Mutex Locking versus Hardware Transactional Memory: An Experimental Evaluation

Sean R. Moore

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Roberto Palmieri
Dongyoon Lee

September 16, 2015
Blacksburg, Virginia

Keywords: Hardware Transactional Memory, Global Lock, GNU C Library

Mutex Locking versus Hardware Transactional Memory: An Experimental Evaluation

Sean R. Moore

(ABSTRACT)

It has historically been the case that CPUs have run programs ever faster without significant intervention on the behalf of the programmer. However, this "free lunch" has largely ended due to the end of exponentially increasing core frequency and the current slow increase in instruction-level parallelism but continues to a degree in cache size improvements. But since Moore's law still largely continues "lunch", i.e. program performance, can still be bought at the price of rewriting code for multiple cores, which is enabled by the trend Moore's law describes. Multicore architectures cannot aid performance for problems whose solutions are necessarily sequential in nature and writing efficient and correct concurrent programs is not easy in all cases when using synchronization methods like fine-grained mutex locks.

Transactional memory, and its implementation as hardware transactional memory, allow programmers to write concurrent applications without the attendant complexity of programming with mutex locks. This allows programmers to focus on optimizing the application for performance. Given that transactions can run two segments of code in parallel that a mutex lock would force to run sequentially and that transactions can abort, causing a program to do the same work more than once, whether transactions perform better or worse than mutex locks is dependent on the program's execution profile and the coarseness or fineness at which mutex locks are used.

In this thesis the GNU C Library's futex implementation of mutex locks and Intel's Restricted Transactional Memory have been compared and the behavior of those transactions has been analyzed. This analysis includes a pathological behavior permitted by the GNU C Library's hardware transactional memory implementation of mutex locks. The tradeoffs between fine-grained and global locking implementations have been discussed, compared, and used in the context of fallback locks for hardware transactions. This thesis provides evidence to the effect that fine-grained locking is not critical for program performance and that in many cases global locking and hardware transactions can provide nearly equivalent performance without the programming difficulties. This work has shown that across the 23 applications examined, with relation to their original locking implementation, a global locking scheme without elision has a 0.96× speedup, Intel's Restricted Transactional Memory (RTM) with the application's original locks as a fallback has a 1.01× speedup and with global lock fallback RTM has a speedup of 0.97×.

This work is supported in part by NAVSEA/NEEC under grant 3003279297. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of NAVSEA.

Acknowledgments

I want to thank

Dr. Binoy Ravindran for advising me during my time as a graduate student at Virginia Tech and chairing my advisory committee,

Dr. Dongyoon Lee for serving on my advisory committee,

Dr. Roberto Palmieri for serving on my advisory committee and providing excellent guidance,

Dr. Mohamed Mohamedin and Dr. Ahmed Hassan for getting me up to speed on relevant work in hardware transactional memory,

Duane Niles Jr. for discussing with me his share of interesting dilemmas from the software side of transactional memory,

the Systems Software Research Group members and alumni that I have had the pleasure of meeting,

and lastly my friends and family for enabling me to pursue this degree and for giving me innumerable opportunities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multicore is the present and the foreseeable future of CPUs ever since hardware advances which sped up execution of serial applications have largely come to a stop [1]. The trend which Moore's law describes has allowed CPU designers to increase the number of cores that can be crammed onto a chip while other trends helping serial execution speed have come to a halt or at least slowed down [1, 2]. Programmers are now faced with the reality that getting a program to take full advantage of improvements in hardware requires thinking about their program as a parallel one.

When programmers plan or write concurrent code, whether from scratch or for maintenance, they have to understand and deal with the difficulties associated with the current approaches of programming concurrent and parallel applications. For instance, Java had to fix the memory semantics it had previously employed [3] and C++ had postponed defining a memory model leaving projects like the Linux kernel to assume certain memory semantics where none were specified [4]. In addition to the ambiguity and errors introduced and left by the standards, concurrent processing is prone to hard-to-debug programming errors such as data races which are hard to replicate as they are triggered pseudorandomly and their behavior is left undefined by languages like Java and C++ [3, 4].

Programmers writing concurrent programs also have to worry about possible side-effects of their synchronization methods. Deadlocks are one of these side-effects and occur when a program causes one or more streams of execution to wait on some condition to become true and each stream in the deadlock eventually requires itself to progress before being allowed to progress. Another such side-effect is livelocking. Livelock requires that a stream of execution is created with the assumption that it may read or write data which other streams of execution may also write. Those streams of execution are also responsible for redoing (and possibly undoing) work upon interference from another stream of execution or reaching some undesirable state (e.g. deadlock). Livelock occurs when each stream in the livelock indefinitely performs work and rolls that work back without ever actually completing the task.

Ensuring that a program is free from these and other deleterious effects is only complicated by maintaining the program over its life-cycle. The original programming effort to architect the original program to prevent these effects can be cross-cutting, making modifications of the original program difficult.

How is a programmer to take advantage of the performance benefits of hardware parallelism while avoiding the complications that come with concurrency?

Mutex locks, while a useful abstraction, have their drawbacks. Mutexes must, in many cases, follow a prescribed locking order throughout the entirety of a program to ensure that deadlocks cannot occur. This can be hard to enforce and even harder to maintain if complex, application-level procedures acquire locks that the stream of execution holds after returning from those procedures and even worse if that lock is determined dynamically. However, this lock-ordering requirement can be bypassed if the programmer is clever enough to realize that trylocking and rolling back work can violate a locking order yet still be free of deadlocks or data races.

Let us say for example that some programmer wants to perform a macro-operation on a dataset which accepts three records in any order each with their own lock which operates on the first item, the second then the third, carrying data between records with the entire macro-operation needing to appear atomic. If the sub-operations take a lengthy period of time then the programmer would either be forced to acquire locks before they are needed in order to satisfy a lock-ordering requirement or could "cheat" the lock ordering by allowing some permutations to attempt to acquire locks out of order but roll back work if an attempted acquire fails to immediately gain the lock (for example, using trylock). This brings with it, if not planned carefully, the possibility of a livelock with some of the streams of execution continually causing each other to roll back indefinitely.

While not necessarily the hardest conceptual issue to solve, mutex locks also require that operations that appear atomic be wrapped in an acquire and release pair. If a programmer uses a library with thread-safe data structures the programmer can (probably) be assured that any single function call to the library caused some effect or returned some result that is sensible to a programmer familiar with sequential programs. However, if the programmer puts two function calls back-to-back that they expected to occur together they might not get the expected result because the library did not know that those operations should be executed together atomically. If a region of memory accessible to the application is made accessible to the library as well (or vice-versa) it may be unclear how to protect this segment of memory with mutexes.

Can *Transactional Memory* (TM) help improve the situation? TM, abstractly speaking, is a programming interface intended to allow programmers to easily write lock-free programs (in the sense that there is no mutual exclusion) [5]. TM ensures *atomicity* with *begin* and

commit semantics where mutex locks ensure *mutual exclusion* using *lock* and *unlock* semantics. Mutual exclusion ensures that in any group of streams of execution at most one stream "holds" the mutex, i.e. it issued a lock call without yet having issued an unlock call. Any stream that wants to lock an already acquired lock must wait until the stream which owns it releases it. TM attempts to avoid this waiting using *transactions*.

Transactions start with a "begin" which marks the beginning of enforcement of *isolation* requirements, preventing other streams of execution updating that transaction's data in an unexpected way. The "begin" also marks a point to which a transaction can undo its work if it is *aborted*. Aborts can occur because the TM implementation could not guarantee that a transaction would be sufficiently isolated from streams of execution other than the one on which it executes, or because it could not satisfy the all-or-nothing requirement for that transaction. It may be possible for two critical sections which run serially under mutual exclusion to run in parallel with transactions but it runs the risk of taking longer, due to aborts. Each successful transaction completes with a "commit" which ensures that its updates are visible to all other streams of execution.

A draft has been proposed that would, if ratified, make TM part of the C++ standard [6]. The *GNU Compiler Collection* (GCC) has already begun including this proposed functionality which has been used and evaluated in previous studies [7, 8]. However, the C++ draft specification syntax suggests (if not outright requires) that a transaction begin and end on the same syntactic expression (e.g. a statement, expression, basic block, etc.) [6]. While it does not necessarily forbid practical programming patterns it does introduce a dilemma. One can apply an atomic section over entire functions calls that need only the beginning (or end) to be atomic with preceding (or succeeding) operations. Or one can violate the principles of *encapsulation* and *composability* to get better performance.

Given that TM is gaining mainstream support and that issues like deadlock and livelock are largely relegated to the TM implementation it appears that this method of synchronization will have longevity. Since TM has varying types of implementations, each with different performance considerations, some of them have been grouped together and are discussed below.

One grouping of TM implementation is *Software Transactional Memory* (STM). STM is non-blocking [9] and its contention management, i.e. atomicity/isolation violation detection, is done away from the programmer. Instead of needing to work out some synchronization scheme that avoids side-effects which break the software, the programmer can just mark which sections of code are executed with the appearance of atomicity. Once such sections are marked, the library which the implementation uses or code inserted by the compiler handles this synchronization. However, it currently has drawbacks that limit its usability and act as hurdles to widespread adoption as a software engineering tool.

The GCC implementation of C++'s STM proposal (plus a few extensions), in many cases, instruments memory accesses for the transactional versions of code segments, this increases the time taken to execute an atomic section [7]. Because of this instrumentation when a function is usable within a transaction both an instrumented version (usable within a transaction) and a non-instrumented version (fast but unsafe within a transaction) may be generated by the compiler [7]. However, not all function definitions, including common library functions, may be capable of generating a transaction-safe version or a function may indeed be provably transaction-safe but the compiler cannot recognize this fact [7]. This has lead to functions as conceptually simple as determining a C-string's length needing to be re-written to prevent unnecessary serialization of the active transaction in prior work [7].

For these reasons STM is close to but not ready to be a mainstream engineering tool. This is in addition to the overhead in performance which is necessarily incurred from memory access instrumentation.

This brings us to *Hardware Transactional Memory* (HTM). The concept of HTM was originally just TM when proposed by Herlihy *et al.* [5] ("TM" is used to refer to either HTM, STM or both in this work) and recently got a mainstream, commodity implementation in Intel's Haswell processors with *Restricted Transactional Memory* (RTM) [10]. RTM and *Hardware Lock Elision* (HLE) make up Intel's *Transactional Synchronous Extensions* (TSX) [11]. RTM has some shortcomings as well. Even segments of code which are known to otherwise take finite periods of time and access only private memory cannot be guaranteed to complete and must have a fallback path to guarantee progress [11]. This fallback path can take the form of any synchronization method which interoperates with or prevents/terminates operation of potential, parallel hardware transactions. In addition to this, most implementations of HTM also have *responder loses* conflict management, i.e. a core which writes a memory location accessed by a transaction on another core aborts the transaction (whether or not the core which did the writing was in transactional mode). This type of conflict management is more prone to a finite form of livelock than *responder wins* [12].

However, HTM (and as a consequence RTM) has upsides which make it a useful engineering tool. When using a global recursive mutex lock as a fallback for HTM a deadlock cannot be created without writing code which explicitly waits for another stream of execution to change some memory value. Also, while HTM on its own cannot make any progress guarantees, when combined with properly employed mutex locks progress can be ensured, including with a very coarse global recursive mutex. HTM with a global lock is arguably fairly easy to maintain and support for these reasons. Code which uses fine-grained mutex locks needs to make clear to the maintainer which locks are held at a given state of the program, which locks cannot be held during a procedure call, which locks cannot be acquired while the program is in a particular state and what data that lock protects. Of those concerns, a maintainer of a program written with global lock fallback HTM only has to worry about what data needs to be protected from race conditions. Also, in HTM atomic sections are not prolonged by

software instrumentation of memory accesses. Since instrumentation does not need to occur the libraries that are called from code which uses HTM do not need to be rewritten to use such atomic sections or recompiled to be instrumented.

One might ask whether or not HTM is capable of achieving the same level of performance of fine-grained mutex locks, especially HTM which uses only a global lock fallback. This work shows that it is indeed the case that the vast majority of multi-threaded programs when run with global lock fallback HTM, in this case RTM, are competitive with their fine-grained mutex (futex) implementations.

1.1 Motivation

Why should a programmer want to use hardware transactional memory over mutex locks?

This question can be answered with the ease of use of hardware transactional memory while not introducing prohibitive overhead. Given that this work has already introduced the issues with the usability of mutex locks and STM this work now provides an example of the low overhead possible with HTM compared with other synchronization methods. To show this advantage we will look at a particular application included in the PARSEC [13] suite of programs called "fluidanimate". The distributors of PARSEC describe fluidanimate as a program for "[f]luid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method" [13]. It is also a program which uses POSIX threads and mutexes to manage concurrency. Given that, in this case, a fine-grained mutex version is available can it be improved for performance or preserve performance and be simplified for maintainability?

In theory a programmer could modify the source code of the program so that all locks referred to a single global lock visible across the entirety of the program. If that programmer did this they would be spared having to follow a lock-ordering scheme or much of the effort in proving, testing, or re-proving various progress or correctness guarantees as there would only be a single, recursive, blocking lock. This might not be a bad approach if this does not affect the correctness/deadlock-freedom of the program. However, if more than one thread of a program spends a large amount of time holding *different* locks then those threads will now be unnecessarily delayed when converted to the global lock approach. This turns out to be the case with fluidanimate, as it spends an appreciable amount of time (compared to the duration of the program) holding locks at higher numbers of threads which, when converted to the global futex implementation, causes a noticeable slowdown.

One might also consider going with the approach of maintaining multiple mutex locks but using HTM to attempt to allow concurrent threads to act as if they both own the lock at the same time. This approach implies the expectation that the overhead of the transaction and its aborts will not be worse than the improvement from eliding the lock. In cases where two

threads which acquire the same lock would rarely cause conflicting accesses with each other this could be a good idea. However, because the fine-grained approach also has to account for the fallback path it has the same difficulties that come with guaranteeing that the mutex version is free from deadlocks, livelocks and data races in addition to maintenance complications. This occurs because while HTM is the fast path, progress cannot be guaranteed with RTM alone (or any best-effort HTM for that matter) and may need to revert to the mutex locks to move forward, which can reproduce the fine-grained mutex execution. It also turns out that for fluidanimate any improvements in throughput gained with speculative execution are lost, which indicates that the contention represented by the fine-grained locks closely represents the true data contention of the program.

Lastly, we come to the combination of a global futex lock and HTM. This approach has the simplified deadlock properties of the global lock approach as well as the ability for multiple threads to optimistically "own" the same lock at once. As stated earlier the global lock approach can become inefficient if threads would have spent a lot of time holding different locks but now hold the same lock. But because HTM allows those threads to "mutually exclusively share" (an intentional oxymoron) that lock they can largely simulate fine-grained locking with a single lock. As a result, we end up with a program with much simpler synchronization without incurring the full performance penalty associated with it.

Configuration	Time (s)
Fine-grained Futex	69.1243
Global Futex	457.904
Fine-grained HTM	76.3357
Global HTM	79.861

Table 1.1: fluidanimate Proof of Concept: region-of-interest duration under different configurations at 8 threads.

When we apply these approaches to fluidanimate this bares out. Table 1.1 shows what happens when each of these configurations is applied to fluidanimate. Since fluidanimate spends a large proportion of its time inside critical sections (this is shown in Figure 5.10 but needs to be explained further) it suffers significantly when converted to using a global lock from fine-grained mutexes with a 6.62× slowdown. Even fine-grained fallback HTM does not end up achieving a speedup but is slowed down 1.10× compared to fine-grained mutexes alone. Lastly, the global fallback HTM approach also slows down 1.16× compared to the fine-grained mutex approach but, like the global lock approach, has a simpler locking scheme. The fact that the global mutex version has such a high slowdown but the global fallback

HTM version suffers so little comparatively indicates that, for at least some programs, coarse-grained locking can incur a high overhead which HTM may counteract.

1.2 Contributions

This thesis presents the following research contributions:

- This work describes a conversion of glibc's implementation of mutex locks so that a program originally written to use fine-grained mutex locks can be evaluated for performance using other locking implementations without modifying the source code. glibc originally provided a fine-grained futex and fine-grained fallback RTM lock implementation. This work added a global lock futex implementation as well as a global lock fallback RTM implementation. In addition to the already listed implementations versions of both fine-grained fallback RTM and global fallback RTM were created with trylock aborts removed to examine their impact on performance.
- This work shows that global fallback lock HTM is competitive with fine-grained mutex locks in terms of performance for nearly all of a set of 23 multi-threaded programs. Notably, the implementations are modified by changing the libraries to which the program links, leaving the source code intact.
- Additionally, a performance pathology of the lock elision code used in glibc, termed as *lock cascade failure*, is analytically described, to the knowledge of the author of this work, for the first time, in this work. This can cause atomic sections which do not account for the effect to grow quadratically in duration which are otherwise bounded by periodic interrupts and transaction nesting depth. Additionally, a method to determine with high probably the maximum nesting depth is provided to aid in determining this bound in RTM-capable processors. Recommendations on how to prevent it are provided.
- As a part of this work a set of open-source modifications are made available which allow global locking and global lock fallback.

1.3 Thesis Organization

Chapter 2 explains the background necessary to understand the difficulties with using mutex locks as well as the basic mechanics of HTM. Chapter 3 discusses other attempts to bring transactional memory implementations into practical use. Chapter 4 describes the changes made to the GNU C Library to support global mutexes (including the HTM fallback version), the statistics recording mechanics used for data gathering as well as some of the differences

between mutexes and HTM as well as fine-grained and global locks. Chapter 5 lays out the experimental setup used in this thesis as well as the performance and characteristic results of each synchronization implementation including important data gathered. Chapter 6 summarizes this thesis and the main takeaways as well potential next steps. Appendix A lists short pieces of code to illustrate how RTM transactions are used and how to extract otherwise hidden implementation-specific data.

Chapter 2

Background

From here onwards the "streams of execution" and the overall context will be assumed to be threads. Some of the concepts discussed here also apply to other models of parallel execution but are not discussed here.

2.1 Mutex Locks

Mutex locks are characterized by the fact that they may only be owned by 0 or 1 threads simultaneously. The exclusive ownership of the lock can be used to allow only 1 thread access to a particular data item at a time, a property known as mutual exclusion. This behavior can be formally stated as in Equation 2.1. Where $owns(x, a, t)$ means "thread x has ownership of lock a at time t".

$$\begin{aligned} \forall a \in Lock,\ x \in Thread,\ t \in Time,\ owns(x, a, t) \rightarrow \\ (\nexists y \in Thread \mid (x \neq y) \wedge owns(y, a, t)) \end{aligned} \tag{2.1}$$

Different techniques exist to enforce the property of mutual exclusion with varying tradeoffs. Traditional implementations deal with multiple threads contending for the lock by causing threads requesting ownership to wait in some way.

The use of mutex locks is motivated by the need to prevent race conditions when accessing data used by more than one thread. A simply illustrated race condition involves incrementing a shared counter.

1	c = b
2	c = c + 1
3	b = c

Table 2.1: Increment with Race Condition: b might lose increments.

1	**lock**(a)
2	c = b
3	c = c + 1
4	b = c
5	**unlock**(a)

Table 2.2: Atomic Increment: no lost increments if b is protected by a.

In Table 2.1, assuming c is thread-local and b is shared, it is possible that a thread reads the value for b and, before it stores the new value back, another thread reads b's value. At this point at least one of the threads will have the increment it has or will perform on b nullified. Whereas for Table 2.2 lock a ensures that two or more threads executing this same piece of code cannot reach a state where the atomic section *appears* to have been logically interlaced with other threads (which also want the lock). The fact that any thread attempting to take ownership of lock a while another thread owns it ends up waiting is incidental as long as the logically atomic operation *appears* atomic.

Mutex locks themselves do not directly protect data from being exposed to race conditions like the one illustrated by Table 2.1. Instead a programmer must ensure that mutex locks are acquired and released in such a way that, as a side-effect, shared data is not accessed in a way that could potentially lead to a violation of the program's consistency expectations.

2.1.1 Difficulty with Fine-grained Locking Designs

For the sake of clarity, deadlock and livelock are discussed in this section in the sense of not being starvation-free, i.e. some non-zero number of threads are unable to complete their work, implicitly. Deadlock and livelock in the sense of being lock-free, i.e. some non-zero number of threads are guaranteed to make progress after a finite period of time, are used explicitly as "system-wide" deadlock or livelock.

2.1.2 Deadlocks

As an artifact of the way in which mutex locks help to ensure against race conditions it is possible that an improperly written program can end up in deadlock, preventing forward progress. Figure 2.1 lays out the situations in which this can occur with mutex locks alone.

Figure 2.1(a) illustrates a generic deadlock in which a thread t has some dependency on some lock being released while holding lock a but those dependencies end up with the thread eventually relying on it releasing its own lock to progress. Figure 2.1(b) shows the simplest form of deadlock in which thread t acquires lock a and then tries to reacquire it without releasing it. This pattern does not necessarily lead to a deadlock if lock a is a recursive mutex. A recursive mutex only acquires the lock (or performs a blocking wait) if it does not already own the lock and only releases the lock when the number of calls to acquire the lock and calls to release the lock are matched for the given thread.

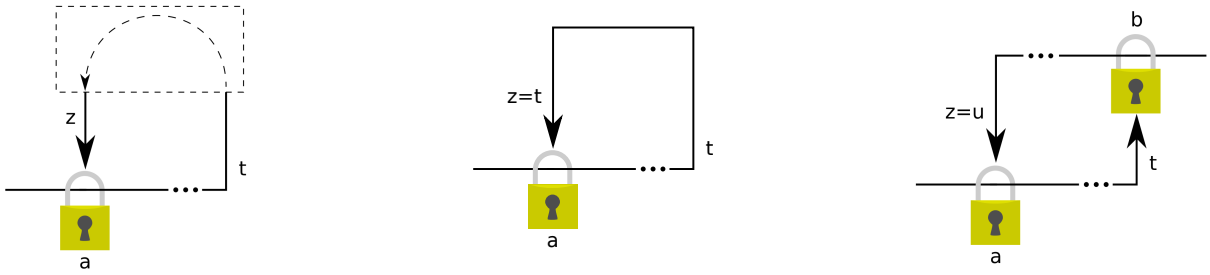
Figure 2.1(c) illustrates a case of deadlock which can occur from failing to enforce proper lock ordering. One way to create a lock ordering is to create a partial order "$<$" over the set of locks which can potentially exist in a program and forbid a thread from attempting to acquire a lock which is "smaller" (according to the ordering predicate) than any one or more of the locks the thread already holds as in Equation 2.2.

$$\begin{aligned}&\forall x \in Thread, a \in Lock, t \in Time, < \in PartialOrder, (\\&\quad AcquireDAGValid(x, a, t, <) \equiv \\&\quad \forall l \in Lock, owns(x, l, t) \rightarrow (l < a)\\&)\end{aligned} \tag{2.2}$$

Such an ordering and its transitive closure forms a *directed acyclic graph* (*DAG*) and the currently held set of locks for a given thread can be viewed as a path on that DAG. Since no subgraph of a DAG can form cycles no dependency cycles can form within or between threads using locks alone. However, if the ordering predicate "$<$" does not form a DAG then proving that the program is deadlock-free is more involved, if it is deadlock-free at all. In Figure 2.1(c) it is possible that the statements $a < b$ and $b < a$ are both simultaneously true due to a programming error. As a result it could be the case that thread t acquires a, thread u acquires b, thread t blocks attempting to acquire b and thread u blocks attempting to acquire a with neither thread making progress.
Partial orderings of mutex locks is a useful mechanism for preventing deadlocks. However, guaranteeing that a given system cannot result in a deadlock requires that the programmer knows what order threads can acquire and release locks statically. If the behavior cannot be sufficiently predicted ahead of time because of its difficulty or impossibility then using locks safely becomes difficult.

2.1.3 Livelocks

Livelock is a condition that occurs when a thread continually performs work but cannot ultimately progress. A livelock condition can appear in fairly subtle ways, the following example is used to show why it can be hard to determine if a program is or is not livelock-free.



(a) Generic deadlock (b) 1 thread deadlock (c) 2 thread deadlock

Figure 2.1: Deadlock Scenarios: there are an infinite number of deadlocked configurations. Deadlock occurs for a thread t when it waits on thread z to release a lock while z waits on t to release lock a.

Take a program that read-modify-updates an object Q_0, then Q_1, then Q_2 with each read and update occurring together, atomically. Each of these objects is associated with a lock L_0, L_1 and L_2 respectively. If locks A, B and C are instantiated with the lock ordering property $A < B < C$ it is possible for this program to operate on the objects in six permutations, P0 through P5. For example, one permutation could set $L_0 = A$, $L_1 = C$ and $L_2 = B$.

The programmer could choose to acquire A (L_0), operate on Q_0, acquire B (L_2), acquire C (L_1), operate on Q_1 then operate on Q_2 and then release all the held locks. Such an approach respects the lock ordering requirement but does work on Q_1 while holding B (L_2), which is not logically necessary except to prevent a potential deadlock involving a thread which respects, and exercises, the lock ordering.

A clever programmer might consider getting around this situation by using non-blocking acquires (e.g. trylocks) and roll-back the work (including releasing acquired locks) of the atomic operation upon failure to acquire a lock in a certain period of time and immediately restart the operation. A trylock only needs to occur when the lock order is being violated to avoid a deadlock. All permutations of the locks A, B and C are shown in Figure 2.2 as well as whether or not a trylock must occur to avoid deadlock (under the assumption that P0, which uses only blocking acquires, may be concurrent with any of the permutations). The rest of this discussion assumes lock acquisitions are performed as shown in Figure 2.2. It is straightforward to show that any subset of the permutations are deadlock-free as no permutation attempts a blocking acquire which violates the lock ordering and any acquires which do violate that ordering immediately release all locks, preventing any dependency cycles. No guarantees are made here that such a program is faster or slower than using only blocking acquires which respect lock ordering under a given workload.

These trylocked permutations *may* introduce system-wide livelock into a system which makes use of them. The "may" is emphasized as it is possible that subsets or other constrained

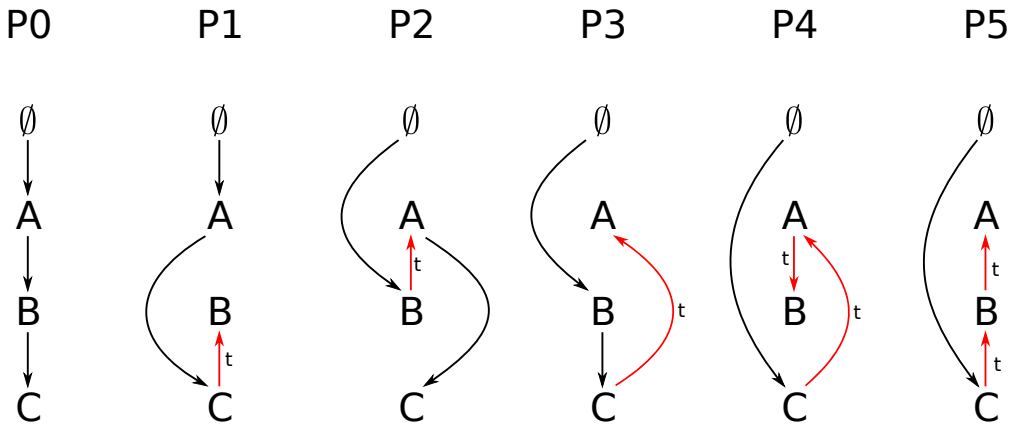


Figure 2.2: Trylock Livelock: black arrows indicate blocking acquisitions, red arrows marked "t" indicate an attempt at a trylock acquisition which finishes in a finite period of time. The arrow's target indicates the lock (A, B or C) it is attempting to acquire. Attempting to acquire 3 locks in any order permits these permutations as long as a failed trylock rolls back all work and jumps to ∅ and releases the held locks before retrying while still being deadlock-free.

uses of the permutations may be used while still ensuring that system-wide livelock is impossible. For example, a system which only issues operations using P0 and P5 (using locks which at least guarantee finite bypass) can guarantee that some thread makes progress [1]. If the system does not know *a priori* which permutations of a set, S, of statically known permutations a thread will issue then some values of S are safe from system-wide livelock while other values are not.

This variation in system-wide livelock-freedom is due to the fact that some states of the atomic operation permutations can cause others to roll back their work where others cannot. Take for example a system which only issues operations using permutations P0, P1 and P3. It is possible that a thread issuing P0 acquires lock A, and acquires lock C while another thread issuing P3 acquires lock B. At this point P3's thread cannot advance as it is using a blocking acquire on lock C but P1's thread holds it. P1's thread must fail its trylock because P3's thread already holds it and undoes its work and releases locks A and C then attempts to retry. P1's thread can reacquire lock A but block on C (as P3's thread already holds it) causing P3's thread to fail its trylock, undo its work and release B and C. P3's thread can reacquire lock B and P1's thread can reacquire lock C. This brings the system back to a previous state, resulting in a potentially infinite loop of changing states, in other words a

[1] A system which uses only P0 and finite bypass locks can guarantee starvation-freedom in addition to the absence of livelocks but no system which uses any of the other permutations can guarantee starvation-freedom as a system may continually issue P0 operations, denying progress to other permutations. However, systems using only P0 and P5 can guarantee that if a finite number of atomic operations are started (or infrequently enough) all atomic operations complete in finite time, making its starvation-freedom conditional.

livelock.

It is possible for a livelock between some number of threads to grow to encompass any larger number of threads since any two threads which do not hold any locks (whether due to rolling back work or because they have not attempted the operation yet) that try the same permutation are indistinguishable in this protocol. This means that permutations which can mutually cause each other to roll back their work can expand to system-wide livelock[2]. Guaranteeing that the livelock eventually ends would require comprehensive knowledge about that system's instruction timing and thread scheduling.

However, system-wide livelocks in the proposed program can be statically ruled out if the set of usable permutations, S, is known statically as well. Some definitions will be helpful to show this. The "lively" relationship can be stated as in Equation 2.3. Note that P$k[n]$ is a permutation state of permutation k about to acquire lock L_n. To summarize, a is in a "lively" relationship with b if a is rolled back by b or b is a's successor, this relationship is not necessarily symmetric.

$$\begin{aligned}
&\forall a \in PermutationState,\ \forall b \in PermutationState,\\
&\quad Lively(a,b) \equiv\\
&\qquad (\exists k \in \mathbb{N},\ \exists n \in \mathbb{N},\ a = Pk[n] \wedge b = Pk[n+1]) \vee\\
&\qquad RolledBackBy(a,b)
\end{aligned} \tag{2.3}$$

The *RolledBackBy* relationship is true if-and-only-if the locks held by both operands are disjoint and the 1st operand is attempting to trylock a lock held by the 2nd operand. This can be formally stated as in Equation 2.4, note that time is part of the "owns" relationship but is largely irrelevant for *RolledBackBy*'s definition. *Executing* is true if-and-only-if the 1st operand (a thread) is executing the 2nd operand (a permutation state) at the 3rd operand (a point in time). A permutation state *Trylocks* a lock if-and-only-if it is attempting to acquire it via trylock.

$$\begin{aligned}
&\forall x \in Thread,\ \forall y \in Thread,\ \forall t \in Time,\\
&\forall p \in PermutationState,\ \forall q \in PermutationState,\\
&\quad RolledBackBy(p,q) \equiv\\
&\qquad Executing(x,p,t) \wedge\\
&\qquad Executing(y,q,t) \wedge\\
&\qquad (\nexists a \in Lock, owns(x,a,t) \wedge owns(y,a,t)) \wedge\\
&\qquad (\exists a \in Lock, Trylocks(p,a) \wedge owns(y,a,t))
\end{aligned} \tag{2.4}$$

If no cycles occur in the graph of "lively" edges and permutation states then no system-wide livelock can occur. Figure 2.3 shows the cycles in a graph of the states of each permutation

[2]Regardless of the subsequent work.

with a "lively" relationship. That figure indicates that the maximum sets of permutations which do not permit a system-wide livelock are {P0, P1, P4, P5}, {P0, P2, P3, P5} and {P0, P3, P4, P5}. One can imagine a situation where a clever programmer writes a program

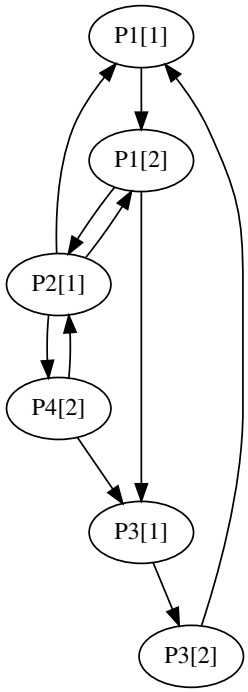


Figure 2.3: Livelock Cycles: Each node, P$m[n]$, is a state of permutation k (i.e. P0, P1, P2, P3, P4 or P5) which is about to attempt to acquire the nth lock (0th, 1st or 2nd lock) as described in Figure 2.2. For example, P0[0] attempts to blocking acquire lock A while P5[2] attempts to trylock lock A. The edges are the union of edges P$k[n] \rightarrow$ P$k[n+1]$ and nodes which can fail a trylock $\rightarrow$ nodes which can cause that failure. Edges and nodes which cannot contribute to cycles are removed. Cycles represent potential system-wide livelocks.

that can be proven to be free from both deadlocks and system-wide livelocks by selecting a subset of one of these sets but does not insert sanity-checking code which ensures that only a subset of one of those sets is used. In such a situation a maintainer may return to the program, possibly years later, unaware of the constraint in the program. The maintainer might add code which uses a new permutation which is incompatible with the maximum, safe sets. This maintainer may or may not see the detrimental effects of livelock because the particular hardware, scheduler or system load the code is tested with might not match the production system.

The main takeaway here is that protocols which are susceptible to livelock and system-wide livelock and factors which cause a system to suffer from it can be hard to engineer for when intertwined with the application itself. After all, the system under examination just involved acquiring three locks in some order.

2.2 Transactional Memory

Transactional Memory (TM) was introduced as a way to implement data structures that are lock-free, in the sense that they are not starved for work, with support from hardware mechanisms like cache [5]. Transactional memory is similar to mutex locking in that it is also built around atomic sections that must appear to be logically indivisible called *transactions*. However, transactions are distinct from locked atomic sections in that they do not necessarily artificially delay the execution of an atomic section by waiting. Instead, in the normal case, they optimistically begin execution as soon as the atomic section is encountered. If there are any conflicts which could cause the program to become inconsistent by failing to ensure logical atomicity then the transaction is aborted, canceling the effects the transactions would have taken and resumes execution just before the start of the transaction.

Since the time transactional memory was proposed as a hardware mechanism Software Transactional Memory (*STM*) system has been introduced [9]. When distinguishing between the two *hardware transactional memory* (*HTM*) will be used to refer to the original, hardware-centric conception.

Intel's *Transactional Synchronous Extensions* (*TSX*) contains an implementation of HTM known as *Restricted Transactional Memory* (*RTM*) which usually cannot guarantee that a given transaction can complete for various reasons [11]. As a result this implementation is known as a "best-effort-only" method of synchronization [12]. In situations where best-effort-only HTM cannot guarantee progress, fallback locks are critical to ensure that a transaction eventually completes.

2.2.1 Fallback Locks

Among the reasons why an HTM transaction cannot, in general, be guaranteed to complete [11] are:

- True data conflicts: occurs when a transaction cannot guarantee that a transaction's start, commit and all operation in-between appear atomic. Because the conflict resolution scheme has the transaction whose data was made outdated abort it is possible that transactions could mutually abort each other indefinitely [12] when not using a fallback lock.
- False data conflicts: which behave similarly to true data conflicts but occur due to cache lines being shared between processors as opposed to the data itself.
- Cache pressure: if the cache cannot store the data from the *read-set* and *write-set* of the transaction it must abort. This also includes cases when the set associativity of the cache prevents it from storing a cache line even while other sets are not full.

- Incompatible Instructions: some assembly instructions are guaranteed to cause aborts while others may do so according to implementation.
- Transactional Nesting: when composing transactional routines of other transactional routines transactions may be nested. However, the TSX specification laid out in [11] only guarantees that 7 levels of nesting are allowed.
- Software XABORT: some high-level guarantees may be required of the software by the programmer for which an explicit abort prevents transactional completion. This factor might be controllable by a software engineer with sufficient knowledge of the application but may also prevent a transactional region from committing. This issue will appear in the analysis of memcached in Section 5.3.1
- Exceptions, Traps and Interrupts: notably, this can apply to timer interrupts meaning that if a transaction is interrupted by the kernel for thread preemption the thread may be aborted [10]. This applies to thread preemption but not directly to other threading techniques such as hyper-threading or fibers.

As a result it is typical that if a thread's transaction surpasses some number of transactional aborts or a condition in which re-execution is not expected to result in a transactional commit then the thread acquires a fallback lock.

HTM-Lock Coherence

All transactions which may be replaced by an acquisition of a fallback lock must also check if the fallback lock is held and abort if so. This ensures that the HTM execution path is consistent with the fallback lock execution path and does so in two ways.

The first is that a thread following the HTM path may begin a transaction while the fallback lock is held and commit its state before the lock is released. This may cause a race condition because the thread holding the fallback lock must assume that it is the only one accessing the data guarded by that lock. When the thread following the incorrect HTM path commits it could violate that assumption and corrupt the data the thread on the fallback path sees. Explicitly aborting from the transaction when the lock is observed (i.e. read) as being held prevents this behavior in this case.

The second way is that a thread following the HTM path will observe (i.e. read) the state of the lock sometime between the start and commit of a transaction as being free. By simply observing (reading) the lock as not being held causes the memory locations which describe the lock status to be added to the transaction's read set. If those memory locations are overwritten before the end of the transaction, i.e. the lock is acquired by a thread on the fallback path, then the transaction will abort without an explicit instruction. Failing to

abort in this case can also lead to race conditions as above. Implicitly aborting from the transaction when the lock state changes prevents this behavior in this case.

For this work the fallback lock is read before any application memory locations are accessed. This is the approach used in [14] for code samples not marked "wrong". Such an approach is referred to as *eager subscription* in [15]. *Lazy subscription* instead reads the state of the fallback lock as late as possible [15].

Lazy subscription techniques are not considered here as they cannot guarantee *opacity* without potentially invasive analysis [15]. Opacity is important because it means that the transaction will always observe a consistent state (according to the isolation requirements and so long as it is capable of committing) [15]. In [15] an example is given which relies on the application to cause RTM's hardware sandbox to capture an exception when an inconsistency occurs. The example given in [15] uses the invariant $Y = X + 1$ and invokes an exception by calculating $\frac{1}{Y-X}$ when an inconsistency causes $X = Y$ to be true simultaneously with the division in the transaction. If lazy subscription is distinct from eager subscription in a hardware system then it implies that cache lines can be updated by a snooping mechanism after a transaction on that processor has started but before that line has been added to the read or write set of that processor's transaction. This would mean that any data read before the transaction reads the cache line(s) that fallback paths are obligated to write to then could be inconsistent with any other data. Such consistency behavior is incompatible with that of a single global lock.

Lemming Effect

Since the first piece of code inside a transaction determines if the fallback lock is held and aborts if so, it is wise to prevent unnecessary aborts by first checking outside the transaction that the lock is available. Failing to do so could lead to the *lemming effect* [14, 16]. The lemming effect starts when a transaction exhausts its number of tries and the thread reverts to the fallback lock path. Another thread then comes along and attempts to start a transaction, sees that the fallback lock is held and aborts explicitly (and either quickly exhausts its transaction attempts or immediately falls back to the lock). Once the lemming effect begins this feedback occurs until no more threads are attempting to acquire the fallback lock and the thread currently holding the lock releases the lock before any new incoming thread falls back on the lock.

The lemming effect can be prevented by making sure that while the fallback lock is held transactions are not attempted immediately one after another. This can be done by reading the lock state in a spinlock-like loop before the transactional section and not attempting to start a transaction until the lock is free. The point at which the thread breaks out of the spinlock-like loop is the earliest chance the transaction could begin and commit successfully.

Chapter 3

Related Work

3.1 Draft C++ TM

A proposal was put forward to include transactional memory constructs directly into C++ [6]. This proposal has been included in implementations of the GNU Compiler Collection (GCC) for C++ and C which behaves as if it were guarded by a single, global lock [17]. Since the proposed C++ constructs are at the language level they also require that any program adapted to use transactional memory this way be modified at the source level, as they would have to be converted from another synchronization method to C++ transactions. This language extension is not explicitly targeted at Hardware Transactional Memory or Software Transactional Memory alone.

3.1.1 Transactionalized Programs

Ruan *et al.* applied C++'s proposed transactional memory constructs to memcached to assess the proposal [7]. Their analysis involved: applying different types of transactional blocks to various parts of memcached previously guarded by locks, modifying condition synchronization, replacing volatile and atomic variables with regular variables protected by transactions [7]. Ruan *et al.* claimed the need to convert some parts of code to transactions in order to support the portions they had originally sought to convert [7]. They claimed to have to make this additional transformation for per-thread statistics counters because "*any* operation on a mutex lock is unsafe to perform in an atomic transaction" (emphasis theirs) [7]. Ruan *et al.* claimed to need to transform condition synchronization variables manually because the C++ proposal did not support condition variables [7]. However, there is now work which attempts to solve the problem [8]. Additionally, Ruan *et al.* had to

manually verify that volatile and atomic variables did not attempt to communicate between critical sections/transactions and voiced skepticism that this would be applicable to larger programs [7].

According to Ruan *et al.*, certain functions used in transactions are duplicated (by the compiler), one version to be used from outside transactions and another version for use inside of transactions that have their loads and stores instrumented for use in rollbacks on abort [7]. As the language constructs are lexically scoped to aid in this instrumentation, beginning and committing transactions must be done at the same lexical level.

Ruan *et al.* conclude that incremental transactionalization is a myth [7]. However, the reason for this seems to be due to the lack of the transactional "safety" property in locking/unlocking mutexes. Hardware Transactional Memory is not restricted from acquiring a mutex in the normal way. They also assess that the GCC C++ TM implementation assumes that serialization is common and optimize for that case at the expense of the optimistic case [7]. This assumption is not necessarily well-founded. The authors also suggest the introduction of more transaction-safe libraries to obviate the need for workarounds and re-implementing library functionality [7].

Ruan *et al.*'s experiments on partially applying the C++ TM extension to memcached resulted in, *at best*, an approximately 2× slowdown at the high end of the number of threads in their experiments for the vanilla STM library implementation [7]. However, they were able to regain much of the performance when they removed the reader/writer lock from the library but this was on the condition that no thread could become serialized [7]. When transactions were fully applied memcached performed even worse [7].

Wang *et al.*'s work, which evaluates a C++ TM version of PARSEC, which uses condition variables which are safe in that transaction implementation, yielded noticeably poorer performance for *both* STM *and* HTM [8]. The HTM version appears to also have used the same source code as the STM version, just a different implementation of transactions. As a result Wang *et al.*'s work does not use the same source code as this work and is instead an STM-compatible transformation of the original PARSEC code.

3.2 GNU C Library Mutex Locks

The GNU C Library (glibc) is a software library used in Unix-like operating systems to handle common functionality required by programs written in the C ([18]) as well as C++ programming languages. While necessary for C/C++ programs in general, it is important to this discussion because it has already had hardware transactional memory introduced into it [19]. The glibc use of HTM is switched on or off by a configuration flag as well as code within the library which detects if the processor on which it runs is capable of HTM. Intel's

Restricted Transactional Memory is supported by glibc when running on processors which support it and the library is built with the correct configuration.

This usage of hardware transactional memory provides an important baseline from which to create other versions of the library which also use hardware transactional memory. Instead of requiring that the application writer use a language-level interface in the targeted application glibc has code on the library side alone which, instead of performing operations on the standard futex, elides the futex, pursuant to certain rules, and silently carries the transactional state from the library to the application.

The way hardware transactional memory is used within the library has some properties that need to be explained. The first is the fact that transactions are tied to individual application-level mutexes. This is important because it means that the library allows for fine-grained fallbacks (whereas the GCC implementation of TM assumes a global fallback). This does not mean that a global fallback lock is impossible with the unmodified library since it can be easily done by creating one, recursive, mutex used throughout the application to the same effect. However, using multiple, distinct fallback locks has different behavior apart from simply causing threads to serialize less often in this case.

Whenever a mutex is constructed it sets a variable which indicates how many times a thread is allowed to try to carry out a hardware transaction before being required to fall back and acquire the futex (this is set to 3 tries). This variable is initialized by `aconf.retry_try_xbegin` in code Table A.1 but is absent for trylock elision in Table A.2. A thread might fall back before trying that many times if an abort reason is given in the abort code which indicates that it should not attempt to retry a transaction. Note that this does not occur in the same way for trylocks because trylocks without elision are assumed to not put a thread to sleep by blocking which allows a programmer to assume that a call to a trylock to be very fast especially if it fails. While the elision locking code has a loop (code Table A.1 lines 9-39) which allows for transactional retries the elision trylock does not.

In order to prevent a lock from being elided resulting in the program trying a transaction constantly and failing to commit, a fixed backoff is introduced (this is set to 3). This means that when a lock falls back and the abort reason indicates that it should not retry the transaction on that futex it will acquire the futex 3 times normally before allowing it to be elided again. This backoff is tracked by `adapt_count` in code in Table A.1 and Table A.2. Given that the library is set up with multiple mutexes in mind it will possibly incur less of a serialization penalty compared to a global lock implementation since other threads are less likely to also be in contention for that lock so there is less of a consequence to serializing. But when using a fine-grained implementation more futex acquisitions have to be performed to clear the backoff period, possibly resulting in more serialized work.

One should note that Intel's Restricted Transactional Memory is flat-nested, meaning that

if an abort occurs while more than one level of transaction is active all levels for that thread get aborted and their work rolled back (it also means there are no partial commits) [10]. As a result, if there is more than one lock currently being elided in place of a transaction and an abort occurs then the outermost lock will handle the abort and have its try counter decremented instead of the more local innermost lock.

In glibc's elision implementation of trylock one of the first (also unconditional) statements is an explicit abort. The presence of this explicit abort is explained as being due to the need to handle an "assert" condition [19] (in which a program may perform a sanity check on the state of the program by ensuring that a certain lock is held). This behavior is considered by the author to be indicative of an already existing bug in the application's code [19].

Chapter 4

GNU C Library and Hardware Transactional Memory

As described in Section 3.2 the GNU C Library (glibc) is already capable of performing lock elision using Intel's Restricted Transactional Memory (RTM) for fine-grained mutex locks, as well as the requisite mutex locking using futexes. In this section the necessary changes to the library to introduce global locking are described.

While a programmer would not necessarily want to have all of their mutex locks silently converted to a single lock out of their control in some applications, especially those that thoroughly exercise the functionality of the pthread mutex locks, it can provide insight into whether or not certain synchronization approaches are effective without a significant investment of effort.

Where the behavior of hardware transactions and global locks cannot cover the behavior provided by mutexes is discussed later. Such issues are best resolved by fixing or deprecating the original code behavior or by using a synchronization method which allows for sufficiently weak isolation (which a global lock by itself does not permit).

4.1 Modifying the Library

Since glibc already incorporates RTM into its implementation of mutex locks introducing global locking into the library will provide the same kind of simplified programming model provided by the proposed C++ TM without source code modifications.

global_lock.{c,h}

`global_lock.h` holds the declaration for a process-wide global mutex where `global_lock.c` holds its definition and initialization. The global mutex is initialized at startup in the global scope but could have been initialized by the first thread to call a lock-related API. There would need to be a mutex-less method of making sure that the mutex was initialized otherwise the initialization problem regresses infinitely. Atomic fetch-and-set to an auxiliary variable with a known initialization at process startup and spin-waits is sufficient to do this.

Since the expectation is that the original program that we are dealing with has at least one lock, but likely more, this case needs to be considered upon conversion to a global lock. It is possible that the original application writer had a thread acquire some lock and then acquire another lock without releasing the original lock. This behavior does not necessarily cause a deadlock in the original code but if those two locks were the same and non-recursive it necessarily would. This is the case with a global lock. So the global lock is guarded in the same way as a recursive mutex. A thread-local counter `GlobalLevel` is initialized to zero when each thread is started so that it does not act as if it has the global lock. When the thread makes a call from the application to attempt to acquire or unlock *any* mutex lock if the compiled version of the library is enabled to use the global lock then all application locks will be quietly substituted with the global lock inside the library. If the global lock is being used then the thread-local `GlobalLevel` may take on the change of state instead of the global lock.

nptl/pthread_mutex_{lock,timedlock,trylock,unlock}.c

The labels `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_timedlock`, `pthread_mutex_unlock` (or more precisely the labels to which those labels alias) were "lifted" away from their original definitions. Meaning that those labels had a `_sub` suffix appended and moved to declare and define new functions. `pthread_mutex_lock` uses a preprocessor switch to choose between using a global lock or the application's fine-grained lock. If the application lock is to be used then it simply calls the original implementation. If the global lock is to be used then the thread first checks `GlobalLevel`, if it is zero then it calls the underlying lock implementation and increments `GlobalLevel` if successful but if it is non-zero then it simply skips the call to the underlying implementation and increments the counter. The source code for the similar trylock can be found in Table 4.1.

```
1  int
2  __pthread_mutex_trylock (mutex)
3           pthread_mutex_t *mutex;
4  {
5      int r = 0;
6
7      #if GLOBALLOCK
8          if(!GlobalLevel)
9              {
10                 r = __pthread_mutex_trylock_sub(&GlobalLock);
11             }
12         if(!r)
13             {
14                 GlobalLevel++;
15             }
16     #else
17         r = __pthread_mutex_trylock_sub(mutex);
18     #endif
19
20     return r;
21 }
```

Table 4.1: Mutex Lock: permits use of a global lock via preprocessor switch and implements half of the recursive locking behavior. Unused code is removed from this excerpt.

elision-{lock,trylock,unlock}.c

Because `__lll_lock_elision`, `__lll_trylock_elision`, `__lll_unlock_elision` (which map to the similarly named files) are already implemented in the version of glibc used for this evaluation only the changes made will be discussed. Their implementation, without the statistics gathering, can be seen in Appendix A. The first addition is code to prevent threads from starting transactions while the relevant lock is held by another thread then starting a transaction and almost immediately aborting because it sees the lock is already held. This anti-lemming effect code (Table A.1 lines 14-17) is simple but critical as it prevents unnecessary serialization.

The other modifications are those used to gather timing information about synchronization methods. Directly after finding that `adapt_count` is greater than zero `elision_stat_self` is called (which is described later in detail) which returns a per-thread structure holding the sum of time intervals and counters. If `_xtest` indicates that the thread is not in a

transaction then it increments a counter, `count_xbegin_flat`, in the per-thread structure to indicate that the thread is about to attempt to begin a hardware transaction. If, in addition to not currently being in a hardware transaction, the transaction try-loop (Table A.1 lines 9-39) has not previously been entered on the current function invocation a timestamp is captured to mark the start of the first attempt at a transaction. Additionally, if the try-loop is currently not in a hardware transaction then before a transaction starts, another timestamp is captured (the timestamp associated with the first transaction attempt is the same as the first timestamp capture). These periods represent the intervals "HTM abort interval" and "HTM interval" in Figure 4.1 and Figure 4.2. Note that the "lock interval" in Figure 4.1 does not exclude a simultaneous occurrence of "HTM interval" within it as it is possible that an outer mutex is acquired normally but that an inner mutex is elided.

When `__lll_unlock_elision` is called if the futex it is concerned with is acquired normally then the futex is unlocked normally. But if the futex is not marked as locked and the unlock function is called then: the program has attempted to unlock a mutex which failed to be locked in the matching lock call, the program never attempted to lock the mutex, or the mutex is elided. Under the POSIX specification not all types of locks need to cleanly return an error so we will leave the original assumption of not needing to cleanly handle bad unlocks [20]. If the futex was elided and after calling `_xend` is no longer in a hardware transaction then a thread-local counter, `count_xend_flat`, is incremented.

Every time a thread attempts to elide a futex a thread-local counter, `count_xbegin` (as opposed to the flat version), is incremented and for every end to a lock elision `count_xend` is incremented. Note that these counters are not durable as they can be rolled back on a transactional abort.

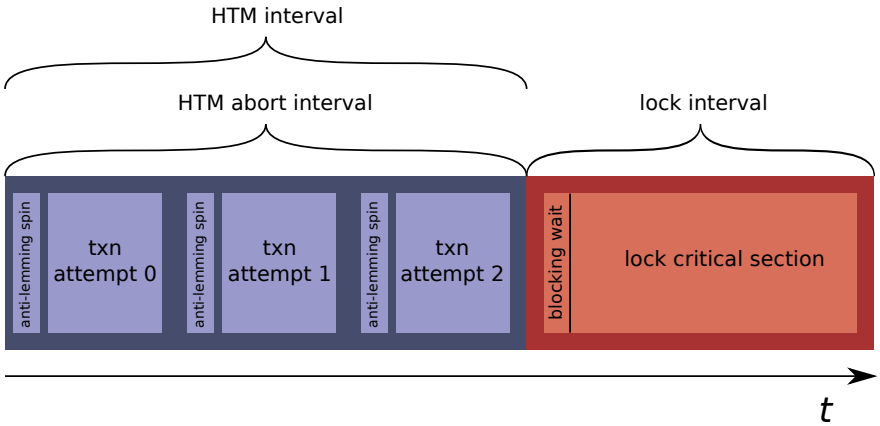


Figure 4.1: Lock Fallback Timing: timing for fallen back synchronization. Height is not a quantity, width is time but is not to scale. There is no gap between the HTM abort interval and the lock interval or "blocking wait" and "lock critical section". Other intervals may be non-zero. The lock interval period is not reported in this analysis.

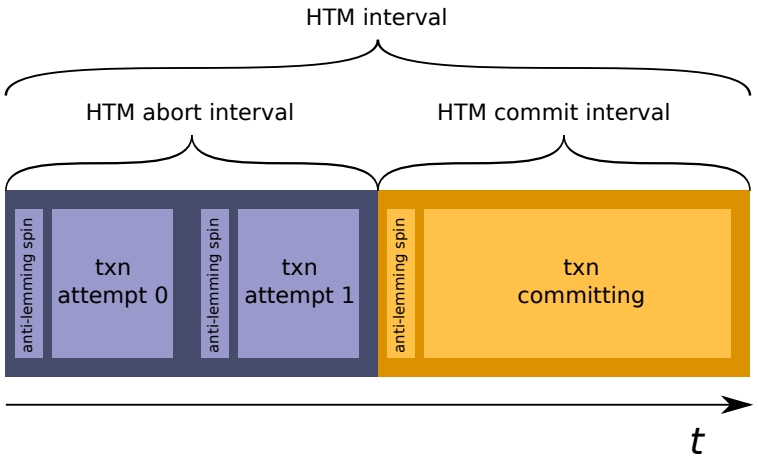


Figure 4.2: Committed Transaction Timing: timing for committed transactions. Height is not a quantity, width is time but is not to scale. There is no gap between the HTM abort interval and the HTM commit interval. The HTM interval is exactly the sum of the abort and commit intervals. Other intervals may be non-zero.

elision-stat.{h,c}

This file defines the interval periods and counters recorded by the elision-{lock,trylock,unlock}.c files in a thread-local structure. This is done using a short, inline function which determines if the thread already has such a structure and returns it immediately if so. If not it allocates the structure and prepends it, wait-free, to a thread-safe linked-list.

These files also define an inline function which gets a timestamp using the `RDTSC` assembly instruction. According to Intel's documentation ([10]) the `RDTSC` instruction is not guaranteed to not cause a transactional abort. To sidestep whether or not this incurs aborts this instruction is not called by the changes introduced to the library during a transaction. If the CPU's timer is an "Invariant TSC" it ticks at a constant rate in various power states [10]. People working with the `RDTSC` instruction ([21, 22, 23]) have found that timer warps do not generally occur between cores on the same socket. Section 5.1.1 will show that the experiments which use this library follow these conditions. The timing methodology is not expected to necessarily be easily *replicatable* but is expected to be *reproducible* and *repeatable*.

nptl/pthread_cond_{signal,broadcast,wait,timedwait}.c

Since the applications that were evaluated required use of condition variables, which interact with mutex locks, these functions had to be modified. Unlike the work involved in making condition variables safe for the proposed C++ TM specification ([8]), which had to reimplement condition variables to conform to the TM's requirements, converting condition variables to use a global lock is much more straightforward. Like the pthread mutex inter-

faces, the symbols `__pthread_cond_timedwait` and `__pthread_cond_wait` were lifted away from their definitions which were relabeled with the `_sub` suffix. The original labels were redefined to use a preprocessor switch to use the global lock or the provided fine-grain lock and the public (non-double-underscored) version of the label aliased to this new definition.

`pthread_cond_signal` and `pthread_cond_broadcast` are left untouched as they do not directly accept an application mutex lock.

4.2 Fine-grained Versus Global Fallback and Futex Versus HTM

4.2.1 Semantic Differences

In some situations locking implementations which use fine-grained futexes, global futex, fine-grained fallback HTM and global fallback HTM can behave differently in the same application.

Deadlock Introduction and Hiding

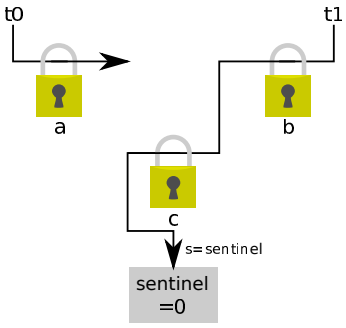
One such instance is the appearance of deadlocks. In an implementation which used fine-grained futex locking one might accidentally leave a valid execution which causes a deadlock. While it is not advisable to determine if a program is free from deadlocks dynamically (or by trial-and-error as the case may be) it is possible that more forgiving lock implementations will provably operate correctly whereas others will not. For example, let us consider a simple case which violates the suggested lock-ordering property in Section 2.1.2 for code Table 4.2.

```
1  | threadA(){
2  |         lock(a)
3  |                 ...
4  |                 lock(b)
5  |                         ...
6  |                 unlock(b)
7  |                 ...
8  |         unlock(a)
9  | }
10 |
11 | threadB(){
12 |         lock(b)
13 |                 ...
14 |                 lock(a)
15 |                         ...
16 |                 unlock(a)
17 |                 ...
18 |         unlock(b)
19 | }
```

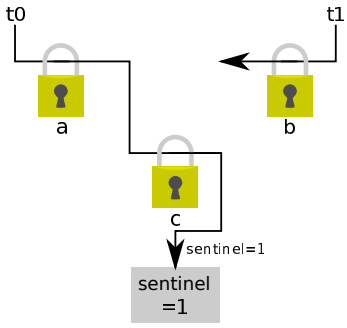
Table 4.2: Deadlock Hiding: deadlocks present in incorrect programs may hidden by different lock implementations.

Using fine-grained futex locks the program can end up with thread A holding lock a and thread B holding lock b simultaneously and then become dependent on each other to release their locks causing a dependency cycle. Fine-grained fallback HTM is similar except that deadlocks which could occur with fine-grained futexes could be avoided due to the fact that thread A will not always need to truly acquire lock a and B, b but this only affects the probability of a deadlock, it does not guarantee that some that would normally occur cannot.

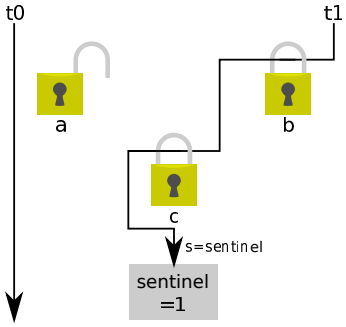
However, global futex and global fallback HTM will not deadlock by mutex locks alone. The reason for this is that for the "global" implementations all calls to lock a mutex have the pointer which indicates which mutex to lock silently substituted with a global mutex. To better handle this silent substitution the global mutex behaves recursively in the futex and HTM versions whether or not the mutex locks used in the original application are recursive. As a result since only one thread at a time can hold a lock at all the graph template in Figure 2.1(a) is constrained to dependency cycles of size 1, i.e. Figure 2.1(b). But, as stated earlier, recursive mutexes prevent deadlocks in this situation so no deadlocks can occur at all (without some other explicit inter-thread synchronization).



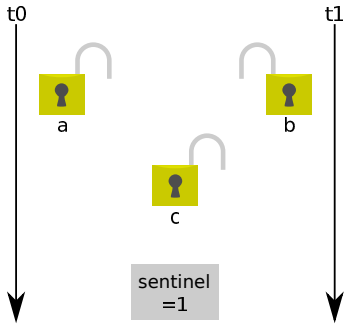
(a) Thread 1 reads sentinel value as 0.



(b) Thread 0 writes sentinel value as 1.

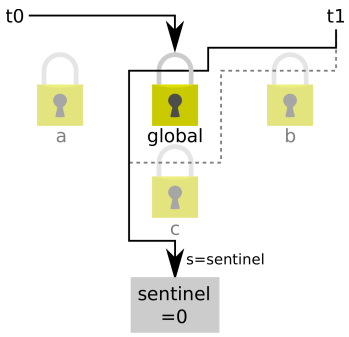


(c) Thread 1 reads sentinel value as 1.

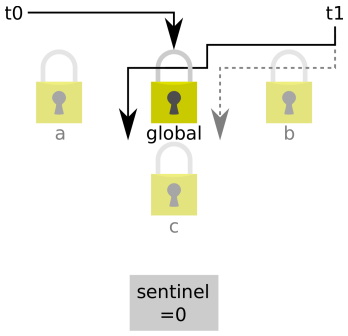


(d) Thread 1 breaks out of loop, releasing locks.

Figure 4.3: Fine-Grained Sentinel Spinlock: Thread 1 spins on both lock c and the sentinel variable while holding lock b. Thread 0 holds lock a and takes lock c and writes 1 to the sentinel which allows Thread 1 to break out of the spinlock loop.



(a) Thread 1 acquires virtual b and virtual c, reads 0 from sentinel. Thread 0 blocks virtual a.



(b) Thread 1 decrements the global lock recursive count (releases virtual c).

Figure 4.4: Global Sentinel Spinlock: an attempt to convert the fine-grained program in Figure 4.3 to a global lock program would end up deadlocking because Thread 1 depends on the progress of Thread 0 but Thread 1 impedes the progress of Thread 0.

Global/Fine-grained Lock Behavior in Communicating Critical Sections

Certain situations which do not involve synchronization primitives can still result in deadlocks when threads require non-serializable behavior with respect to the acquisition and release of the global lock. A programmer might want to use multiple mutex locks when writing a multi-threaded program so that no unnecessary waiting is done. However, it is possible that a programmer could write a program in which a thread acquires a lock, another thread acquires a different lock, both threads do some work and both threads pass messages to each other with those messages protected by a 3rd lock atomically with its previous work. This behavior is not necessarily explicable via a linearizable specification and is not intended to be explained that way. Regardless of how common this situation is it lies outside of the set of programs that can be created using a single global lock. Additionally, this behavior cannot be replicated in transactions alone (it must fall back to mutexes) and as a consequence cannot occur in HTM with a global fallback.

Figure 4.3 illustrates a possible implementation of the above specification. Thread 1 locks `b`. It then locks `c`, reads the `sentinel` value, unlocks `c`, sees that the value of `sentinel` was 0 and repeats while `sentinel` is 0. At some point thread 0 acquires lock `c`, writes 1 to `sentinel`, unlocks `c` and continues on unlocking `a` later on. Thread 1 then locks `c`, reads `sentinel`, unlocks `c`, sees that the value of `sentinel` was 1, breaks out of its loop and later unlocks `b`.

Figure 4.4 illustrates why this behavior could fail for the global lock version. Thread 1 virtually locks `b` (but really locks the global lock), virtually locks `c` (but really increments the recursion counter), reads `sentinel`'s value as 0, virtually unlocks `c` (but really decrements the recursion counter). At this point thread 1 holds the global lock and will spin forever locking, reading and unlocking. This is because while thread 1 holds the global lock thread 0 cannot virtually lock `a`, `c` and set `sentinel`. Whether a deadlock occurs in this situation is down to a race condition but sending messages in both directions can guarantee a deadlock.

This situation is similar to one mentioned by Ruan *et al.* for the proposed C++ implementation of transactional memory involving relaxed transactions communicating to each other via atomic variables [7].

Empty Critical Sections

The synchronization methods used in the modified version of the library follow *single global lock atomicity* semantics for the global locking methods and an analogous set of semantics for the fine-grained methods [24]. This "lock atomicity" can lead to situations which vary with the intended lock-free nature of TM.

Take for example Table 4.3. If `begin()` and `end()` are both lock-free then `threadB` is guaranteed to print "Hello" eventually. However, because best-effort HTM may have to resort to a fallback lock it is possible that `threadA` acquires a lock that it never releases and causes `threadB` to never make progress. Programmers creating an application from scratch may incorrectly assume that this use of HTM is lock-free because TM is meant to be a simple way to implement lock-free data structures. However, this is of little concern as it alters the program's semantics only when a critical section cannot complete in isolation and is converted to a global (fallback) lock, similar to the communicating critical section example.

```
1  | threadA(){
2  |     begin();
3  |         while(1){};
4  |     end();
5  | }
6  |
7  | threadB(){
8  |     begin();
9  |     end();
10 |     print("Hello");
11 | }
```

Table 4.3: An example to show the difference between purely atomic sections and global lock semantic critical sections from [24]. Truly non-blocking atomic sections would guarantee that "Hello" is eventually printed but global locking semantics do not.

Another semantic difference from TM occurs which causes the synchronization methods to behave more like locks is shown in Figure 4.5. For transactional memory implementations which publicize their writes to non-transactional atomically inserting empty atomic sections does not change the semantics of the example program in that figure. However, empty critical sections can, as a side-effect, cause published writes to be synchronized with memory accesses that are not directly within a critical section where some of the effects may otherwise be unordered. This could be a concern if a program is written to rely on the fact that atomic sections will not make their changes visible until commit time, even to non-transactional code, and do so atomically and is then converted to using lock semantics. The recommended remedy would be to simply not implement application-level synchronization primitives and to access shared data only while the corresponding lock is held or an atomic section is known to be active.

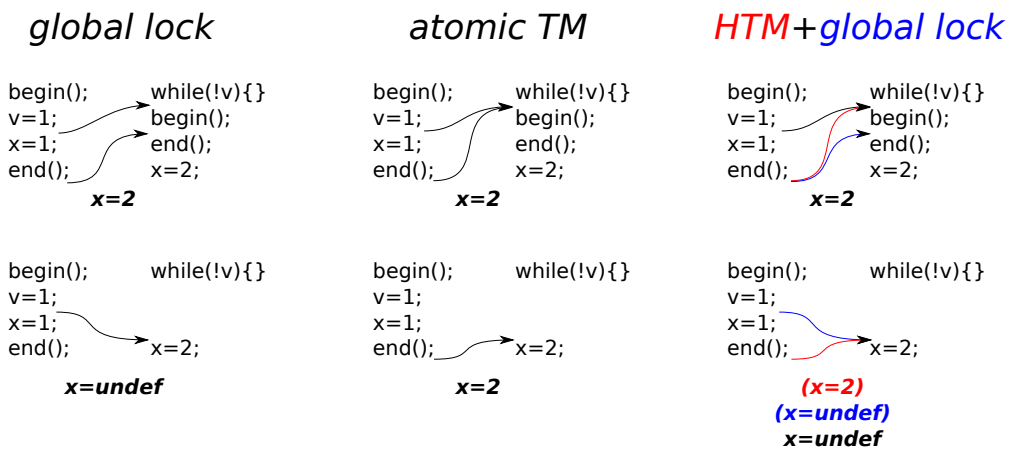


Figure 4.5: Empty Critical Section Ordering: the semantics of HTM with a global lock fallback follow much the same semantics as global locks as seen in this example from [24]. Each arrow indicates a "happens before" constraint except in the HTM+global lock case where either the read or blue arrows are enforced but not both.

4.2.2 High-level Performance Differences

For global lock implementations, since only one thread can truly hold a lock at a time if threads spend a large fraction of their time in critical sections this could negatively impact performance. However, for global fallback HTM it has more slack in performance. For the global futex implementation it is not possible for the sum of the time spent by each thread in the critical section to be longer than the run-length of the entire program

$$T_{program} \geq \sum_{i=0}^{n} G_i \tag{4.1}$$

This relationship is described in Equation 4.1 where $T_{program}$ is the duration of the entire program and G_i is the amount of time spent in the global futex lock critical section by thread i. Even with a global fallback, HTM can exceed this limitation because it can elide locks causing two threads to act as if they both held the same global lock without causing race conditions. Instead, the total time spent in critical sections is bounded by the sum of the of the duration of the threads as in Equation 4.2.

$$\sum_{i=0}^{n} T_i \geq \sum_{i=0}^{n} G_i \tag{4.2}$$

There is no direct benefit to spending a large fraction of a program's lifetime inside critical sections but there is in how and why those variants become enforced. If a program has two threads which both have critical sections and within those critical sections each

thread has high data locality with itself and low data locality with the other thread then, among the global locking implementations, HTM would be preferred because it would avoid unnecessary waiting.

Chapter 5

Experiments

5.1 Experimental Setup

5.1.1 Hardware

As stated in Section 4.1 the validity of the `RDTSC` instruction is dependent on whether or not the timestamp counter is invariant and whether or not all the cores from which `RDTSC` is called are in the same socket. Table 5.1 shows that, indeed, all cores are located in the same socket and additionally in the same, lone NUMA zone. One will also observe that there are 8 CPUs while there are 4 cores; Hyper-threading is turned on which provides 2 hardware threads per core.

In order to show that the timestamp counters in each of the cores are invariant (and thus stable intra-core and synchronized between cores of the same socket) the 8th bit of the EDX register must be set by `CPUID` [10]. Linux provides a utility to dump the raw data from the `CPUID` instruction so it can be directly examined. This output is shown in Table 5.2 and confirms that for each CPU, (including Hyper-threaded ones) their timestamp counters are invariant.

The CPU the experiments are run on is an Intel Core i7-4770 with a nominal clock speed of 3.40GHz, with 16GiB of memory, 8192K L3 cache, 256K L2 cache and 32K each of data and instruction L1 cache. Additionally, the maximum allowed depth of RTM nesting as determined by Table A.4 is 7, which successfully commits in 99.9851% of 1 million trials while a nesting depth of 8 commits in no trials out of 1 million. This nesting depth will be important for a principle to be introduced later.

```
$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   60
Stepping:                3
CPU MHz:                 3401.000
BogoMIPS:                6798.27
Virtualization:          VT-x
L1d cache:               32K
L1i cache:               32K
L2 cache:                256K
L3 cache:                8192K
NUMA node0 CPU(s):       0-7
```

Figure 5.1: Contents of lscpu: 1 socket, 1 NUMA zone, 4 cores, Hyper-threading enabled

5.1.2 Software

The version of the GNU C Library used was version 2.19 and based on (truncated) commit hash value 2623655 which can be found in [25]. Each instance of the GNU C Library was compiled with the `-O2` code optimization option. These experiments were run in Ubuntu 14.04 LTS, Linux version 3.13.0-63-generic, and the relevant source code compiled with gcc/g++ version 4.9.2.

5.2 Data Reporting

To aid in understanding the data represented in the graphs in this thesis their descriptions are provided below.

Region-of-Interest Duration: each bar in these graphs represents the average time it takes, in seconds, for the given program to execute a given "region-of-interest" which is in-

```
$ cpuid -r | grep -E "0x80000007|CPU" | cut -f 1,2,9 -d " "
CPU 0:
  edx=0x00000100
CPU 1:
  edx=0x00000100
CPU 2:
  edx=0x00000100
CPU 3:
  edx=0x00000100
CPU 4:
  edx=0x00000100
CPU 5:
  edx=0x00000100
CPU 6:
  edx=0x00000100
CPU 7:
  edx=0x00000100
```

Figure 5.2: Contents of cpuid: according to [10] when cpuid loads address 0x80000007 into register EDX bit 8 being set indicates that the timestamp counter is invariant.

teresting for analysis for a given number of threads. The duration of the region of interest as reported for memcached is given by memslap and for PARSEC and SPLASH-2x applications this is reported by the "hooks" library. Note that the region of interest intentionally avoids one-time costs like program startup.

Region-of-Interest Duration Normalized: this is the same information as reported by "Region-of-Interest Duration" but each data point is normalized so that it is expressed as a speedup over the fine-grained futex locking implementation of the same number of threads.

Ratio Transaction Commits – Starts: each bar indicates what fraction of started transactions successfully commit. This is effectively the ratio of *outermost* `xend` invocations over *outermost* `xbegin` invocations. Since the outermost invocations can be counted without risk of being rolled back and are counted per-thread and collected after all other threads have stopped there is no risk of losing updates.

Ratio Ticks Aborted – Region-of-Interest: each bar indicates the ratio of the number of ticks counted by `RDTSC` around transactions (again, outside outermost `xbegin` and `xend` calls only) over the number of ticks for the region of interest. Care is needed when considering these values. One thread, in theory, could end up with a value slightly higher than 1 since it may perform a transaction in startup code. Also, if there are N threads it is possible that they each perform transactions for more than $\frac{1}{N}$ of the duration of the region-of-interest

leading to a ratio greater than 1 but not much greater than N.

Ratio Tick Transaction – Region-of-Interest: these graphs are similar to "Ratio Ticks Aborted – Region-of-Interest" but instead report timing for both successful and aborted transactions.

Configurations: All programs evaluated in this work use the following synchronization implementations: global futex lock (*futex-global*), fine-grained futex locks (*futex-fine*), HTM with fine-grained fallback locks (*htm-fine-ta*) and HTM with a global fallback lock (*htm-global-ta*). Additionally, memcached is also evaluated with the following synchronization implementations: HTM with fine-grained fallback locks without a trylock abort (*htm-fine-nta*) and HTM with a global fallback lock without a trylock abort (*htm-global-nta*).

5.3 memcached

memcached is an in-memory object cache which can be used in distributed configurations and is meant to speed up web applications by moving some work the database system would have to do into an object cache and is used in very large websites [26]. It makes use of some real-world or at least non-trivial synchronization techniques which make it an interesting candidate for evaluating different synchronization implementations.

memcached version 1.4.24 was used for this evaluation and was tested using memslap from libmemcached-1.0.18.

5.3.1 Notable Synchronization Methods

Nested Trylocks

There are instances in which `pthread_mutex_trylock` is called while locks are already held which, as previously stated in Section 3.2, will cause a hardware transaction to abort if one is active. However, removing this explicit abort does not necessarily cause an actual error and may be removed in some correctly written applications. This overhead can be examined by compiling out the explicit abort. Leaving the abort in can incur a significant overhead if not handled correctly which will be examined in Section 5.3.2. For this reason memcached will be examined using a global futex lock, fine-grained futex locks, HTM with fine-grained fallback locks, HTM with a global fallback lock, HTM with fine-grained fallback locks without a trylock abort and HTM with a global fallback lock without a trylock abort. Strictly speaking, trylock aborts should not occur for the HTM global fallback version of the library as nested transactions are not possible and so a version of that library without

trylock aborts should be unnecessary, however it is included for completeness. No trylock aborts were incurred with either of the HTM global fallback implementations.

Condition Variables

Unlike C++'s TM for GCC which needs to instrument functions and make individual functions safe for its transaction model ([7, 8]) hardware transactions are more flexible. With condition variables a mutex needs to be acquired by the thread before waiting on a condition variable which atomically releases the mutex and blocks on the condition variable which may cause the thread to sleep. The condition variable wait does not need to follow special rules or need particularly close integration with the application to work correctly. Although, if the thread goes to sleep blocking on the condition variable the transaction will be aborted but since the thread which incurs the abort was about to go to sleep this is unlikely to negatively affect performance anyway.

Hanging Atomic Sections

In some cases a thread will acquire a mutex lock and hold it indefinitely while another thread attempts to acquire the same lock with the next function immediately being a lock release. The original behavior would cause the first thread to hold the lock while the second thread would go to sleep for extended periods until the first thread releases the lock. Any number of additional threads could go to sleep in the same way as the second thread.

However, with hardware transactions it is possible that the first thread elides the lock while the second thread comes upon the lock function call and then runs through the lock and unlock functions without waiting using a hardware transaction because it sees that the mutex is not acquired. This can be an issue if the original program ensured an ordering such that the first thread is guaranteed to acquire the lock before the second thread attempts to acquire it. A blocking realization of such an ordering constraint would end up behaving like condition synchronization and end up with the intended consequence of the empty critical section anyway. However, if one was so compelled to write such code, or a non-blocking realization of the constraint which "remembers" having passed through the hanging atomic section, it may break the original program's semantics on conversion to HTM.

5.3.2 Lock Cascade Failure

In some situations it is possible that a thread will acquire multiple locks (and possibly do other work in-between acquisitions) but will always, or almost always, be aborted causing the thread to re-elide all but a certain bounded number of locks. This can cause the amount of work actually performed to grow unexpectedly. The version of glibc used for this evaluation

has an "adapt count" which causes the library to skip attempts at lock elision for a given lock, without regard for the thread, for a certain number of acquisitions when a thread incurs certain kinds of aborts. In Table A.1 the pointer to `adapt_count` is specific to the mutex lock but not to the thread which incurs the abort. Since the abort does not reliably relay the conditions which led to the abort or the locks that were elided when the abort occurred a thread which attempts to elide multiple locks simultaneously and then aborts only knows to fallback for the outermost lock in glibc. Ideally the thread would be aware of all the locks which were elided when the abort occurred so it would not attempt to re-elide until after reaching the point at which it last aborted.

As a result it is possible that a thread acquires N locks (L_{N-1} being the first, outermost lock and L_0 being the latest, innermost lock) doing some amount of work W_i between acquiring/eliding L_i and the point at which the thread aborts which takes a quadratic amount of processor effort to perform a linear amount of work (assuming all segments of work take about the same amount of time). On the first pass the thread does the work W_{N-1} and then aborts, acquiring L_{N-1} normally and then performs W_{N-1} again, aborts, acquires L_{N-2} normally and performs W_{N-2}. This program control flow is depicted in Figure 5.3. W_i can be re-written as the work done between acquisition of L_i and L_{i-1} (or between L_0 and the abort for W_0), P_i, as in Equation 5.1. The total work that has to be done in this situation can be expressed as in Equation 5.2. Note that, for a given i, the amount of work done in P_i is assumed to be constant every time its encountered.

$$W_i = \sum_{k=0}^{i+1} P_k \tag{5.1}$$

$$W_{Total} = \sum_{i=0}^{N} (i+2) * P_{N-i-1} \tag{5.2}$$

There will also be a minimum segment of work, P_{min}, and a maximum segment of work, P_{max} (which need not be distinct). Knowing this, the upper and lower bounds of W_{Total} can be found as described in Equation 5.3.

$$\frac{P_{min}(N^2+3N)}{2} \leq W_{Total} \leq \frac{P_{max}(N^2+3N)}{2} \tag{5.3}$$

This behavior is relevant to memcached in particular because it acquires more than one lock at a time and then performs a trylock which always causes an abort in the original glibc. Intel's implementation of hardware transactional memory has a counter which is inaccessible from software which tracks the depth of transaction nesting for a given core [10]. Each core is also restricted in how many transactions deep it can nest, this value is `MAX_RTM_NEST_COUNT` [10] and for the first implementations was set to a value of 7 [11]. For implementations which still follow this rule the additional constraint can be written as

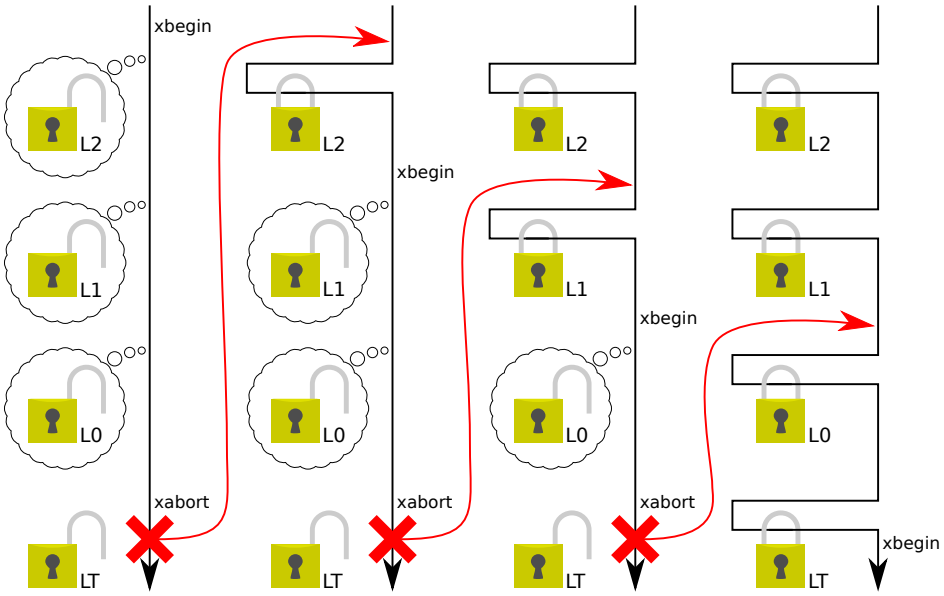


Figure 5.3: Quadratic Elision/Acquire Lock Cascade Failure: lock 2 through 0 perform normal locking or normal elision with LT being trylocked. L2 will elide once before being acquired; L1, twice; L0, three times. This results in a quadratic amount of work needing to be done before advancing instead of a linear amount. LT itself does not add to the length of the cascade and is here only to illustrate that a trylock can reliably cause an abort. This same behavior can occur for segments of code for which repetitive aborts are possible.

in Equation 5.4.

$$\frac{P_{min}(N^2+3N)}{2} \leq W_{Total} \leq \frac{P_{max}(N^2+3N)}{2} \leq 35P_{max} \tag{5.4}$$

`MAX_RTM_NEST_COUNT` can be detected with very high probability using a small program as shown in code Table A.4. While any number of runs cannot determine this value with absolute certainty, the likelihood of a spurious event which causes an abort is very low for small nesting depths.

One should note that the quadratic nature of this effect disappears with a recursive global lock as no would-be nested transactions are issued, if the global lock is locked (not elided) then no inner transactions can start, the counter for the global lock is just incremented.

5.3.3 Results

All data points presented in this section are generated using a linear average of ten runs of memcached of the same configuration. Additionally, memcached is pinned to four Hyperthread cores all on the same two real cores and memslap is pinned to the other four Hyperthread cores. As a result when using more than four threads the CPUs on which memcached

and memslap run are overcommitted.

One will note that in Figure 5.5(a) the global futex implementation, surprisingly, actually performs virtually equal to or better than the fine-grained futex implementation in some cases (as it has a speedup greater than 1 in some cases). This would suggest that memcached, at least under this load, is actually too fine-grained for the period of time it spends in locked critical sections. Instead of incurring a slowdown due to blocking waits introduced by coarsened locking a speedup is achieved because once the global lock is acquired by a thread it does not need to execute the full locking function again until after the global lock is completely released.

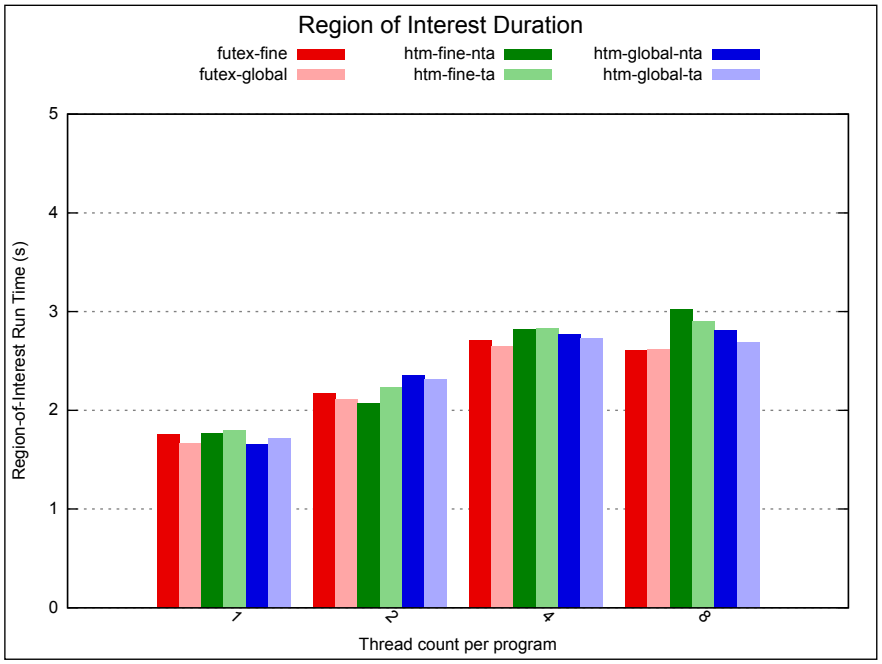
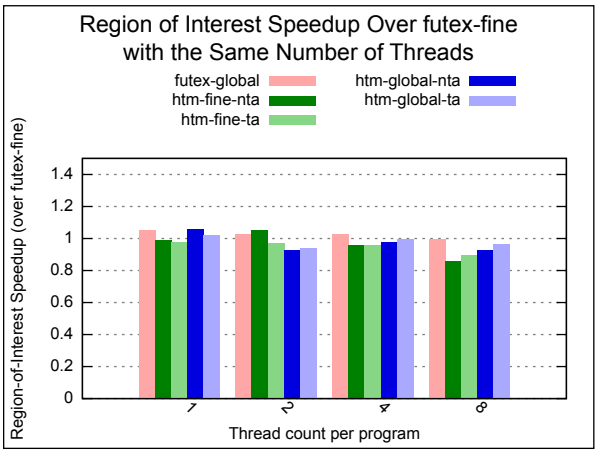


Figure 5.4: memcached: Region-of-Interest Duration

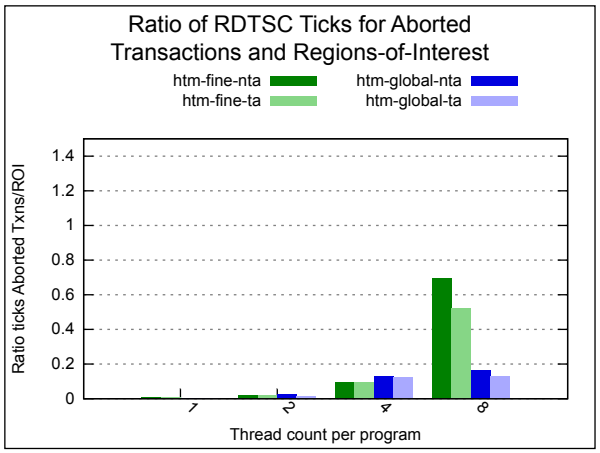
While the data gathering is not invasive enough to unambiguously determine if any significant changes to the amount of time spent in aborted transactions are due to lock cascade failure Figure 5.5(b) does show a significant difference in the amount of time spent on aborted transactions for fine-grained fallbacks. Interestingly, the trylock abortive implementations show noticeably less time spent in aborted transactions than their non-trylock abortive counterparts despite having a closely matching commit rate (Figure 5.5(c)).

The time spent on aborted transactions and transactions overall is much higher for memcached than almost all of the programs in PARSEC and SPLASH-2x. This suggests that there is probably a coverage gap in the union of PARSEC+SPLASH-2x consisting of pro-

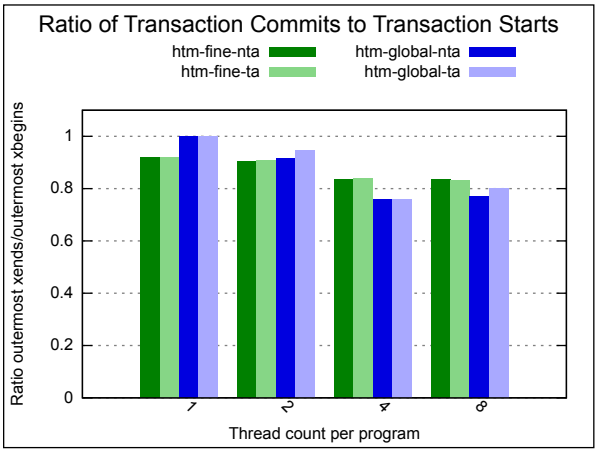
grams with a high ratio of time spent in transactions to the overall program duration. Despite this, the HTM global fallback version performs on par with the original fine-grained futex version.



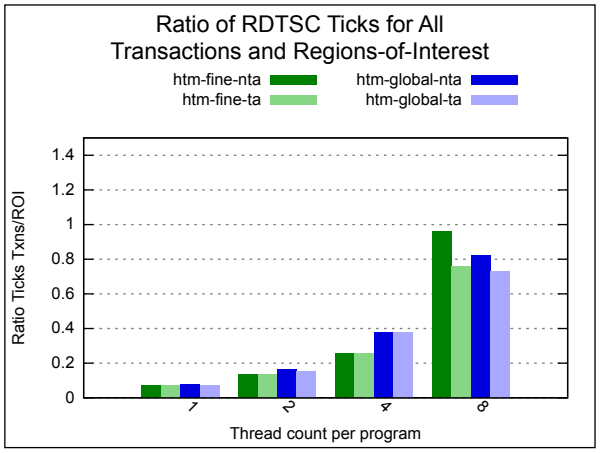
(a) memcached: Region-of-Interest Duration Normalized — same as Figure 5.4 normalized for fine-grain futex for each number of threads.



(b) memcached: Ratio Ticks Aborted – Region-of-Interest



(c) memcached: Ratio Transaction Commits – Starts



(d) memcached: Ratio Tick Transaction – Region-of-Interest

Figure 5.5: memcached: Quantitative Results

5.4 PARSEC and SPLASH-2x

PARSEC is a benchmark suite of programs geared towards multithreading and emerging workloads but explicitly away from High-Performance Computing niches [13]. The creator(s) of the PARSEC suite describe "emerging applications" as being a new software type

that might arise because of progress in its field of research, the way users interact with the software or the computational burden it puts on the computer system [13].

SPLASH-2x is different as it is based on a benchmark suite of multithreaded programs released in the early 1990's, SPLASH-2, when computers that could make use of parallel programs were uncommon [13, 27]. As a result it targets high-performance computing [13, 27].

For this evaluation only a subset of the programs packaged with the suites will be considered. The actual download comes with PARSEC 3.0, SPLASH-2 and SPLASH-2x but, as SPLASH-2x is an update to SPLASH-2 based on input set scaling, SPLASH-2 will not be considered in this evaluation [27]. However, SPLASH-2x will be used as its authors packaged it with PARSEC with the understanding that it has been found to complement the characteristics which PARSEC exercises [27]. PARSEC comes with 16 programs, ten of which are classified as applications, three as kernels and three as network applications. Since the networked applications also have a non-networked version the networked version will not be examined. Five of the ten applications and all three kernels will be examined. The programs which will be evaluated are shown in Table 5.1.

5.4.1 Results

All data points presented in this section are generated using a linear average of three runs of the program with the same configuration. Unlike memcached there were no trylock aborts reported for any run of any of the PARSEC or SPLASH-2x tests so the versions of glibc without the explicit trylock abort were not used.

Figure 5.6 shows the absolute run time (in seconds) of each of the programs' region-of-interest. Figure 5.7 shows the run time for each configuration normalized such that they are expressed as a ratio of the run time of the given configuration at the given number of threads over the run time of the fine-grained futex configuration at the given number of threads. The data for Figure 5.7 is simply a different representation of the data in Figure 5.6. If the fine-grained futex data points were displayed in Figure 5.7 they would be, by definition, 1.

Note that the data for SPLASH-2x's volrend program is invalid for single-threaded operation since the program duration it reports in the original implementation is negative and must be incorrect.

One should note that the bars in Figure 5.6 are relatively flat for a given number of threads, meaning that in many cases the actual implementation of synchronization is unimportant. There are however some exceptions.

Suite	Class	Program	Domain [13]
PARSEC	apps	blackscholes	Financial Analysis
		bodytrack	Computer Vision
		facesim	Animation
		ferret	Similarity Search
		fluidanimate	Animation
	kernels	canneal	Engineering
		dedup	Enterprise Storage
		streamcluster	Data Mining
SPLASH-2x	apps	barnes	High-Performance Computing
		fmm	High-Performance Computing
		ocean_cp	High-Performance Computing
		ocean_ncp	High-Performance Computing
		radiosity	Graphics
		raytrace	Graphics
		volrend	Graphics
		water_nsquared	High-Performance Computing
		water_spatial	High-Performance Computing
	kernels	cholesky	High-Performance Computing
		fft	Signal Processing
		lu_cb	High-Performance Computing
		lu_ncb	High-Performance Computing
		radix	General

Table 5.1: PARSEC and SPLASH-2x Programs: programs used for this evaluation.

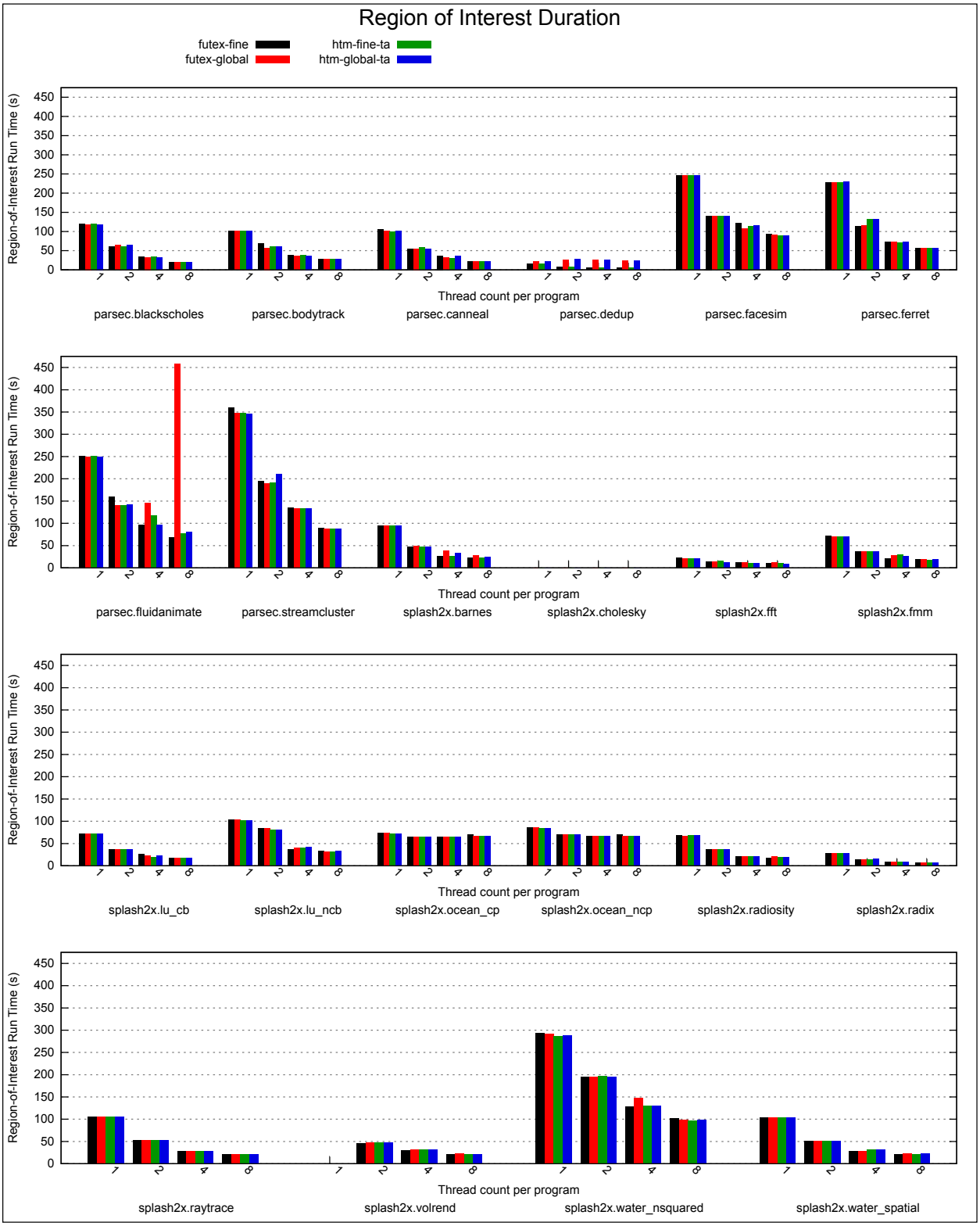


Figure 5.6: PARSEC and SPLASH-2x: Region-of-Interest Duration

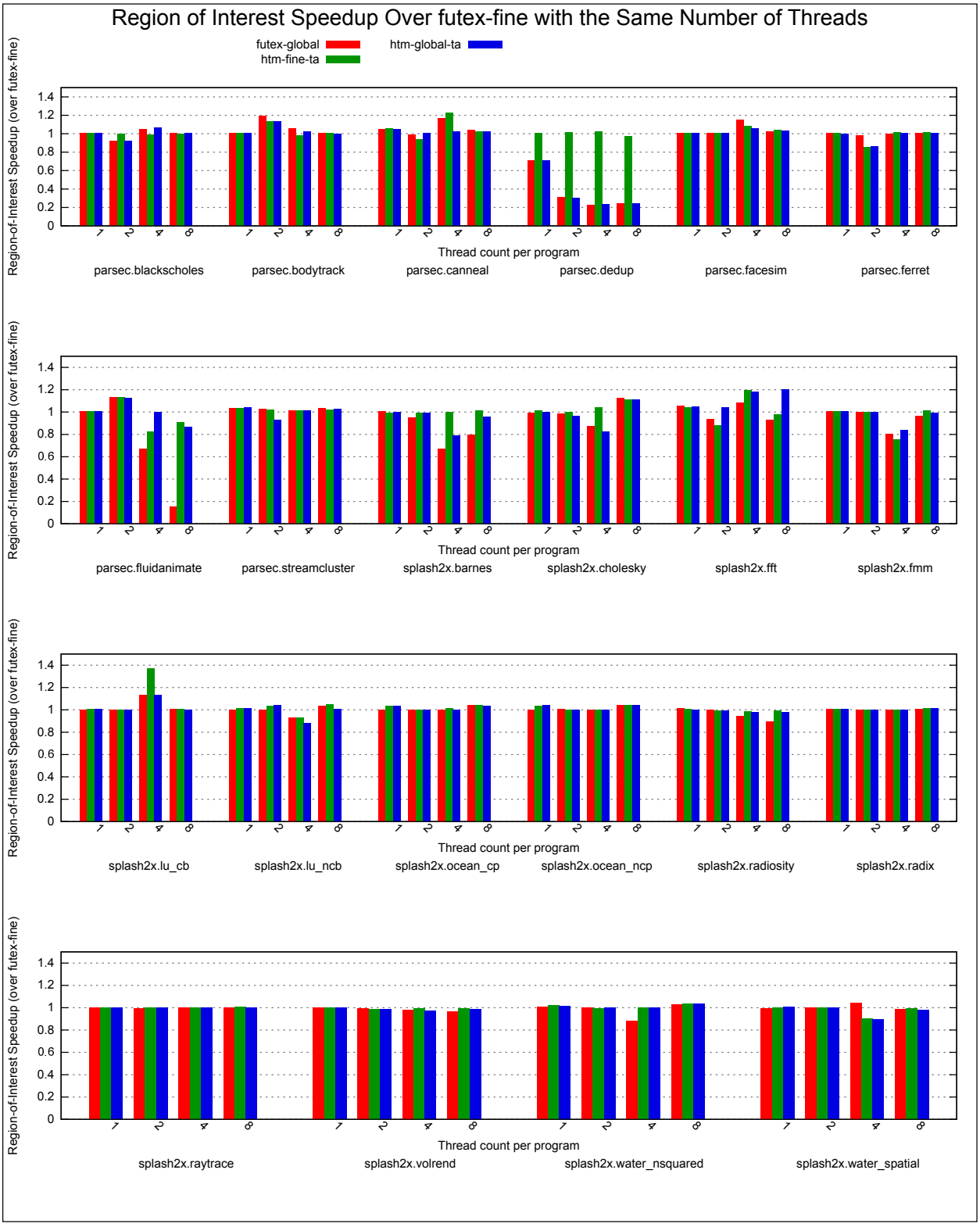


Figure 5.7: PARSEC and SPLASH-2x: Region-of-Interest Duration Normalized

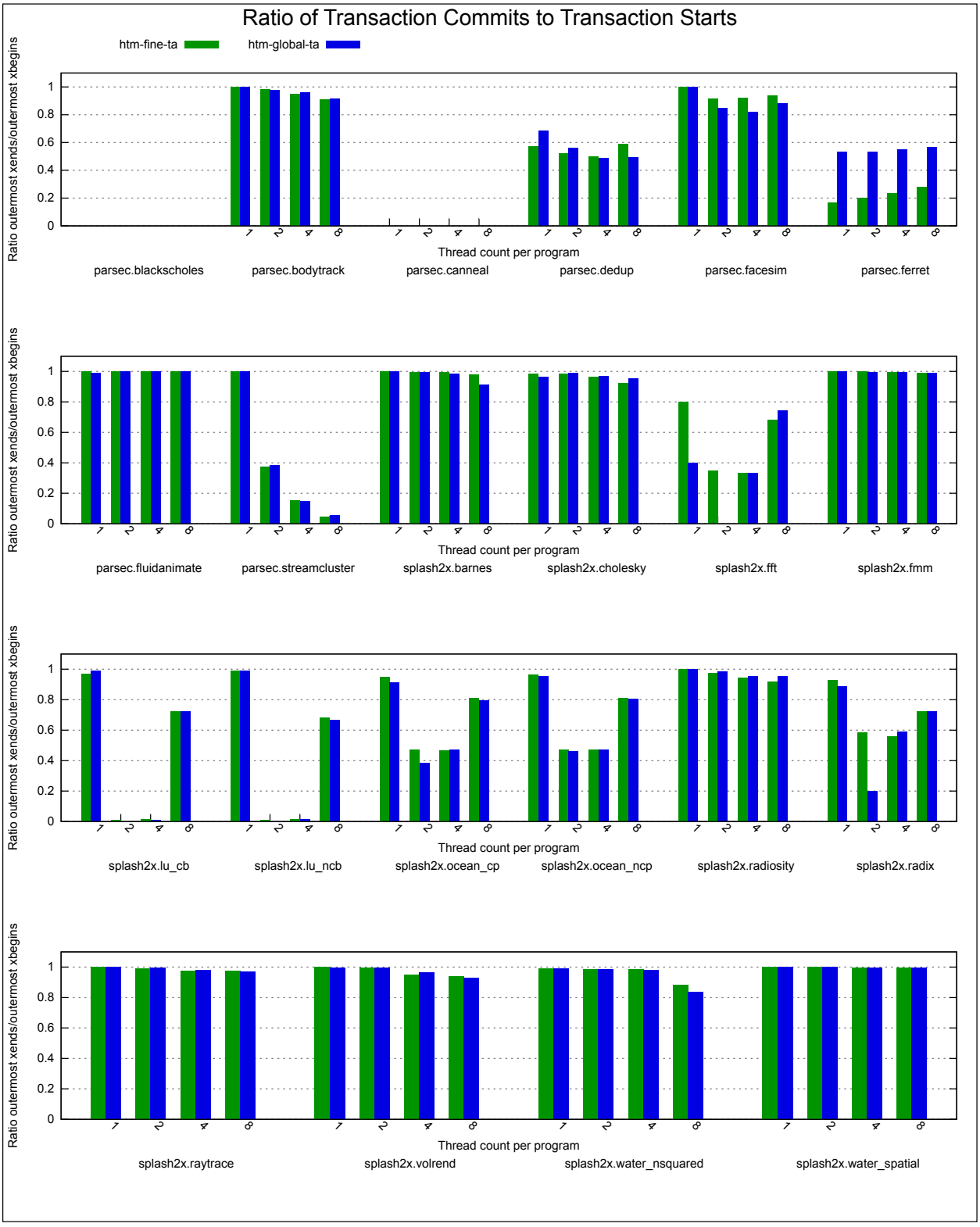


Figure 5.8: PARSEC and SPLASH-2x: Ratio Transaction Commits – Starts

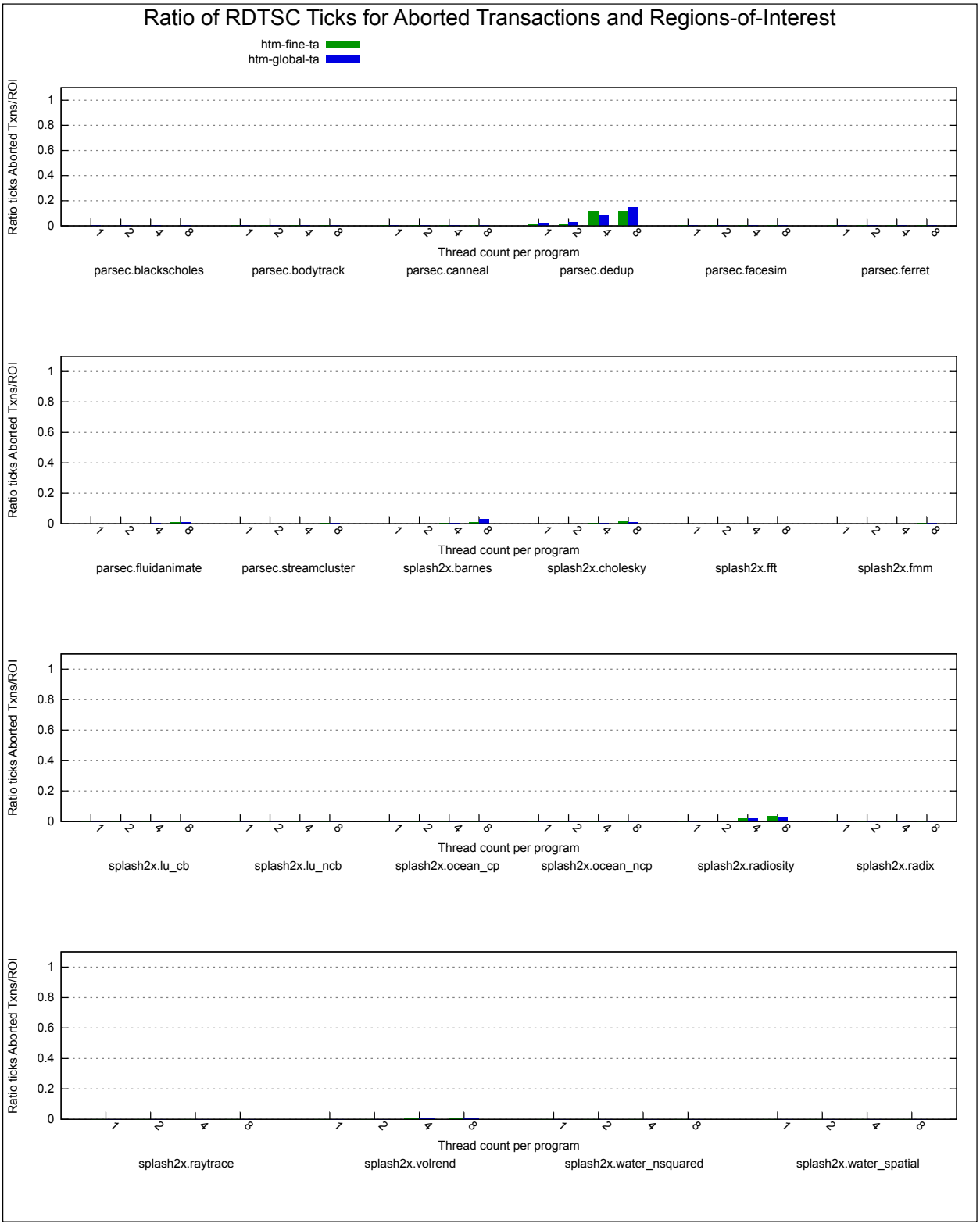


Figure 5.9: PARSEC and SPLASH-2x: Ratio Ticks Aborted – Region-of-Interest

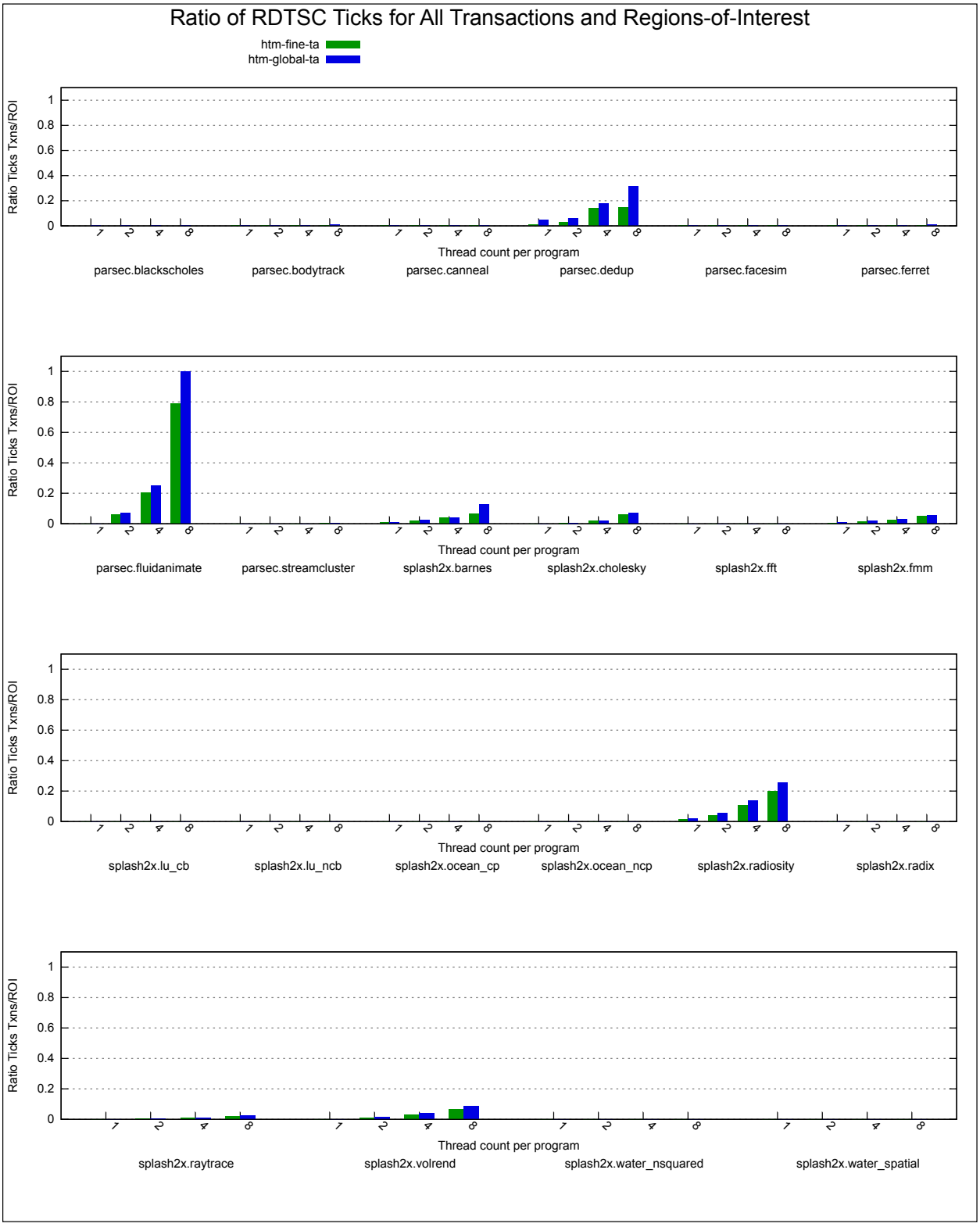


Figure 5.10: PARSEC and SPLASH-2x: Ratio Tick Transaction – Region-of-Interest

PARSEC:dedup

A dramatic exception is PARSEC's dedup which incurs almost a 5× slowdown when using 8 threads and a global lock but maintains near parity with the fine-grained futex implementation when only hardware transactions are introduced. Figure 5.9 shows that the program spends less than 20% of its time (across all threads with respect to the duration of the region-of-interest) in aborted transactions (in either fine-grained or global fallback HTM) despite being the one that spends the largest percentage of its time on aborted transactions (when running using 8 threads). This suggests that the overhead primarily comes from blocking synchronization using global locks. This agrees with the fact that the global futex version has a nearly equivalent slowdown and that the fine-grained fallback version has almost no slowdown.

PASREC:fluidanimate

Another dramatic exception is PARSEC's fluidanimate. Like dedup the global futex implementation has a high overhead and a more than 5× slowdown. However, unlike dedup the global fallback HTM version maintains most of its performance, slowing down by no more than 20%. Figure 5.8 shows that fluidanimate very rarely ended up aborting a transaction and Figure 5.9 shows that even when it did the cost for doing so was extremely low. One should note that the proportion of the program spent on aborted transactions is very small in comparison to the slowdown the global fallback HTM incurred, meaning that this remaining extra time was also likely spent on synchronization of futexes.

Unlike most of the other programs, fluidanimate spends a fairly appreciable amount of time doing work inside transactions as shown in Figure 5.10 (but note that the value of that ratio is only strongly bound by the number of threads; the fact that for eight threads using global lock fallback HTM the value is near 1 is coincidental). This, along with the fact that slowdowns are rare among other applications using the global futex implementation, indicates that these programs spend little time in synchronized sections as a programmer would aim to do when parallelizing code.

General Trends

Even for programs with very high success committing transactions (PARSEC: bodytrack, facesim, fluidanimate; SPLASH-2x: barnes, cholesky, fmm, radiosity, raytrace, volrend, water_nsquared, water_spatial) all but a few (PARSEC: fluidanimate; SPLASH-2x: barnes, radiosity) spend an appreciable amount of time actually performing transactions.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work has shown that for many multithreaded programs the distinction between fine-grained and coarse-grained locking approaches is not largely critical to an application's performance. Additionally, this work has shown that in some cases hardware transactional memory can alleviate the performance penalty imposed when switching to a coarse-grained (global in this case) locking scheme.

This work has also covered situations in which global locks and Intel's Restricted Transactional Memory may lead to pathologies which affect a program's ability to operate well or at all under either synchronization method. In cases where global locks cannot support the behavior of the program certain assumptions about how threads communicate with each other have to be rolled back in order to regain support. For programmers considering whether or not to use hardware transactions in their new application assumptions which explicitly break isolation properties do not necessarily need to be introduced. For maintainers that are instead supporting legacy applications the program should be analyzed for such communication patterns so that deadlocks can be ruled out.

If memcached, PARSEC and SPLASH-2x, together, are representative samples of the space of multithreaded programs the conclusion must be that in the majority of cases a locking structure more complex than a single global lock incurs little performance penalty and that in the cases with appreciable overhead hardware transactions can typically recover much of the performance. This leads to a recommendation. Programmers writing multithreaded software should assume, if there is no evidence to the contrary, that under conditions similar to those given in this evaluation the total duration of critical sections will be small compared to the duration of the program overall. As a consequence of this coarse-grained locking is likely to be more appropriate than fine-grained locking as fewer restraints need to be followed

to ensure that no potential deadlocks are introduced when writing a new application. However, there exists situations where the critical section is an appreciable duration compared to that of the program overall. Some of these situations can recover much of the performance of the fine-grained locking version by using hardware transactional memory to elide coarser locks. These findings indicate that fine-grained locking should be considered a last resort to speed up program execution and should instead favor using coarse (even global) locks and hardware transactional memory, both of which have much more straightforward methods of reasoning about their correctness.

6.2 Future Work

Fundamentally speaking, hardware transactions do not need to suffer from lock cascade failure. If a condition which would consistently cause an abort could be detected from within a transaction then a small amount of information could be passed back through an abort status code so that the locking implementation knows when it is performance-safe to begin eliding again. Assuming that the the abort code does not hold enough information to perfectly represent the relevant state the locking implementation would need to satisfice and start eliding after having run a certain duration or doing a certain amount of work without seeing the offending condition.

POSIX defines a spinlock synchronization primitive which is similar to the futex implementation of mutexes except that the thread which blocks on a spinlock does not go to sleep but instead busy waits on that lock [20]. This behavior is rather similar to what hardware transactional memory does except that it has the opportunity to do less waiting. However, spinlocks by their nature have very low latency because they do not put threads to sleep when waiting for the lock to be released. A series of hardware transactions could potentially take noticeably longer to execute than the sum of the durations of the critical sections if they are attempted more than once. This evaluation considers "performance" to only consist of how long a program takes to execute its region of interest but for realtime programs faster average throughput is not necessarily as important as lowering maximum latency.

Bibliography

[1] H. Sutter. (2009, August). *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software* [Online]. Available: `http://www.gotw.ca/publications/concurrency-ddj.htm`

[2] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, April, 1965. Available: `www.computerhistory.org/semiconductor/assets/media/classic-papers-pdfs/Moore_1965_Article.pdf`

[3] J. Manson; B. Goetz, (2004, February). *JSR 133 (Java Memory Model) FAQ* [Online]. Available: `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html`

[4] H. Boehm; S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in Programming Language Design and Implementation, Tuscon, AZ, 2008, pp. 68-78. Available: `http://rsim.cs.illinois.edu/Pubs/08PLDI.pdf`

[5] M. Herlihy; J. Eliot; B. Moss, "*Transactional Memory: Architectural Support For Lock-free Data Structures*," Proceedings of the 20th Annual International Symposium on Computer Architecture, pp.289-300, May, 1993. doi: 10.1109/ISCA.1993.698569 Available: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=698569&isnumber=7905`

[6] *Draft Specification of Transactional Language Constructs for C++*, version 1.1, 2012 February 3. Available: `https://sites.google.com/site/tmforcplusplus`

[7] W. Ruan; T. Vyas; Y. Liu; M. Spear, "*Transactionalizing Legacy Code: an Experience Report Using GCC and Memcached*," Proc. of the 19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, p399-412, March, 2014. doi: 10.1145/2541940.2541960 Available: `http://dl.acm.org/citation.cfm?id=2541960`

[8] C. Wang; Y. Liu; M. Spear, "*Transaction-Friendly Condition Variables*," Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques p198-p207, June, 2014. doi: 10.1145/2612669.2612681 Available: `http://dl.acm.org/citation.cfm?id=2612681`

[9] N. Shavit; D. Touitou, "Software Transactional Memory," in Principles of Distributed Computing, Ottawa., ON, 1995, pp. 2014-213. Available: `http://dl.acm.org/citation.cfm?id=224987`

[10] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Co., Santa Clara, CA, June 2015, Available: `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`

[11] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Co., Santa Clara, CA, September 2014, Available: `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`

[12] M. Scott, "Transactional Memory Today," *ACM SIGACT News*, vol. 46, no. 2, pp. 96-104, June, 2015. Available: `http://dl.acm.org/citation.cfm?id=2789166`

[13] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. Dissertation, Comp. Sci., Princeton Univ., Elizabeth, NJ, 2011. Available: `http://parsec.cs.princeton.edu/publications/bienia11benchmarking.pdf`

[14] A. Kleen. (2014, March 26). *TSX anti patterns in lock elision code* [Online]. Available: `https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code`

[15] I. Calciu; T. Shpeisman; G. Pokam; M. Herlihy, "Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory," in ACM SIGPLAN Workshop on Transactional Computing, Salt Lake City, UT, 2014

[16] D. Dice; M. Herlihy; D. Lea; Y. Lev; V. Luchangco; W. Mesard; M. Moir; K. Moore; D. Nussbaum, "Applications of the Adaptive Transactional Memory Test Platform," TRANSACT, Salt Lake City, UT., 2008, Available: `http://wwwa.unine.ch/transact08/papers/Dice-Applications.pdf`

[17] T. Riegel. (2012, Feb. 6). *Transactional Memory in GCC* [Online]. Available: `https://gcc.gnu.org/wiki/TransactionalMemory`

[18] GNU Project. (2015, February 6). *The GNU C Library (glibc)* [Online]. Available: `https://www.gnu.org/software/libc`

[19] GNU Project. (2013, July 2). *Add the low level infrastructure for pthreads lock elision with TSX* [Online]. Available: `https://sourceware.org/git/?p=glibc.git;a=commit;h=1cdbe579482c07e9f4bb3baa4864da2d3e7eb837`

[20] *The Open Group Base Specifications Issue 7*, IEEE Std 1003.1, 2013. Available: `http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_lock.html`, `http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_spin_destroy.html`

[21] E. Mikulic. (2014, March 16). *Benchmarking* [Online]. Available: `https://unix4lyfe.org/benchmarking`

[22] L. Brown. (2015, May 31). *Linux Kernel Mailing List* [Online]. Available e-mail: (len.brown@intel.com) Message: ([PATCH 1/1] x86 TSC: set X86_FEATURE_TSC_RELIABLE, per CPUID) Available: `https://lkml.org/lkml/2015/5/31/14`

[23] D. Magenheimer. (2010, May 15). *Linux Kernel Mailing List* [Online]. Available e-mail: (dan.magenheimer@oracle.com) Message: (RE: [PATCH] x86: Export tsc related information in sysfs) Available: `https://lkml.org/lkml/2010/5/15/145`

[24] H. Boehm, "Transactional Memory Should Be an Implementation Technique, Not a Programming Interface," in Hot Topics in Parallelism, Berkeley, CA, 2009, pp. 1-6. Available: `https://www.usenix.org/legacy/event/hotpar09/tech/full_papers/boehm/boehm.pdf`

[25] GNU Project. (2015, August 16). *GNU C Library master sources* [Online]. Available: `https://sourceware.org/git/?p=glibc.git;a=summary`

Using commit 262365511f6fafa4e15090fe16a295c03c6f6c5e

[26] Dormando. *memcached - a distributed memory object caching system* [Online]. Available: `http://www.memcached.org`

[27] PARSEC Group, "A Memo on Exploration of SPLASH-2 Input Sets," Princeton Univ., Elizabeth, NJ, 2011. Available: `http://parsec.cs.princeton.edu/doc/memo-splash2x-input.pdf`

Appendix A

Source Examples

```
1  int
2  __lll_lock_elision (int *futex, short *adapt_count, EXTRAARG int private)
3  {
4      if (*adapt_count <= 0)
5          {
6              unsigned status;
7              int try_xbegin;
8
9              for (try_xbegin = aconf.retry_try_xbegin;
10                 try_xbegin > 0;
11                 try_xbegin--)
12             {
13               /* Code to protect against the lemming effect. */
14               while(*((volatile int*)(futex)) != 0)
15                   {
16                       __asm__ ( "pause;" );
17                   }
18
19               if ((status = _xbegin()) == _XBEGIN_STARTED)
20                   {
21                       if (*futex == 0)
22                           return 0;
23
24                       /* Lock was busy. Fall back to normal locking.
25                            Could also _xend here but xabort with 0xff code
26                            is more visible in the profiler. */
27                       _xabort (_ABORT_LOCK_BUSY);
28                   }
```

```
29
30                  if (!(status & _XABORT_RETRY))
31                      {
32                          /* Internal abort. There is no chance for retry.
33                               Use the normal locking and next time use lock.
34                               Be careful to avoid writing to the lock. */
35                          /*else*/ if (*adapt_count != aconf.skip_lock_internal_abort)
36                              *adapt_count = aconf.skip_lock_internal_abort;
37                          break;
38                      }
39                  }
40          }
41      else
42          {
43              /* Use a normal lock until the threshold counter runs out.
44                  Lost updates possible. */
45              (*adapt_count)--;
46          }
47
48
49
50      /* Use a normal lock here. */
51    return LLL_LOCK ((*futex), private);
52  }
```

Table A.1: Elision Locking: statistics gathering code is removed from this excerpt. `aconf.skip_lock_internal_abort` is a global configuration value set to 3 which causes the thread to acquire 3 (or more) locks normally before trying elision again if requested. `aconf.retry_try_xbegin` is similar, indicating that the thread may attempt to take the *outermost* lock (up to) 3 times before falling back. `aconf` is a global configuration structure variable. Note that the lemming effect code (lines 14-17), which only appears for locking, was not present in the original source code.

```
1   int
2   __lll_trylock_elision (int *futex, short *adapt_count)
3   {
4       int abort_time_set = 0;
5
6       #ifndef NOTRYABORT
7       /* Implement POSIX semantics by forbiding nesting
```

```
 8           trylock. Sorry. After the abort the code is re-executed
 9           non transactional and if the lock was already locked
10           return an error. */
11       _xabort (_ABORT_NESTED_TRYLOCK);
12       #endif
13
14       /* Only try a transaction if it's worth it. */
15       if (*adapt_count <= 0)
16           {
17               unsigned status;
18
19               if ((status = _xbegin()) == _XBEGIN_STARTED)
20                   {
21                     if (*futex == 0)
22                         return 0;
23
24                     /* Lock was busy. Fall back to normal locking.
25                           Could also _xend here but xabort with 0xff code
26                           is more visible in the profiler. */
27                     _xabort (_ABORT_LOCK_BUSY);
28                   }
29
30               if (!(status & _XABORT_RETRY))
31                   {
32                     /* Internal abort. No chance for retry. For future
33                           locks don't try speculation for some time. */
34                     if (*adapt_count != aconf.skip_trylock_internal_abort)
35                         *adapt_count = aconf.skip_trylock_internal_abort;
36                   }
37               /* Could do some retries here. */
38           }
39       else
40           {
41               /* Lost updates are possible, but harmless. */
42               (*adapt_count)--;
43           }
44
45       return lll_trylock (*futex);
46   }
```

Table A.2: Elision Trylocking: the same notes as Table A.1 apply here.

```
1  int
2  __lll_unlock_elision(int *lock, int private)
3  {
4      /* When the lock was free we're in a transaction.
5            When you crash here you unlocked a free lock. */
6      if (*lock == 0)
7          {
8              _xend();
9          }
10     else
11         {
12             lll_unlock ((*lock), private);
13         }
14     return 0;
15 }
```

Table A.3: Elision Unlock: the same notes as Table A.1 apply here.

```
1  # include <stdio.h>
2  # include "hle.h" // defines _xbegin, _xend and _xtest
3
4  static const int NEST_RUN = 1000000;
5  static const int NEST_TRY = 10;
6
7  int main(){
8          int nest_hist[NEST_TRY];
9
10         for(int i=0; i<NEST_TRY; ++i){
11                 nest_hist[i] = 0;
12         }
13
14         for(int k=0; k<NEST_RUN; ++k){
15                 for(int i=1; i<=NEST_TRY; ++i){
16                         // enter transactions
17                         bool success = true;
18                         for(int j=0; j<i; ++j){
19                                 if(_xbegin() != _XBEGIN_STARTED){
```

```
20                                      success = false;
21                                      break;
22                              }
23                      }
24
25                      if(success){
26                              // exit transactions
27                              for(int j=0; j<i; ++j){
28                                      _xend();
29                              }
30                      }
31                      else{
32                              break;
33                      }
34
35                      // mark the nest depth
36                      ++nest_hist[i-1];
37              }
38      }
39
40      // find the maximum nest
41      // assume at least 1 txn succeeds
42      int max = 0;
43      for(int i=0; i<NEST_TRY; ++i){
44              if(nest_hist[i]){
45                      max = i;
46              }
47      }
48
49      // print max depth
50      if((max+1)<NEST_TRY){
51              fprintf(stderr,
52                      "Max depth is %d (%.4lf%%)\n",
53                      max+1,
54                      double(100*nest_hist[max])/double(NEST_RUN)
55              );
56      }
57      else{
58              fprintf(stderr,
59                      "Max depth is >=%d (%.4lf%%)\n",
60                      NEST_TRY,
61                      double(100*nest_hist[max])/double(NEST_RUN)
```

```
62                  );
63          }
64
65          return 0;
66  }
```

Table A.4: Max Nest Depth Detection: probabilistically detect the maximum allowed RTM nesting depth. This program assumes the same nesting depth for all cores but the process may be pinned to a core if there is reason to suspect it may vary.