On the Platform Specificity of STM Instrumentation Mechanisms

Wenjia Ruan Yujie Liu Chao Wang Michael Spear
Lehigh University
{wer210, yul510, chw412, spear}@cse.lehigh.edu

Abstract

Supporting atomic blocks (e.g., Transactional Memory (TM)) can have far-reaching effects on language design and implementation. While much is known about the language-level semantics of TM and the performance of algorithms for implementing TM, little is known about how platform characteristics affect the manner in which a compiler should instrument code to achieve efficient transactional behavior.

We explore the interaction between compiler instrumentation and the performance of transactions. Through evaluation on ARM/Android, SPARC/Solaris, IA32/Linux, and IA32/MacOS, we show that the compiler must consider the platform when determining which analyses, transformations, and optimizations to perform. Implementation issues include how TM library code is reached, how per-thread TM metadata is stored and accessed, and how a library switches between modes of operation. We also show that different platforms favor different TM algorithms, through the introduction of a new TM algorithm for the ARM processor. Our findings will affect compiler and TM library designers: to achieve peak performance for transactions, the compiler must perform platform-dependent analysis, transformation, and optimization, and the interface to the TM library must differ according to platform.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Design, Performance

Keywords Transactional Memory, ARM, relaxed memory consistency, thread-local storage

1. Introduction

After two decades of effort, hardware support for Transactional Memory has finally arrived in mainstream processors, in the form of extensions in the Intel Haswell microarchitecture [11]. 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 [2].

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.

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 [23] 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/Linux. As appropriate, we also consider differences between Linux and Mac OSX. Our results include the following:

- TLS introduces a significant overhead on all but IA32/Linux; on other platforms the transactional compiler should add manual management of per-thread metadata.
- Fine-grained support for mode-switching within the TM library causes a noticeable slowdown on ARM, necessitating less flexible mechanisms.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE...$15.00

- 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, we discuss platform-specific thread-local storage (TLS) costs and alternatives. Section 3 focuses on how instrumentation is reached, and shows that different platforms sit at different points on the flexibility/overhead spectrum. Section 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 5 concludes.

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.

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 latest version of the open-source RSTM library [25] 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 [8] for IA32, TL2 [7] on SPARC, and NOrec [4] on ARM.

- 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.
- SPARC: 1.165 GHz Sun SPARC Niagara T2000 with 8 cores/64 threads, 32GB RAM, Solaris 10, GCC 4.7.1.
- ARM: NVIDIA 1.4GHz Tegra3 with 4 cores, 1GB RAM, Android Linux 4.0.3, GCC 4.4.1.
- IA32/MacOS: 2.66GHz Intel Core i7 M620 with 4 cores, 4GB RAM, MacOS 10.7.2, GCC 4.2.1.

2.1 Results

The results of our stress test appear in Figure 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.

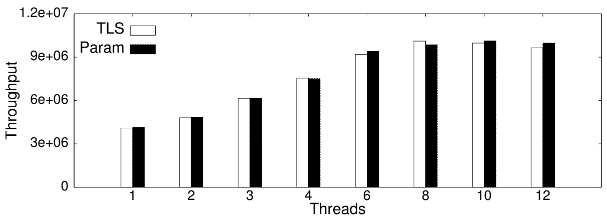
The remaining differences stem from the platform-specific TLS ABI. On IA32/Linux, the thread local data region is at a constant offset in the binary. Thus, a TLS access can be translated to a simple register-to-register move, in which the TLS address is computed directly from the segment register (%gs).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

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

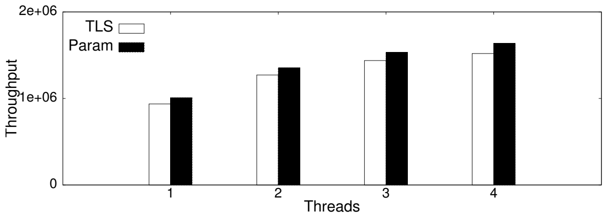
On SPARC, accessing TLS involves more instructions than on IA32. The initial assembly code consists of 12 instructions, but during linking several constants are known, resulting in a three-instruction sequence:

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

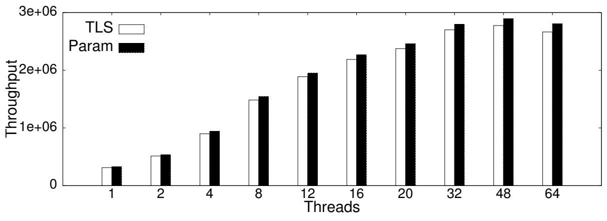
The first two instructions load a 32-bit constant into %g1. The final instruction is specific to the TLS model on SPARC, and loads the TLS offset. Since the Niagara2 pipeline is



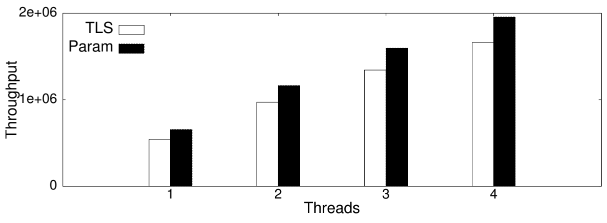
(a) Platform: IA32(12-way)/Linux, Algorithm: TinySTM



(b) Platform: IA32(4-way)/MacOS, Algorithm: TinySTM



(c) Platform: SPARC(64-way)/Solaris, Algorithm: TL2



(d) Platform: ARM(4-way)/Linux, Algorithm: NOrec

Figure 1: The cost of thread-local storage vs. additional function parameters

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.

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 [12, 17, 18, 25]. However, the requirement is not merely one needed for switching among algorithms. At least a limited form of adaptivity appears to be fundamental.

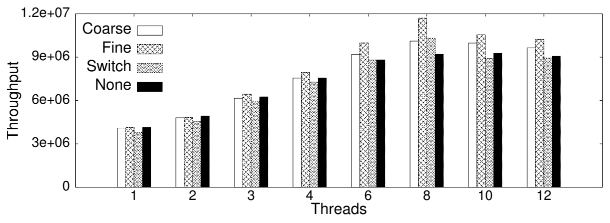
Consider the draft C++ TM specification [2], which includes *relaxed* transactions [23]. A relaxed transaction is expected to transition between a concurrent mode and a serial mode if the transaction attempts an operation that cannot be rolled back.[1] While this support may be provided by a branch on 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 [3, 12], at which point full adaptivity support [18, 28] 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 [18]. 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 [17, 25]. 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 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.

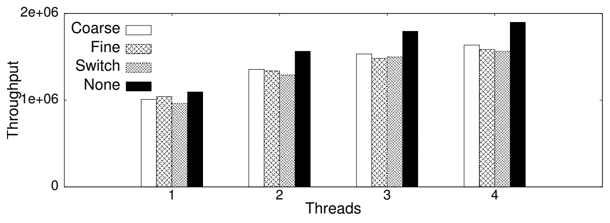
In Figure 2, we contrast the performance of these options by repeating the red-black tree microbenchmark from Section 2. As before, a high incidence of transactions amplifies the impact of differences in the instrumentation mechanisms. We compare four manners in which transactional instrumentation can be achieved:

- Fine: This curve corresponds to a configuration with per-thread function pointers [17, 25]. "Fine" affords maximum flexibility, since individual transactions can vary their behavior without branching. However, it incurs TLS overhead to locate function pointers.

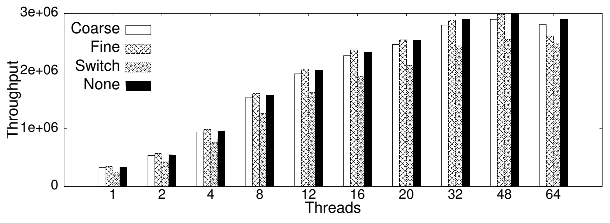
[1] A simple understanding of relaxed transactions is that they allow the programmer to specify that the transaction cannot self-abort, so that irrevocable operations [29] can then be permitted.



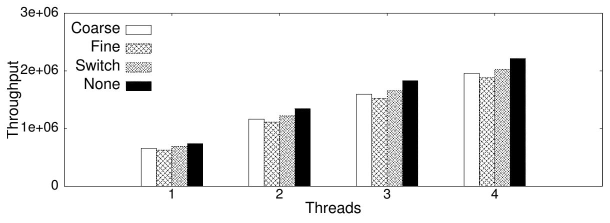
(a) Platform: IA32(12-way)/Linux, Algorithm: TinySTM



(b) Platform: IA32(4-way)/MacOS, Algorithm: TinySTM



(c) Platform: SPARC(64-way)/Solaris, Algorithm: TL2



(d) Platform: ARM(4-way)/Linux, Algorithm: NOrec

Figure 2: The cost of mechanisms for reaching (adaptive) instrumentation

- 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.
- 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.
- 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.

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 [25]). 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 availability of predicated instructions on ARM makes the switch statement relatively more efficient than an indirect branch.

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 [7, 10], various IA32 platforms [6, 8, 17, 20, 25, 27], IBM POWER [26], and SPARC Niagara [6]. A few algorithms employ custom OS features [1, 7], and several rely on specific architectural properties (such as low-overhead memory fences in the TLRW byte-lock mechanism [6], an atomic-or primitive in the Unified STM algorithm [17], or more generally the fact that compare-and-swap (CAS) is cheap on modern IA32 chips [25]). Some trade high latency for fewer bottlenecks and improved scalability. Others are best when the core count is low.

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 [4].
LSA	The TinySTM write-through algorithm. This algorithm acquires locks eagerly, modifies memory locations before commit time, and maintains undo logs [8]. LSA uses a table of versioned locks ("orecs") to detect conflicts among transactions.
TL2	TL2 uses commit-time locking and redo logs, and otherwise closely resembles LSA [7].
OrecELA	This algorithm expands upon LSA and TL2 to offer stronger language-level semantics ("ELA" semantics in the taxonomy of Menon et al. [14]). 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 [4].
TLRW	TLRW uses an array of reader/writer bytelocks. Transactions lock every location they read, but do not validate reads at commit time [6].

Table 1: Representative STM algorithms.

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 algorithms, such as the original McRT STM [20], still (a) some algorithms with commit-time locking can exploit relaxed consistency checks in `TxRead` [26], 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.

To illustrate the benefit of a wide interface, consider the case of a write after read in the Unified STM system [17]: 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 [6, 8, 9, 17], 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 [26], which have low value for early locking STM algorithms.

To assess whether either set of optimizations is universal, Figure 3 presents an evaluation on ARM, SPARC, and IA32 using the STAMP benchmark suite [16]. We consider the algorithms listed in Table 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 low. Since this lone exception always performed poorly, its higher variance does not affect our conclusions.

4.1.1 Summary of Results

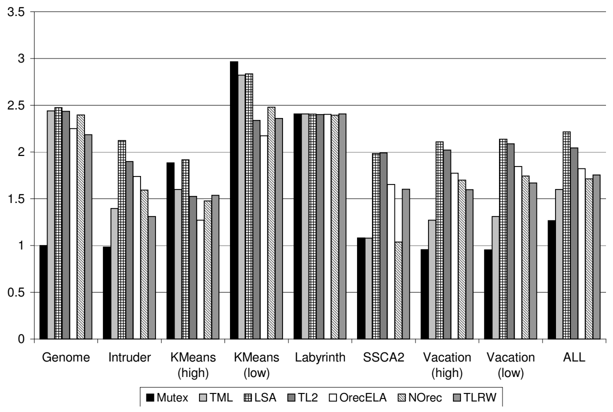
There are two dimensions on which algorithms can be partitioned. First, the LSA, TML, and TLRW algorithms are the only ones in which locks are acquired before commit time. Thus they are the only algorithms that could potentially benefit from Ni's optimizations [17]. Second, TML, OrecELA, and NOrec are 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 [14]. 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 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.

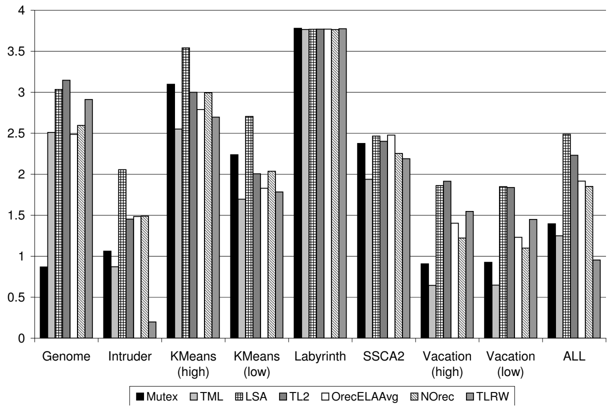
We did not measure either of the extended interfaces to the TM library discussed earlier. Ni's optimizations [17] would offer an improvement to LSA, TML and TLRW, while Spear's optimizations [26] would benefit NOrec, OrecELA, TML, and TL2. Since no single algorithm stands out on all platforms, the most 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).

4.2 Tackling the Cost of Fences

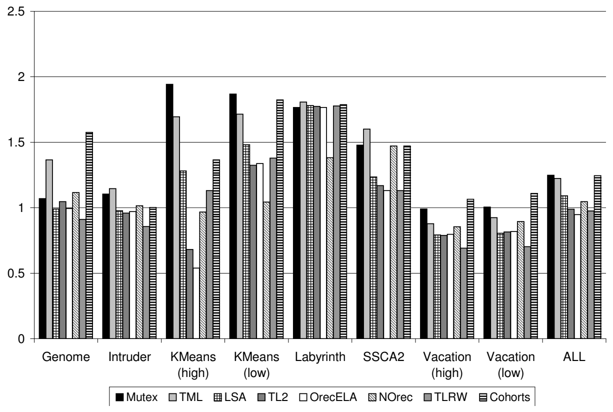
On ARM, none of the algorithms from Table 1 consistently outperforms Mutex. The latency of individual transactions is simply too high, in large part due to memory fences.



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



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



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

Figure 3: STAMP speedups vs. single-threaded Mutex

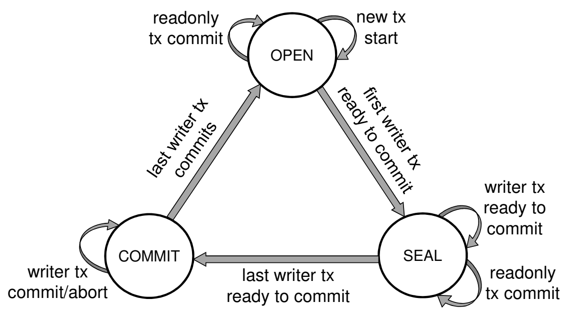


Figure 4: State Transitions of a Cohort

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 to 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 but a constant number of fences at transaction boundaries.

Cohorts outperforms all other algorithms on ARM (Figure 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.

4.2.1 The Cohorts Algorithm

Cohorts employs a state machine to control which transactions can commit, and when (Figure 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.

Algorithms 1 and 2 present the Cohorts algorithm. The state machine is implemented by two fields: a per-transaction variable $status$ (`START` indicates the transaction has entered the cohort but has not yet reached the commit point,

Algorithm 1: Cohorts Algorithm

```
CONSTANT: START, FINISH
GLOBAL VARIABLES:
  $tail$            : Boolean*    // initially nil
  $transactions$    : Tx[]        // array of transactions
PER-TRANSACTION VARIABLES:
  $spin$     : Boolean    // entry in commit queue
  $status$   : Integer    // transaction status indicator
  $turbo$    : Boolean    // go turbo indicator
  $writes$   : WriteSet   // write set
  $reads$    : ReadSet    // read set

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

TxRead($addr$)
8      if $turbo$ then
9          WriteBack (); return $*addr$
10     if $addr \in writes$ then return $writes[addr]$
11     $v \leftarrow *addr$
12     $reads \leftarrow reads \cup \{\langle addr, v\rangle\}$
13     return $v$

TxWrite($addr, v$)
14     if $turbo$ then
15         WriteBack (); $*addr \leftarrow v$
16     else $writes \leftarrow writes \cup \{\langle addr, v\rangle\}$
```

Algorithm 2: Cohorts Algorithm continued.

```
TxCommit()
       // commit a read-only transaction or a (notified) turbo
          transaction that has performed write back
17     if $writes = \emptyset$ then
18         $status \leftarrow$ FINISH
19         return
       // commit a read-write transaction: enqueue and wait
20     $pred \leftarrow$ AtomicSwap($tail, \&spin$)
21     $status \leftarrow$ FINISH; MemoryBarrier
       // notify the last transaction to go turbo
22     if TxLeft() $= 1$ then
23         for each $tx$ in $transactions$ do
24             $tx.turbo \leftarrow$ true
25     if $pred = $ nil then
           // first writer waits until everyone reaches commit
26         while TxLeft() $\neq 0$ do wait
27     else
           // otherwise wait until signaled
28         while $(*pred)$ do wait
29     if $\neg$Validate() then
30         FinishCommit()
31         abort
32     WriteBack()
33     FinishCommit()

WriteBack()
       // do write back if buffer is not empty
34     if $writes \neq \emptyset$ then
35         for each $\langle addr, v\rangle$ in $writes$ do
36             $*addr \leftarrow v$
37         $writes \leftarrow \emptyset$

Validate()
       // value-based validation
38     for each $\langle addr, v\rangle$ in $reads$ do
39         if $*addr \neq v$ then return false
40     return true

FinishCommit()
41     $spin \leftarrow$ false
42     if $tail = \&spin$ then $tail \leftarrow$ nil

TxLeft()
       // number of tx left in a cohort
43     $counter \leftarrow 0$
44     for each $tx$ in $transactions$ do
45         if $tx.status = $ START then $counter$++
46     return $counter$
```

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 [13]) 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 [4], commit-time validation does not check a lock table, but instead compares actual values in memory. This leads to trivial read (lines 10 to 13) and write (line 16) instrumentation and also obviates memory fences while reading and writing.

At the point of transition from SEAL to COMMIT, any arbitrary contention management policy [22] 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 paper, 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.

4.2.2 Compiler Optimizations

In Figure 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.

4.2.3 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 and game consoles. Whether it remains an important market or not is uncertain, particularly if these platforms add hardware TM support.

4.3 The Impact of Hardware Assisted STM Libraries

Early work on hardware accelerated STM [15, 21, 24] 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 [3, 5, 12, 19], 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 [18, 29]. In this mode, loads and stores are not instrumented. Multiple cloning also occurs in Christie's TM stack [3] 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. The GNU C++ compiler currently does not generate multiple transactional clones of a function. Thus, for example, a relaxed transaction that requires irrevocability still performs 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 5 presents the performance of the RBTree microbenchmark from Section 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:

```
$T$ TxRead ($T *\ addr$)
    return $*addr$;

$void$ TxWrite ($T *\ addr,\ T\ val$)
    $*addr \leftarrow val$;
```

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:

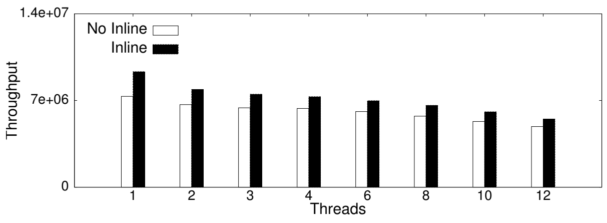
- On IA32, the fact that register `eax` is both the parameter and return value of the read function impedes instruction scheduling, especially when multiple reads are performed in succession. For "Inline", the operands to successive loads can be kept in different registers.
- On SPARC, function calls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

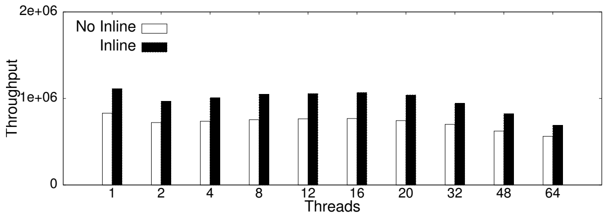
5. Conclusions

In this paper, 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.

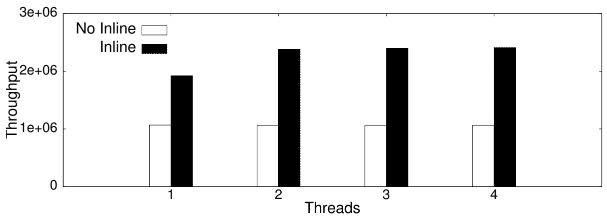
These themes are not specific to TM. As parallelism becomes more pervasive, and as architectural diversity continues to increase, more libraries will become adaptive and self-tuning. This, in turn, will increase dependence on TLS and adaptive instrumentation, which we have shown to have platform-dependent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



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



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



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

Figure 5: Inlined and non-inlined versions of the Mutex algorithm on different platforms. Differences between IA32/Linux and IA32/MacOS are negligible.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.