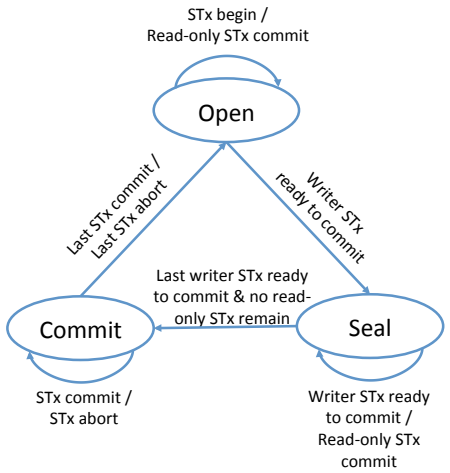
5.2 The Hybrid Cohorts Algorithm

The foundation of the HyCo algorithm is a state machine that governs when transactions may begin, as well as when and how they commit. This state machine appears in Figure 5.1(a). For reference, the original Cohorts state machine is provided in Figure 5.1(b).

In the original Cohorts algorithm, the role of the state machine was to ensure that memory remained constant whenever a transaction was in-flight (i.e., between its begin and end points). This entailed blocking writing transactions from committing whenever a transaction was in-flight, and blocking transactions from beginning whenever a transaction was committing. The Cohorts algorithm also assumed that a transaction requiring irrevocability [94, 87] (e.g., in order to perform I/O with transactional data) could do so by starting directly in the commit state.



(a) State transitions of the Hybrid Cohorts algorithm. STx refers to a software-mode transaction, and HTx refers to a hardware-mode transaction. The lack of a label on an arc indicates that a transaction behavior is either impossible or not allowed. For example, an HTx is not allowed to commit in the SA or CP states, and it is not possible for an STx to abort in these states.



(b) State transitions of the Cohorts algorithm. The Open, Seal, and Commit states correspond to the SA, CP, and SC states of Hybrid Cohorts.

Figure 5.1: State transitions for the Hybrid Cohorts (top) and Cohorts (bottom) algorithms.

In our new algorithm, we begin by formalizing irrevocability through the addition of a "serial" state (S). We then split the entry state (Cohorts::Open): instead of indicating that software transactions may be active, the split state distinguishes between when at least one software transaction is active (SA), and when no software transactions are running (NS). The commit pending state (CP) is equivalent to the Cohorts::Seal state. Finally, the Cohorts::Commit state is split, so that one-at-a-time slow commit (SC) can be avoided via a HTM-assisted commit phase (HC).

The original Cohorts algorithm also exposed options for how to detect conflicts, to include the use of ownership records [71, 21] or values [63, 18]. In HyCo, we exclusively use value-based conflict detection. To use other metadata would necessitate the use of HTM resources for concurrency control, which would, in turn, reduce the size above which transactions must run in software mode.

The algorithm affords a number of implementation choices and options. For example, the labels marked with an asterisk(*) correspond to a variant in which more hardware-mode transactions (HTx) are allowed. Similarly, there are a variety of ways to choose the order in which transactions perform their slow commit, depending on contention management [73] policies. For this discussion, we assume that the contention manager randomly chooses the order in which transactions attempt to commit.

5.2.1 Transitions

The initial state of the system is NS, indicating that no software or serial transactions are running. Should a transaction require serial-mode execution, it does so by transitioning from the NS state to the S state. This transition may entail either (a) forcibly aborting any in-flight hardware transactions, or (b) setting a flag to prevent subsequent HTx and STx transactions from beginning, and then waiting for the system to be in the NS state with no HTx transactions running. Implementation details

for achieving this transition appear in Section 5.3.

When a transaction is in serial mode, it is not allowed to abort, and no other transactions may execute. When the transaction commits, the system transitions back to the NS state.

In the NS state, there are no STx transactions running. Thus as long as the system remains in NS state, HTx transactions may execute in their entirety, either committing or aborting and retrying. However, as soon as an STx begins, the system transitions from NS to SA. In SA, new hardware and software transactions may begin. However, hardware transactions may not commit: they must abort or wait if they reach their commit point while the system is in the SA state. STx transactions accumulate their reads and writes in thread-private logs, with all writes buffered until commit time. As in the Cohorts algorithm, a read-only STx (detected by its empty write log) can commit directly from the SA state, since it does not modify memory. This may transition the system back to NS, if it results in all remaining transactions being HTx.

From the SA state, as soon as the first writing STx is ready to commit, the system transitions to the Commit Pending (CP) state. From this state, additional read-only STx may commit, writing STx may announce that they are ready to commit, and HTx may begin or abort. Note that none of these transaction behaviors can affect the in-flight STx, since these behaviors do not affect shared memory.

When the last STx reaches the CP state, HyCo transitions to the HTM-assisted Commit state (HC). Any in-flight HTx transactions are permitted to commit immediately; all STx transactions use a hardware transaction to first validate their read set, and if it has not changed, to replay all writes from the thread-private log. Note that when an HTx transaction aborts, it can retry immediately, as can a hardware transaction attempting to commit the STx. However, if the STx validation fails, then the STx does not retry until the system returns to the NS state.

If all STx can commit or abort from the HC state, then the system transitions back to NS, even if HTx transactions are still executing. However, if any STx cannot commit via HTM (e.g., due to its read and write sets being too large to traverse and replay in a hardware transaction), then once there are no further STx attempting to commit in HTM, and no remaining HTx, the system transitions from HC to SC, where STx transactions commit sequentially. As in the S state, some effort is needed to block HTx transactions from beginning, or else this transition may be delayed indefinitely. Once the transition occurs, the remaining STx are guaranteed that (a) no new transactions can start, and (b) no other transactions are attempting to commit. Thus the STx can, in turn, validate their read sets and then either abort or write-back their updates. Once all pending STx have done so, the system returns to the NS state.

5.2.2 Key Properties

Earlier, we argued that a hybrid TM should ensure safety, limit use of hardware capacity for tracking metadata, and should enable some sort of fairness and progress for STx transactions. We briefly discuss each of these points in relation to the HyCo algorithm below:

Safety: The HyCo algorithm provides opacity [29] for all transactions. In Cohorts, opacity is achieved by ensuring that all shared memory is immutable whenever a transaction is in-flight. In HyCo, where there are two flavors of transaction, we modify this criteria: when a STx is in-flight, no concurrent HTx or STx transaction may perform an operation that modifies locations that have been, or may be, read by the in-flight STx. A concurrent STx transaction may progress up to its commit point, and may create pending changes to memory via the `TxWrite` function (as in Algorithm 24). However, it may not transition to the HC or SC state. Thus the

concurrent STx cannot perform an operation that changes the memory visible to the in-flight STx. In this case, the property is achieved through the write buffering performed by STx. Similarly, a concurrent HTx may not transition to the HC state, where it can complete its transaction. Since HTx writes are buffered by the hardware until the commit point, the HTx cannot affect the behavior of the concurrent STx.

Now let us turn to an HTx transaction. Dalessandro et al. established that in a lazy Hybrid TM, an HTx transaction can experience an opacity violation if it overlaps with a concurrent STx commit [17]. The specific issue they identified is that a lazy STx might perform a partial write-back concurrent with the HTx, so that the HTx reads some of the STx's committed state, but not all of it. More generally, a sufficient condition is to prevent incomplete STx write-back from being visible to an HTx execution. In HyCo, this is achieved by (a) forbidding an STx from reaching the SC state until there are no concurrent HTx, and (b) attempting to commit STx in the HC state. In the HC state, the STx uses a hardware transaction to both validate and perform write-back; consequently the STx cannot expose its partial state: the entire set of updates becomes visible when the hardware transaction commits.

Metadata: As discussed above, HyCo does not use per-location metadata. Instead, it tracks the values read by a STx, and then validates those values directly. In this manner, it does not spend precious HTM resources tracking metadata. As we will show in Section 5.3, the state machine can be implemented in a variety of ways, but the only global metadata for HyCo is related to the state machine, and it is only accessed at transaction boundaries. This results in a constant amount of metadata, and a constant overhead to access that metadata, for HTx.

Fairness and Progress: HyCo supports a variety of approaches to ensuring fairness and progress. A few properties are relatively obvious: any transaction can be guaranteed to complete if it executes in Serial mode, and every read-only transac-

tion will complete on its first attempt if it executes in STx mode. Beyond this, HyCo increases fairness by limiting the conditions in which a transaction cannot make progress. In particular, we have taken care to allow HTx to begin and commit when an STx is committing via HTM (HC state). Coupled with the simple existence of HC state, this limits the situations in which the system serializes. In addition, our default HyCo implementation exposes two knobs for tuning progress. The first is a count of the number of HTx aborts before falling back to STx mode. The second is a count of the number of STx aborts before falling back to Serial mode. When combined with optional contention management at the beginning of the HC and SC states, there is ample opportunity to ensure that the most advantageous transactions are given priority. Additional scheudling decisions can be made when transitioning out of the CP state (i.e., by allowing a high priority transaction to abort all HTx, transition directly to SC, and commit first). To the best of our knowledge, HyCo is the first hybrid TM to offer this level of contention management support.

5.3 Implementation

The primary challenge in implementing HyCo is to achieve a low-latency implementation of the state machine from Figure 5.1(a). The most natural solution is to track each thread's state in a thread-private variable. However, doing so results in high latency in the common case: an HTx must check $O(\#Threads)$ locations at begin time. On the other hand, implementing each state as a counter is also a poor choice, since certain counters become contention hot-spots.

Our solution, presented in Listing 1, is to split the state machine into three parts. First, there is a list of `Thread` objects, through which per-thread states for non-transactional, Serial, HTM, and STM mode can be discerned. This list is employed by all transactions. Second, we use an Integer and three Booleans to control when

Listing 1: Hybrid Cohorts metadata. Global variables are clustered according to whether they assist in (a) coordinating all transactions, (b) coordinating HTx transactions, or (c) coordinating STx transactions.

```
Thread Variable Type:
  tx_state   : Enum{NO, S, HW, SW}   // state of thread's transaction
                                     // (nontransactional, serial, HTx, STx)
  writes     : Map<addr,val>         // write set if this transaction is in STx mode
  reads      : Set<addr,val>         // read set if this transaction is in STx mode
  my_order   : Integer               // commit order of this transaction if it is in
                                     // STx mode and using serial commit (SC)
  cp         : Checkpoint            // checkpoint of thread state, for retrying
                                     // after STx aborts.
Global Variables:
  threads    : Set<Thread>       // A way of reaching each thread's per-thread vars

  started    : atomic <Integer>  // Count of current active STx transactions
  ser_kill   : atomic <Boolean>  // Flag to allow a Serial transaction to force
                                 // immediate HTx aborts
  stx_kill   : atomic <Boolean>  // Flag to allow an STx in SC mode to force
                                 //immediate HTx aborts
  stx_comm   : atomic <Boolean>  // Indicate that all STx are ready to commit

  cpending   : atomic <Integer>  // Count of STx that are in the CP state
  order      : atomic <Integer>  // Counter for ordering any STx that require SC
                                 // mode to commit
  time       : atomic <Integer>  // Second counter for STx that require
                                 // SC mode to commit
  serial     : atomic <Boolean>  // Token for granting a transaction permission
                                 // to run in Serial mode
```

HTx can begin, and when they must immediately abort. Finally, three Integers and one Boolean are used to manage the states of STx and Serial transactions.

HTx Behavior: Algorithms 20- 22 describe how HTx, STx, and Serial transactions use these variables to safely transition among states. The default state is NS, in which HTx may begin and commit. Departing from this state requires an STx or Serial transaction to begin. To keep overheads low for HTx, we subscribe to the *ser_kill* flag when an HTx begins. After becoming serial, but before accessing shared memory, a Serial transaction sets this flag to immediately abort all HTx. By optionally using the *threads* set first (`TxBeginSerial` lines 5-6), we can opt to prioritize running HTx

Algorithm 20: Begin and end instrumentation for HTx transactions. Parameters to `xabort` indicate the line to jump to after canceling a transaction attempt.

```
1  function TxBeginHTx()
       // Announce active HTx
2      tx_state ← HW
3      _xbegin
       // Detect Serial and STx-SC transactions
4      if ser_kill ∨ stx_kill then
5          xabort(6)
6      return
       // Wait until no Serial or STx-SC transactions
7      tx_state ← NO
8      while ser_kill ∨ stx_kill do spin
       // Note: option to change to STx or Serial would go here
9      goto Line 2

1  function TxCommitHTx()
       // Commit if all STx in HC mode or no STx
2      if stx_comm ∨ started = 0 then
3          _xend
4          tx_state ← NO
5          return
       // Cannot commit: in-flight STx or STx in SC mode
6      xabort(TxBeginHTx :: 6)
```

over new Serial transactions.

Since HTx can execute concurrently with STx, we do not repeat this behavior when STx begin. Instead, we must ensure that HTx do not commit when either (a) STx are between their begin and end, or (b) STx are performing serial commit. The *stx_kill* flag expresses condition (b). To handle condition (a), we use the *started* and *cpending* counters. When they are equal, every STx transaction has reached its commit point, and are trying to commit using HTM. In this case, HTx can commit, since the HTM will mediate conflicts. However, if they differ, then the HTx must abort.

Algorithm 21: Begin instrumentation for STx transactions.

```
1  function TxBeginSTx()
2      cp ← make_checkpoint()
       // Try to set started while ¬serial and cpending = 0
3      if ¬serial then
           // Wait for committing STx, then announce self
4          while cpending > 0 do spin
5          atomic_incr(started)
           // Double-check that it's safe to start
6          if cpending > 0 ∨ serial then
7              atomic_decr(started)
8              goto 2
9          tx_state ← SW
           // Lazy cleanup of STx-SC flag
10         if stx_comm then stx_comm ← false
11     else goto 2
```

STx Behavior: STx are expected to be less frequent than HTx, and also to be longer-running. Thus we tolerate some contention over metadata, since it reduces the number of locations that HTx must check. Specifically, we use the *started* counter to track the number of STx that are not yet committed, and *cpending* to track the number of STx that have reached their commit point. The *order* and *time* counters are used only for SC commits, to enforce one-at-a-time commit of large STx.

To maximize HTx concurrency with STx, we do not eagerly inform HTx of transitions between NS, SA, CP, and HC. Instead, we use the *stx_comm* flag, which indicates that STx have moved to HC state. While two values are needed to manage the SA-CP-HC transition, this specific pattern avoids aborts for HTx, since *started* changes infrequently when *stx_comm* is set.

The additional transition to SC for serialized commit of STx is expected to be rarest. We employ the same technique as Serial transactions, where a flag (*stx_kill*) is coupled with a traversal of the *threads* set (`TxCommitStx` lines 23-24) to allow HTx to complete before serial STx.

Algorithm 22: End instrumentation for STx transactions.

```
1  function TxCommitSTx()
       // Read-only fast path
2      if writes = ∅ then
3          atomic_decr(started)
4          reads ← ∅
5          return
       // Wait until all STx ready to commit
6      atomic_incr(cpending)
7      while cpending < started do wait
       // STx will try to commit via HTM
8      if ¬stx_comm then
9          stx_comm ← true;
10     xbegin
11     if reads.validate() then
12         writes.writeback()
13         xend
14         atomic_decr(started)
15         atomic_decr(cpending)
16         reads ← writes ← ∅
17         tx_state ← NO
18         return
19     else xabort(36)
       // STx couldn't commit via HTM.
       // Use serialized commit
20     my_order ← atomic_incr(order)
       // Lead thread waits for HC phase to
       // end, others wait their turn
21     if order = 0 then
22         while order < started do spin
           // Optional: allow HTx to complete
23         for tx ∈ {threads − this_thread} do
24             wait_until(tx.tx_state ≠ HW)
           // Interrupt remaining HTx
25         stx_kill ← true
26     else
27         while time ≠ my_order do spin
       // Writeback only if validation succeeds
28     if reads.validate() then
29         writes.writeback()
30     else failed ← true
       // Let next STx commit
31     time ← time + 1
       // Clean up SC metadata
32     old ← atomic_decr(started)
       // Extra work for last thread
33     if old = 1 then
34         stx_kill ← false
35         time ← order ← 0
36     atomic_decr(cpending);
37     tx_state ← NO
38     reads ← writes ← ∅
39     if failed then cp.restore()
40     else return
       // Reachable only on HC validation
       // failure
41     atomic_decr(started);
42     atomic_decr(cpending);
43     reads ← writes ← ∅
44     tx_state ← NO
45     cp.restore()
```

A final complication is that, for the sake of fairness, we do not allow new STx to begin once any STx is ready to commit writes. This necessitates care in `TxBeginSTx`, since we must double-check *cpending* after incrementing *started*.

Serial Behavior: Serial transactions are expected to be least common, and thus we are willing to incur overhead whenever one begins. In particular, after acquiring the *serial* token, a transaction will wait for all active STx and HTx to complete.

Algorithm 23: Begin and end instrumentation for Serial transactions

```
1 function TxBeginSerial()
      // Acquire serial lock
2     while ¬bool_cas(serial, false, true) do spin
3     tx_state ← S
      // Wait for committing STx
4     while started > 0 do spin
      // Optional: allow HTx to complete
5     for tx ∈ {threads − this_thread} do
6         wait_until(tx.tx_state = NO)
      // Interrupt remaining HTx
7     ser_kill ← true

1 function TxCommitSerial()
      // Release lock, re-enable HTx
2     ser_kill ← false
3     serial ← false
4     tx_state ← NO
```

By setting the *serial* flag first, it effectively prevents new STx. After allowing HTx to complete, it sets *ser_kill* to prevent additional HTx, at which point it can begin. Both flags are cleared when the transaction completes.

Per-Access Instrumentation: For completeness, Algorithm 24 presents the read and write instrumentation for the HyCo algorithm. As in the original Cohorts algorithm, per-access instrumentation is minimal, entailing neither metadata access nor memory fences. This is because (a) memory is immutable during STx execution, (b) Serial transactions execute in the absence of concurrency, and (c) HTx conflicts are mediated through the HTM, not through metadata.

5.4 Evaluation

In this section, we evaluate the performance of HyCo. We consider microbenchmarks, the STAMP benchmark suite [56, 69] and a transactionalized version of Mem-

Algorithm 24: Hybrid Cohorts read and write instrumentation

```
1 function TxRead(addr)
      // Serial and HTM fast-path
2     if tx_state ∈ {S, HW} then
3         return *addr
      // Handle read-after-write
4     if addr ∈ writes then
5         return writes[addr]
      // Read the value, and log it for commit-time validation
6     v ← *addr
7     reads ← reads ∪ {⟨addr, v⟩}
8     return v

1 function TxWrite(addr, val)
      // Serial and HTM fast-path
2     if tx_state ∈ {S, HW} then *addr = val
      // Buffer the write until commit time
3     else writes ← writes ∪ {⟨addr, v⟩}
```

cached [70]. Experiments are conducted on a machine with single-chip 3.40GHz Intel Core i7-4770 with 4 cores / 8 threads, running Ubuntu Linux 13.04, kernel 3.8.0-21, and a 4.9 GCC compiler with O3 and m64 flags. Results are the average of 5 trials.

We compare the following TM implementations:

- **STM_Eager** is the default STM implementation provided with GCC (known as ml_wt). It is based on TinySTM's write-through algorithm [25]: write locks are acquired eagerly upon first write access to a location, undo logs track changes made by transactions, in case of an abort, and reads check the version number of locks. Conflicts are detected via validation, and a global counter is used to avoid most validation during transaction execution. Writer transactions use *quiescence* to achieve privatization safety.

- **STM_Lazy** is a commit-time locking version of STM_Eager. Writes are stored in a redo log, which is implemented as a hash table of 64-byte blocks. Write locks are acquired at commit time. In all other regards, the implementation is

the same as STM_Eager. The main value of STM_lazy in our experiments is in identifying overheads related to redo logs.

- **HTM:** a) **HTM** is the default HTM implementation provided with GCC. Transactions attempt to run using Intel RTM, and fall back to a serial execution mode after two consecutive HTM aborts. b) **HTM_20** modifies the above HTM implementation so that fallback to serial mode occurs after 20 attempts.

- **HyNOrec:** There are two suggestted implementations that do not require non-transactional reads in the original HyNOrec proposal [17]. We present the P-counter version (where P equals to the number of threads) in microbenchmarks and the STAMP suite, as it outforms the 2-location version. In Memcached however, we report both.

- **HyNOrec_RH** is the most recent Reduced Hardware Hybrid NOrec implementation adopted from [54]. We did not apply the complier static analysis to reduce the instrumentation of read-only hardware transactions, for fair comparison with other TM implementations, which could all benefit from such analysis.

Our version of HyCo employs the following optimizations:

- **Lightweight Privatization Safety**: Since writer transactions either (a) commit via HTM, or (b) commit during the serialized (SC) phase, there is no need for out-of-band privatization safety. Our HyCo implementation thus skips GCC's quiescence mechanism.

- **Lightweight Irrevocability**: GCC achieves serial execution via adaptivity, which requires coordination among all transactions via a readers/writer futex. In contrast, Serial mode is a first-class behavior within HyCo, requiring no additional overhead on every transaction.

- **Un-instrumented HTM Loads and Stores**: GCC creates two code paths for transactions: one suitable for STM, in which loads and stores of shared memory are transformed into function calls, and one suitable for HTM, in which loads and stores are not instrumented. Given the lightweight instrumentation in Algorithm 24, HyCo is able to use the latter approach for HTx.

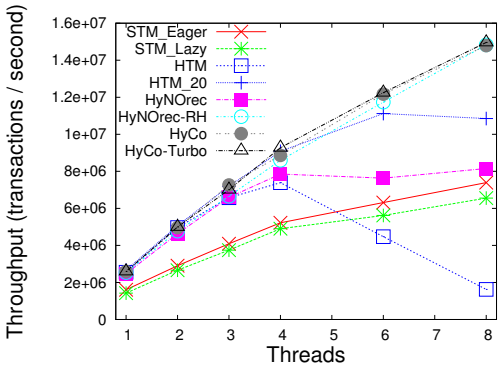
We also present HyCo-Turbo, which additionally provides a lightweight software path. Since it is natural for a STx to be aware of the number of existing software transactions, a STx can change to turbo mode by sealing the cohort early if it a) confirms that no other STx is running and b) successfully aborts all running HTx. A turbo mode STx does not require further instrumentation on reads/writes, or validation at commit time.

We set HyCo thresholds as follows: An HTx transaction will switch to STx mode after 20 failed attempts to commit. An STx transaction will switch from committing in HC mode to committing in SC mode after 2 failed attempts. Fall-back to Serial mode occurs after 5 failed commit-time validations by an STx transaction.

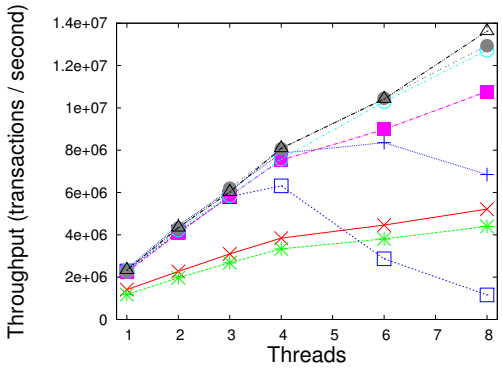
5.4.1 Microbenchmark Performance

We begin our evaluation by looking at microbenchmark performance. We consider four configurations of a red-black tree test, taken from the RSTM library [50]. Configurations differ in terms of the range of keys present in the tree, and the ratio of lookups to inserts and removes (insert and remove operations are always performed in equal amounts). In all cases, the tree is pre-populated to 50% full. The charts in Figure 5.2 present throughput as the average over five trials.

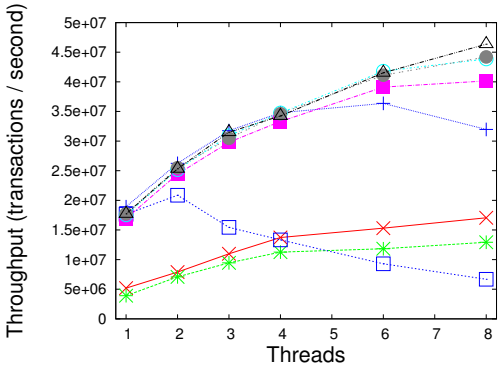
At one thread, HTM and HyCo performance are identical, and uniformly better than STM. This is expected, since transactions are small enough to complete without exceeding hardware capacity. As we increase the thread count, and contention increases, we see a significant shift: the rapid fall-back to serial mode hurts HTM,



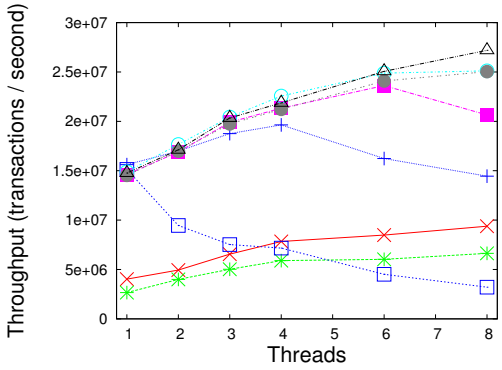
(a) Red/black tree microbenchmark with 20-bit keys and 80% lookup ratio



(b) Red/black tree microbenchmark with 20-bit keys and 33% lookup ratio



(c) Red/black tree microbenchmark with 8-bit keys and 80% lookup ratio



(d) Red/black tree microbenchmark with 8-bit keys and 33% lookup ratio

Microbenchmark	NS	HTx:HC	STx:RO	STx:HC	STx:SC	Serial
20-bit / 33% lookup	1.96M	50.4K	44	37	9	0
20-bit / 80% lookup	2.09M	7.6K	72	8	7	0
8-bit / 33% lookup	5.00M	381K	569	180	70	28
8-bit / 80% lookup	9.32M	25K	1.98K	344	330	14

(e) Frequency of each type of commit for four microbenchmarks with HyCo. Data is taken from a one-second execution with four threads. The workload was heterogeneous, and values were reported by a randomly chosen thread.

Figure 5.2: Microbenchmark performance

both because it is too early, and because it limits concurrency. Even HTM_20, our version of the GCC HTM that retries 20 times before falling back to serial mode, cannot keep up with HyCo: the opportunity cost of serialization, even after 20 failed attempts, is simply too high. This is especially true for the highest contention configuration (8-bit keys, 33% lookup), where HTM_20 performance degrades beyond 4 threads.

The performance of eager and lazy STM was also surprising in this experiment. As expected, both scale well, and their use of validation affords for fewer aborts than the "requester wins" conflict resolution strategy [10] of HTM. However, latency is high: they incur a function call on every load and store, and lazy pays even more due to accesses to the write log on every load and store (these costs are only incurred in HyCo's STx mode). Furthermore, STM scales worse than HyCo. There are two causes: the overhead of quiescence, and the cost to support irrevocability via mode switching.

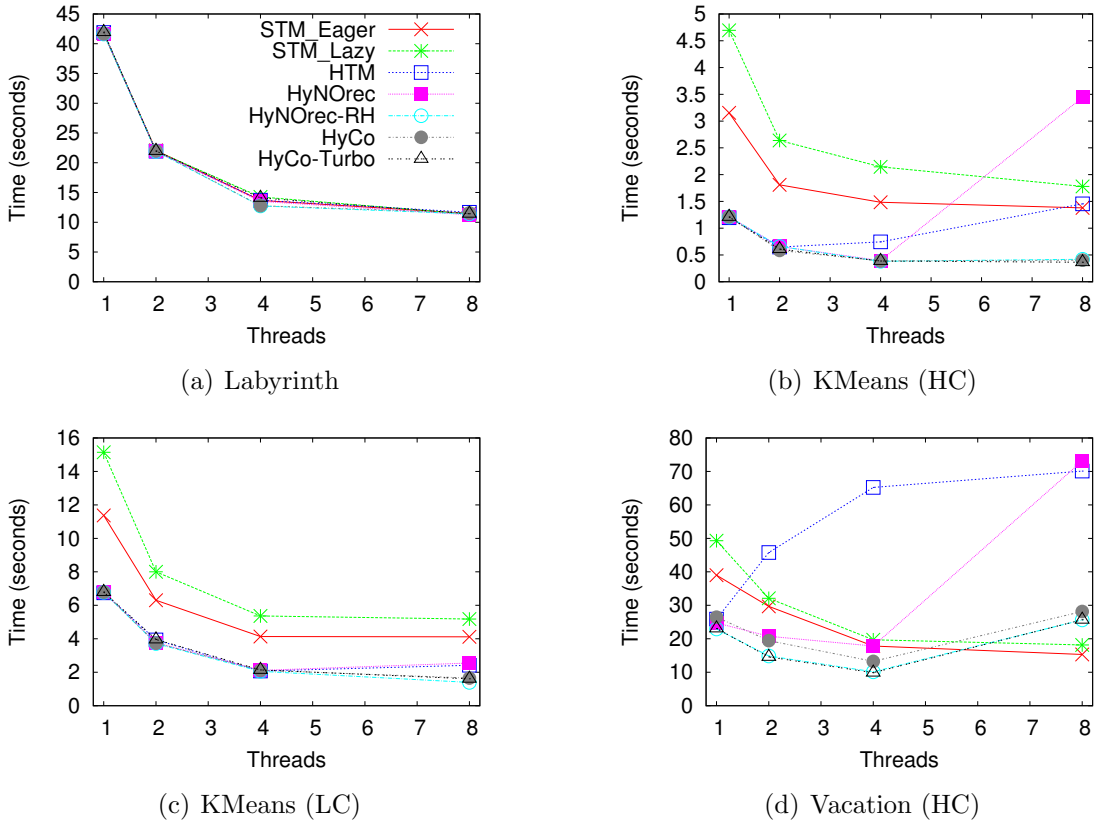
To gain a better understanding of why HyCo scales better than GCC's HTM, we measured the frequency of each type of commit for the HyCo execution of the benchmarks. While the majority of transactions can commit using HTM (NS state), there are nontrivial instances in which transactions fall back to STx mode. While STx transactions are rare, the number of HTx transactions that commit concurrently with STx (i.e., when the STx is in HC mode) is high (indicated by HTx:HC). This confirms that the opportunity cost of serializing is high: in HTM and HTM_20, every fallback to STx becomes a fallback to Serial, and all concurrency among HTx:HC, STx:RO, and STx:HC is lost. This is most unfortunate for read-only STx, which otherwise are concurrent.[1]

5.4.2 STAMP Performance

STAMP performance is shown in Figures 5.3 and 5.4. Unlike the microbenchmark experiments, STAMP performance is shown as total time. The expectation is that more threads will result in a decreased execution time.

As in previous work [69], we observe that the Labyrinth benchmark shows little variation among algorithms. This is a consequence of the benchmark being rewritten

[1]Note that we do not use compiler information to identify read-only transactions; had we used this information, an optimized STx:RO fastpath would be possible. In the tree workloads, many read-only transactions are not statically identifiable (e.g., an insert of a key that already exists), and thus such an optimization would have less value than it might otherwise seem.



(a) Labyrinth

(b) KMeans (HC)

(c) KMeans (LC)

(d) Vacation (HC)

Figure 5.3: STAMP performance(1/2). HC and LC refer to high- and low-contention command-line configurations.

to match the Draft C++ TM Specification: transactions no longer comprise a significant portion of execution time. As has become standard practice, we do not report Bayes performance, since the benchmark exhibits nondeterministic behavior.

Among the remaining 8 benchmark configurations, we see two trends emerge. First, on workloads with high contention, such as KMeans-HC and Vacation-HC, HTM performs best at one thread, but its performance degrades as the thread count increases, due to its reliance on serialization to ensure progress after repeated aborts. In contrast, HyCo manages to maintain its performance as contention increases, by falling back to STx. This trend peters out to some degree at 8 threads for Vacation-HC, due hardware multithreading effects: with four cores and 8 hardware threads, transaction write capacities are effectively halved at 8 threads. The low-contention

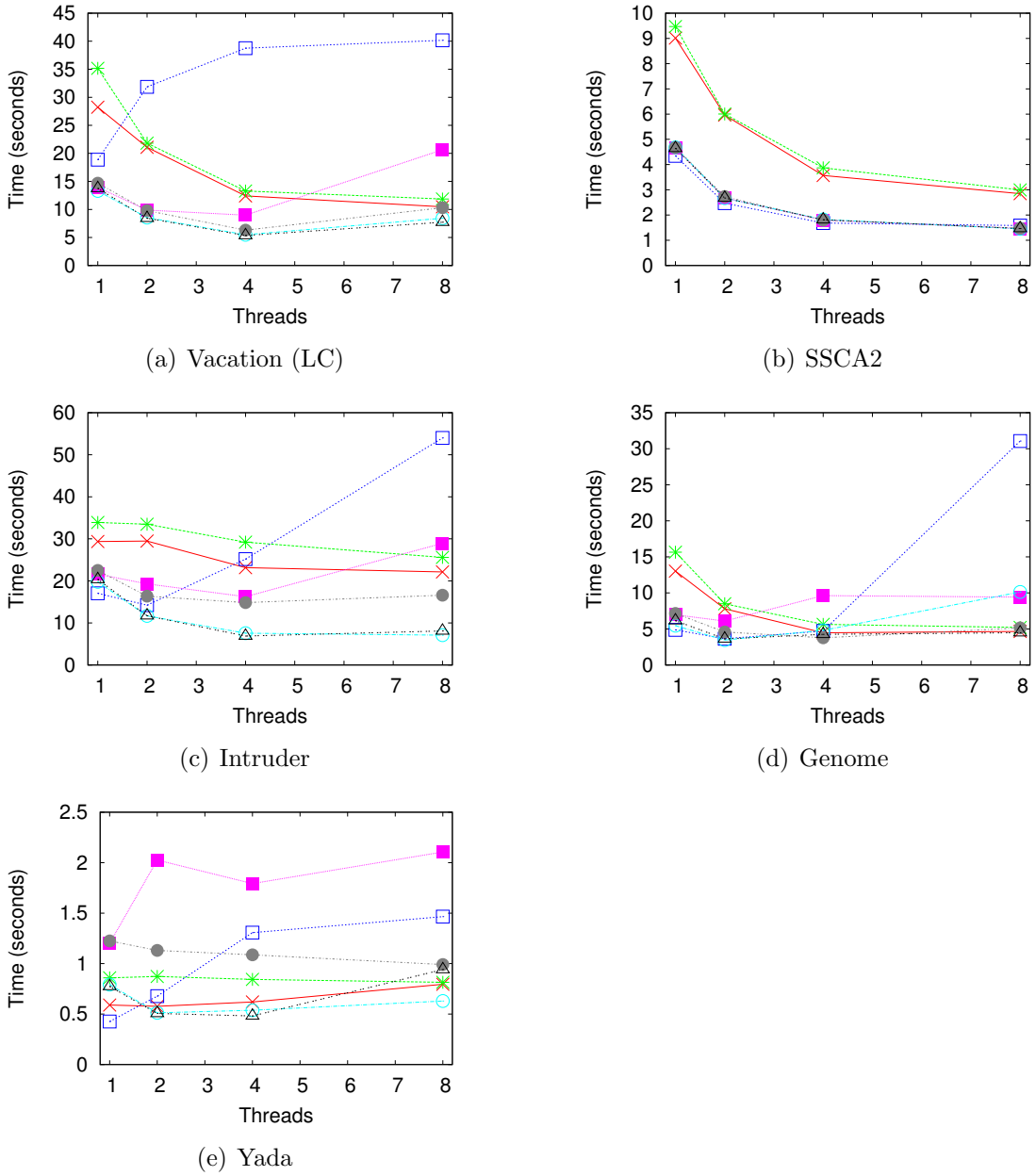


Figure 5.4: STAMP performance (2/2). HC and LC refer to high- and low-contention command-line configurations.

variants of KMeans and Vacation show that as contention decreases, HTM is able to perform on-par with HyCo, but HyCo remains a superior choice overall. The same is true for SSCA2, where small transactions run bottleneck-free in HyCo and HTM.

Also, the contention management can be used to tune HyCo. Figure 5.5 shows

Benchmark	TC	NS	HTx:HC	STx:RO	STx:HC	STx:SC	STx:Turbo	Serial
Vacation(HC)	4	980.9k	858	1	128	792	65.8k	0
Vacation(HC)	8	218.3k	108.6k	763	6.7k	89.1k	100.9k	0
Yada	4	7.1k	18	0	7	19	1.6k	0
Yada	8	182	2.3k	9	687	693	27	2

Benchmark	TC	STx ready to commit but has to abort	STx time spent in TxBegin
Vacation(HC)	4	$\approx 0\%$	3.03%
Vacation(HC)	8	$\approx 0\%$	33.11%
Yada	4	0.06%	2.20%
Yada	8	6.04%	46.14%

Table 5.1: Since performance degrades on Vacation (HC) and Yada at high thread count (8), this table presents the frequency of each type of commit, and analysis of the STx behavior on both high and low thread counts with HyCo-Turbo. "TC" stands for the thread count. Values were reported by a randomly chosen thread.

that by enabling STx:HC mode, with two attempts before fallback to STx:SC mode, 30% improvement can be achieved at high thread counts. This degree of tuning is not available in other Hybrid TM algorithms.

The second trend is shown by Genome, Intruder, and Yada. In these benchmarks, HyCo incurs higher latency than HTM in order to interact with its write set. Recall that for STx transactions, HyCo must perform a lookup on each read, and must buffer its writes in a manner compatible with lookup. This necessitates a more complex data structure (hash of blocks with masks) than the undo log used by eager STM and the HTM fall-back. Consequently, we see that STM_Lazy is a constant factor slower than STM_Eager, and that HyCo similarly incurs high overhead. The problem is most extreme in Yada, where the combination of (a) aborting as HTx before falling back to STx; and (b) incurring write set overhead; results in an insurmountable slowdown at all thread levels. Simi-

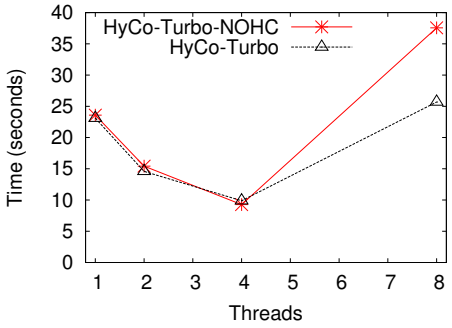


Figure 5.5: HyCO-Turbo with/without STx:HC enabled on STAMP Vacation (High Contention)

larly, in Genome and Intruder, the frequency of lookups creates a significant overhead. Moreover, Since performance degrades on Yada and Vacation(HC), to get a further understanding on the types of HTx commits and STx behavior, we present statistic analysis in Table 5.1. Time spent on spinning in `TxBegin()` significantly increases when thread count is high, which is also one of the factors that hurt the performance.

In effect, these results confirm claims made by Kestor et al. [42]. In their work, they showed that a proper implementation of lazy STM in GCC incurred higher constant overhead than previously believed. While we believe our lazy TM implementations to be more optimal than theirs, the problem remains: the baseline for lazy STM is worse than eager, especially in unmanaged languages.

On this last point, we conducted experiments with two different write set implementations: a hash table and an unbalanced BST. These tests showed that the data structure itself was not the slowdown. Rather, the cost came from manipulating bit masks in order to handle the case where a byte is accessed as part of multiple accesses of varying granularity (e.g., the byte is written, and then the enclosing word is read). These costs are shared by all of our Hybrid TM implementations.

5.4.3 Memcached Performance

Lastly, we evaluate all TM implementations on memcached. We followed the experiment configuration of Ruan et.al [70]. The configuration results in a number of operations proportional to the number of threads: flat curves indicate perfect scaling, higher values represent slowdown. The results are presented in Figure 5.6.

This is the first instance in which HyCo does not match or outperform HyNOrec-RH, and it performs worse only at the highest thread count (8). HyCo suffered from wasted work caused by "ready to commit but has to abort" software transactions, as they could not detect conflicts until commit time. Spinning at the `TxBegin` for STx also contributed to overhead. To confirm and understand more thoroughly of

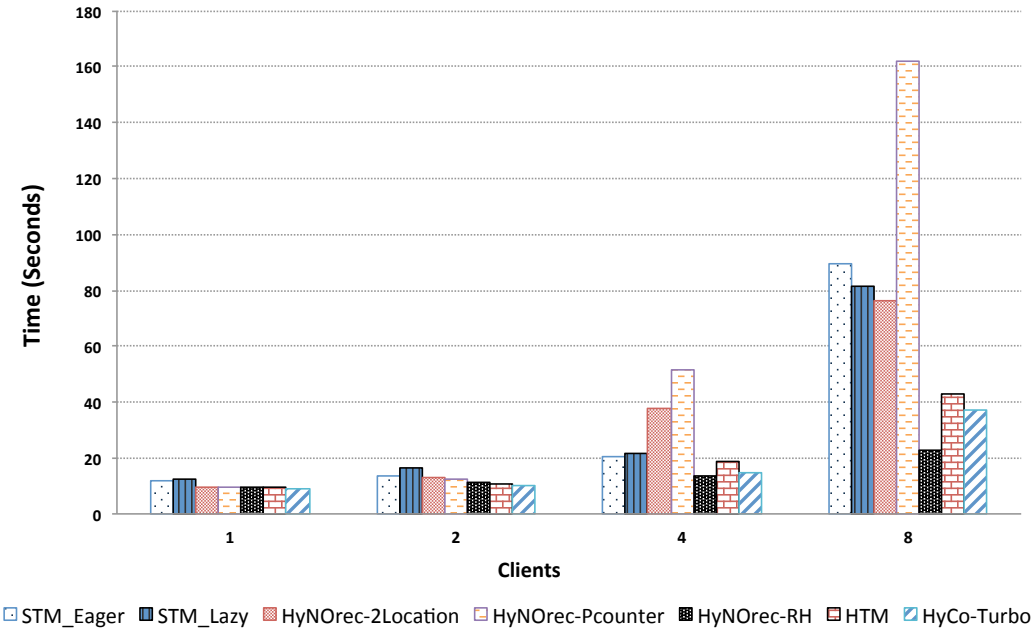


Figure 5.6: Memcached Performance and Analysis

Client Count	NS	HTx:HC	STx:RO	STx:HC	STx:SC	STx:Turbo	Serial
2	7.48M	65	0	85	10	4.4k	0
8	5.44M	544.3K	0	195.7k	72.2k	295.9k	0

Client Count	STx ready to commit but has to abort ratio	STx time spent in TxBegin ratio
2	3.04%	1.55%
8	42.64%	25.97%

Table 5.2: Frequency of each type of commit and analysis of the STx behavior for Memcached on different client counts with HyCo-Turbo. Values were reported by a randomly chosen thread.

such behavior, we did extra analysis, shown in Table 5.2. We present data for a low thread count (client count of 2) and a high thread count (client count of 8). As contention and conflicts are increased, about 43% of the HyCo STx suffer from wasted work undetected until the commit point, and about 26% of the running time is spent on spinning at the transaction begin. Since this slowdown is also coinciding with the point at which the machine begins hyperthreading, and hence sacrificing HTx capacity, the best solution is to exploit HyCo's support of arbitrary contention managers. Good candidates include [7] and [96].

5.5 Summary

In this chapter, we presented the Hybrid Cohorts (HyCo) algorithm. The foundation of HyCo is a state machine that governs when transactions may begin, and when they may attempt to commit. This state machine ensures correctness (opacity) by presenting each transaction with the illusion that memory is immutable during program execution. In HyCo, hardware-mode transactions can run virtually instrumentation-free, all inter-thread coordination is constrained in the transaction begin and commit functions, and even software-mode transactions can use hardware TM support to accelerate their commit operations.

Our evaluation shows that HyCo is competitive with the state of the art. This is particularly useful since HyCo affords more opportunity for contention management than prior Hybrid TM systems. As hardware TM continues to mature, and transaction capacities increase, we believe that support for contention management on the software fallback path will become the most significant distinguishing factor, since the fast path of all known hybrid TMs is roughly the same.

Chapter 6

Conclusion and Future Work

In this dissertation, we first introduced some challenges of concurrent programming and basic concepts of HTM, STM and Hybrid TM. Then we discussed platform related techniques to improve TM performance, distributed throughout four chapters: In Chapter 2, we investigated costs of platform specific instrumentation and provided suggestions and solutions to reduce these costs. A new TM algorithm "Cohorts" was presented to eliminate memory fence costs on processors with relaxed memory consistency. In Chapter 3, we explored a more efficient timestamp implementation that achieves local and global monotonicity via the x86 tick counter. We discussed the possible instruction reordering introduced by accessing cycle counters, and provided a correct solution to apply cycle counter based timestamp to existing TM algorithms. In Chapter 4, we introduced a technique to reduce the transaction abort rate by reordering read-modify-write operations, and therefore improve overall performance. Several algorithms with such technique embedded were presented, as well as a solution to restore language level semantics, which could otherwise be violated by read-modify-write reorderings. In Chapter 5, we presented a new Hybrid TM algorithm "HyCo" that uses "Cohorts" as the fall-back software slow path and Intel Haswell TSX as the fast hardware path. HyCo not only performs well, but provides safety, flexible mode

switching, and an open window for variety of contention management techniques.

These four chapters gradually discussed the research we have conducted on improving the performance of transactional memory by utilizing powerful platform characteristics and avoiding hidden costs that may be introduced by misunderstandings of platform specificity.

As for future work, there are many directions that are worth further investigation. First, the sorts of platform-specific problems demonstrated in previous chapters are not the only such bottlenecks. Some public TM systems have high latency embedded in their implementations. For example, [70] showed that significant overhead was introduced by the readers/writer lock implementation in libitm, a supporting library for TM in the latest versions of GCC. This lock is used to support transaction mode switching. However, platform should be taken into consideration when designing such feature. For example, ARM has a smaller thread count in a foreseeable future, thus there will be a smaller abort rate than on x86/SPARC/POWER, which results in less frequent mode switching, and fewer readers and writers. A fully functional readers/writer lock may be too burdensome in this case. Instead, a simple mutual exclusion lock would suffice, as long as it takes the cost of atomic operations and TLS into consideration. For example, the compare and swap operation is relatively much more expensive on SPARC/ARM than on x86.

Second, timestamps are used in many areas other than transactional memory. A variety of concurrent data structures use timestamps to track version numbers as well. Moreover, these data structures may be widely used in many well known parallel programming models, such as Intel TBB. There are promising opportunities to improve them via tick counters. But extreme caution in applying tick counter timestamps is strongly recommended.

Third, an extension of RMW reordering technique from single-location RMWs to multi-location operations could be beneficial. This effort, which combines elements of

Abstract Nested Transactions [33] and Safe Futures [93], will likely require extensive static analysis to predict the reads of the delayed operation, so that any subsequent read that forces a promotion of the delayed operation can be identified at run time.

Fourth, the HyCo performance leads us to rethink the design of Hybrid TM. Cohorts was not originally supposed to scale well on x86, as it was designed for reducing memory fences by sacrificing a certain degree of concurrency. Surprisingly, despite the complicated state machine, and heavily instrumented STM slow path, HyCo managed to compete with the most recent Reduced Hardware Hybrid NOrec, which is currently the fastest hybrid TM. This demonstrates that the essence of a fast Hybrid TM is probably not the scalability or latency of its software path at all, but the ability of a STM to stop starvation, to ensure fairness, and to avoid pathological behaviors which are introduced by HTM. Also the implementation of the state machine tells us that the traditional counter-only or local-status-only techniques may not be the solution for a low latency state machine. A good implementation in concurrent programming is not a trivial work, and must consider platform specificity. A direct example is that Intel Haswell processor has a hardware adjacent cache line prefetch mechanism, that automatically fetches an extra 64-byte cache line. When enabling HTM, this could potentially lead to higher possibility of false sharing, thus increase the unnecessary HTM aborts. Extra padding for shared variables must be carefully placed.

We expect these opportunities, and many others, to lead to further improvement of TM performance. The key enabling factor, introduced by this dissertation, is that low-level platform issues have a significant impact on high-level decisions about how to implement a TM system. Platform-aware techniques appear to be essential to high performance in transactional memory and other tightly coupled shared memory programming models.

Bibliography

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[3] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[4] 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.

[5] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory Models for Open-Nested Transactions. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, San Jose, CA, Oct. 2006.

[6] 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.

[7] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, Denver, CO, Aug. 2006.

[8] H. Attiya and A. Milani. Transactional Scheduling for Read-Dominated Workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, Dec. 2009.

[9] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[11] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, AB, Canada, Aug. 2014.

[12] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[13] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Pro-*

ceedings of the 27th ACM Conference on Programming Language Design and Implementation, June 2006.

[14] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, and S. Yip. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, March–April 2009.

[15] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[16] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Grossman, and D. Christie. ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, Dec. 2010.

[17] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[18] 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.

[19] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[20] D. Dice, T. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of Lazy Subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory*, Paris, France, July 2014.

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

[22] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[23] 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.

[24] A. Dragojevic, R. Guerraoui, A. Singh, and V. Singh. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, Calgary, AB, Canada, Aug. 2008.

[25] 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.

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

[27] J. Gottschlich, M. Vachharajani, and J. Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, Toronto, ON, Canada, Apr. 2010.

[28] R. Guerraoui, T. Henzinger, M. Kapalka, and V. Singh. Transactions in the Jungle. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[29] 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.

[30] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

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

[32] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[33] T. Harris and S. Stipic. Abstract Nested Transactions. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[34] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[35] 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.

[36] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[37] IBM. Concurrent Building Blocks, 2009–2010. http://amino-cbbs.sourceforge.net/.

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

[39] Intel Corp. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-012a edition, Feb. 2012.

[40] Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.

[41] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[42] G. Kestor, L. Dalessandro, A. Cristal, M. Scott, and O. Unsal. Interchangable Back Ends for STM Compilers. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[43] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[44] Y. Lev and J.-W. Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[45] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[46] P. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, Cancun, Mexico, Apr. 1994.

[47] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, Long Beach, CA, Jan. 2005.

[48] 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.

[49] 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.

[50] V. J. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. 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*, Ottawa, ON, Canada, June 2006.

[51] 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.

[52] J.-P. Martin, C. Rossbach, D. Murray, and M. Isard. Supporting Efficient Aggregation in a Task-based STM. In *Proceedings of the 3rd Workshop on Systems for Future Multicore Architectures*, Prague, Czech Republic, Apr. 2013.

[53] A. Matveev and N. Shavit. Reduced Hardware NOREC: An Opaque Obstruction-Free and Privatizing HyTM. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[54] A. Matveev and N. Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, Mar. 2015.

[55] 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.

[56] 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.

[57] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[58] M. Moir. Unpublished Manuscript, July 2005.

[59] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings*

of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct. 2006.

[60] E. Moss and A. L. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005.

[61] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[62] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[63] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[64] M. Pohlack and S. Diestelhorst. From Lightweight Hardware Transactional Memory to Lightweight Lock Elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[65] 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.

[66] 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.

[67] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[68] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[69] W. Ruan, Y. Liu, and M. Spear. STAMP Need Not Be Considered Harmful. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[70] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[71] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[72] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, Dec. 2006.

[73] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[74] 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.

[75] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[76] 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 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[77] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th International Symposium on Computer Architecture*, Beijing, China, June 2008.

[78] A. Shriraman, M. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[79] N. Sonmez, T. Harris, A. Cristal, O. S. Unsal, and M. Valero. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] M. Spear, M. M. Michael, M. L. Scott, and P. Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, Seattle, WA, Mar. 2009.

[85] M. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[86] M. Spear, A. Shriraman, L. Dalessandro, and M. Scott. Transactional Mutex Locks. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[87] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[88] J.-T. Wamhoff, P. Felber, C. Fetzer, G. Muller, and E. Riviere. FastLane: Streamlining Transactions for Low Thread Counts. In *Proceedings of the 7th ACM SIGPLAN Workshop on Transactional Computing*, New Orleans, LA, Feb. 2012.

[89] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Riviere, and G. Muller. PathTM: Improving Performance of Software Transactional Memory for Low Thread Counts. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, Feb. 2013.

[90] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[91] 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.

[92] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear. A Transactional Memory with Automatic Performance Tuning. In *Proceedings of the 7th International Conference on High-Performance and Embedded Architectures and Compilers*, Paris, France, Jan. 2012.

[93] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, Oct. 2005.

[94] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[95] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.

[96] R. Yoo and H.-H. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[97] R. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[98] 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.

[99] F. Zyulkyarov, S. Stipic, T. Harris, O. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, Vienna, Austria, Sept. 2010.

Curriculum Vitae

Name	Wenjia Ruan
Date of Birth	March 1987
Web	http://www.cse.lehigh.edu/~wer210

Education

August 2015	**Ph.D in Computer Science**
Bethlehem, PA USA	*Lehigh University*
September 2013	**M.S in Computer Science**
Bethlehem, PA USA	*Lehigh University*
June 2009	**B.S in Electronic Engineering**
Qingdao, China	*Ocean University of China*

Professional Experiences

June 2011 – June 2015	**Research Assistant**
Bethlehem, PA USA	*Lehigh University*
May – August 2014	**Summer Research Intern**
Santa Clara, CA USA	*Qualcomm Research*
September 2010 – May 2011	**Teaching Assistant**
Bethlehem, PA USA	*Lehigh University*

Selected Publications

"On the Platform Specificity of STM Instrumentation Mechanisms"

Shenzhen, China **CGO 2013**

"Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters"

Volume 10 Issue 4 **ACM TACO 2013**

"Transactional Read-Modify-Write Without Aborts"

Volume 11 Issue 4 **ACM TACO 2015**

"Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached"

Salt Lake City, UT USA **ASPLOS 2014**