The Report Committee for Thomas Francis Hughes
Certifies that this is the approved version of the following report:

Seven Aspects of Software Transactional Memory

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Christine Julien

Sarfraz Khurshid

Seven Aspects of Software Transactional Memory

by

Thomas Francis Hughes, B.S.

Report

Presented to the Faculty of the Graduate School
of the University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Engineering

The University of Texas at Austin
December 2009

Seven Aspects of Software Transactional Memory

by

Thomas Francis Hughes, M.S.E.

The University of Texas at Austin, 2009

SUPERVISOR: Christine Julien

This paper explores different aspects of transactional memory to identify general patterns and analyze what direction software transactional memory research may be headed. Hybrid hardware-accelerated transactional memory is shown as a better long-term solution than purely software or hardware transactional memory, based on performance and the fundamental issue of software complexity. The appendix provides a chronologically ordered summary of significant transactional memory implementations and transactional memory specific benchmarks.

Table of Contents

Introduction

This paper explores seven different aspects of software transactional memory to identify some general patterns and analyze what direction transactional memory research appears to be heading towards. We show that hybrid hardware-accelerated transactional memory is more likely to be a better long-term solution than either software or hardware transactional memory, based on issues related to performance and the fundamental complexity of large software systems. This paper is not intended to be an exhaustive analysis of transactional memory, but instead focuses on a subset of interesting issues.

MOTIVATION

Building software that runs correctly with concurrent threads of execution is almost universally considered to be difficult:

> [34] A primary challenge is to find better abstractions for expressing parallel computation and for writing parallel programs. Parallel programming encompasses all of the difficulties of sequential programming, but also introduces the hard problem of coordinating interactions among concurrently executing tasks. Today, most parallel programs employ low-level programming constructs that are just a thin veneer over the underlying hardware. These constructs consist of threads, which are an abstract processor, and explicit synchronization (for example, locks, semaphores, and monitors) to coordinate thread execution. Considerable experience has shown that parallel programs written with these constructs are difficult to design, program, debug, maintain, and—to add insult to injury—often do not perform well.

Interest in software transactional memory is based on the belief that using this technique for managing concurrent threads of execution better enables software developers to utilize new highly-parallel CPUs, with both higher performance and lower complexity. Replacing traditional lock-based synchronization with transactional memory eliminates the complexity of fine-grained lock management.

> [9] TM (transactional memory) is a concurrency control paradigm that provides atomic and isolated execution for regions of code. TM is considered by many researchers to be one of the most promising solutions to address the problem of programming multicore processors. Its most appealing feature is that most programmers only need to reason locally about shared data accesses, mark the code region to be executed transactionally, and let the underlying system ensure the correct concurrent execution. This model promises to provide the scalability of fine-grain locking, while avoiding common pitfalls of lock composition such as deadlock.
>
> [36] Using transactional memory, programmers specify *what* should be done rather than specifying *how* this atomicity should be achieved, as they would do today with locks. The transactional memory implementation guarantees atomicity, largely relieving programmers of the above-mentioned tradeoffs and software engineering problems.

Recent transactional memory research purports better performance and less complexity for software developers, with an almost religious zeal: "We often hear claims that people claim that TM is going to solve all the world's problems. We even occasionally hear people make such claims." [15]

This paper examines several areas of transactional memory which are actively being researched, including technically interesting performance and implementation issues as well as higher level considerations regarding which designs will be the most likely to achieve wide-spread acceptance. After a brief background, the following seven sections each examine one specific aspect of transactional memory. A chronologically ordered summary of significant transactional memory implementations can be found in the Appendix, along with details on the transactional memory specific benchmarks.

Background

WHAT IS TRANSACTIONAL MEMORY?

The term transactional memory (TM) was first introduced [29] as a hardware solution that allows multi-processor systems to provide lock-free data structures that "avoid common problems associated with conventional locking techniques in highly concurrent systems". Databases analyze the read and write operations of parallel transactions to ensure that the combined database operations are conflict-serializable, such that the final database state is equivalent to simply executing the transactions one at a time in sequence. Transactional memory applies this same concept to memory read and writes performed by concurrent software threads of execution. When a conflict manager determines two parallel transactions to be in conflict, one will be aborted and rerun.

Transactions may be introduced as a formal programming language construct, and source code need only indicate transactions by wrapping them inside an "atomic{…}" block. If instead an external library is used, then transactions must call functions to start and stop each transaction, and every memory operation must be modified to use the library rather than directly accessing main memory. Compilers may be modified to automatically add transactions to source code, and runtime systems can be updated to automatically instrument pre-compiled applications or libraries.

Transactional memory implementations differ by how read and write changes are monitored, how rollbacks are performed, and when and how conflicts are actually detected. The current state of the art transactional memory systems tend to perform optimistic writes but keep a log to allow rollback, and track memory operations using a bloom filter rather than an exact list, to perform faster conflict detection. Databases deal

with disk latency and have time for more elaborate conflict detection, but transactional memory must be faster by using simple detection, which may even allow some false positives. It is not always clear if conflicts should be detected early (eager conflict detection) when a memory operation occurs, or late (lazy conflict detection) when a transaction is committed. Once of the newest implementations [58] adjusts between eager and lazy conflict detection based on the current runtime performance.

Software Transactional memory (STM) must instrument every read and write operation performed by an application to track memory operations and detect conflicts. Hardware transactional memory (HTM) uses small hardware changes to track memory operations and detect conflicts more efficiently. The primary technique used is modifying the CPU cache-coherency protocol to allow transactions to track the set of memory read and write operations "for free" without software overhead. Although effectively free for the executing code, this requires expensive changes to the physical hardware architecture. The more extensively the underlying hardware supports transactional memory, the more drastic and expensive these hardware changes are. There is currently no widely-available commercial hardware support for transactional memory, so most testing is performed using simulators. Hybrid solutions may switch between software and hardware modes or execute both types of memory transactions simultaneously.

The basic understanding of transactional memory provided by this section is expanded on by the following seven sections. Each section examines a more specialized aspect of TM and how it applies to software transactional memory.

Observation/Analysis

HARDWARE OVERFLOW

Hardware transactional memory requires special considerations for how transactions are tracked and how conflicts are identified. Hardware support is limited by the size of CPU memory cache which presents the challenge of how to best support transactional memory without requiring impractical or unrealistic hardware redesigns.

Observation

As a memory transaction executes, the read and write operations are logged in some way such that the operations can be undone if a conflict is detected and the transaction must be aborted. The earliest transactional memory systems used simple data structures referred to as the hardware transaction logs. Hardware transactional memory introduces novel techniques to track memory operations inside the CPU's local memory-cache, but this is effectively limited to the size of the cache. Small transactions are not a problem, but larger transactions will exceed the size of this cache, which is referred to as overflowing the hardware transaction log. There is a finite limit to the amount of memory operations that can be tracked by hardware transactional memory logs, but more complex designs have been created to support large or even unlimited-size transactions.

Initial Hardware transactional memory designs [1] [51] [10] have evolved over time to support overflowing transaction logs that exceed the size of the hardware memory cache. They may be limited to physical memory size, or fully support virtual memory. Context switching and page faults may be supported for a single process or between multiple concurrent applications on running on the same operating system. Reads and write sets may be tracked for each memory operation using single bits inside the

hardware memory cache, or less exact but more compact methods may be used, such as bloom filters or other signature-based techniques. The transactional workload greatly affects the runtime performance: transactions may be large or small, long running or short, read-heavy or write-heavy, or have either a high or low rate of conflict.

Log-based Transactional Memory (LogTM) [43] builds on previous work to provide what is considered a full featured high-performance reference implementation of a hardware transactional memory system. Benchmarks using a newer version [62] show that: "LogTM-SE with idealized signatures generally performs at least comparably to lock-based programs." Supporting unbounded memory transactions requires significant complex hardware changes to the cache coherence protocol and virtual memory mapping architectures.

Analysis

Complex designs that support unlimited sized transactions in hardware require significant changes to both hardware and the operating system. The cost of the cache-coherency is already expected to increase as the number of parallel processors increases:

> [6] As the number of cores and the subsequent complexity of the interconnect grows, hardware cache-coherence protocols will become increasingly expensive. As a result, it is a distinct possibility that future operating systems will have to handle non-coherent memory, or will be able to realize substantial performance gains by bypassing the cache-coherence protocol.

Transactional read or write sets that exceed the size of the log structure in hardware and overflow can be handed off and managed by a software component. Most high performance implementations abort large (overflowing) transactions and simply rerun them using STM. Early designs could only execute either all software or all hardware transactions, but recent algorithms [58] show that it is possible to execute concurrent software and hardware transactions. It seems impractical to provide complete

hardware transactional memory support for unlimited sized transactions, if a modest hybrid solution turns out to be sufficient. Testing done using the preproduction Rock processor from Sun [15] showed very good results using only minimal hardware support for transactional memory.

PERFORMANCE

Software transactional memory removes mutually exclusive locks but introduces runtime bookkeeping to identify and avoid conflicts between concurrent transactions. This section examines the practical aspects of this new overhead and how it is handled differently by software vs. hardware transactional memory.

Observation

The Haskell implementation [27] may be the poster child for software transactional memory. In addition to very strong formal language support for transactional memory operations, Haskell even introduces two specialized user-controlled transactional actions: retry and OrElse. Recent benchmarks [47] were performed using Haskell's STM implementation with workloads ranging from very simple to medium complexity transactions. Results show that using more than eight processors makes the normalized per-processor execution time go up, which conflicts with STM's premise as a solution for highly-concurrent software systems. Among other things, the Haskell authors suggest hardware mechanisms to help speed up Haskell's STM performance.

The TinySTM [20] provides a lightweight transactional memory implementation similar to and inspired by TL2, which is considered the role model for high-performance software transactional memory systems. Benchmark results show that TinySTM performed better than TL2, but only under certain conditions.

Benchmarks of HyTM and PhTM, both hardware transactional memory systems, running on Sun's pre-production Rock CPU showed up to 4 times better performance than software-only TL2, depending on the specific test parameters.

Intel's experimental STM compiler [19] was used to convert a portion of the Apache HTTP server to use software transactional memory. Benchmarks on an 8-processor quad-core Opteron system showed STM performance worse than the original non-transactional implementation. Under certain conditions, the STM version performed only marginally better than the non-transactional version.

Recent real-world tests like QuakeTM show that software transactional memory performance can fall short of even coarse-grained mutually-exclusive locking.

> [38] Many other works proposed different forms of STM to tackle various performance and correctness issues involved in the STM paradigm. However, these STM implementations are far too expensive in terms of runtime overhead. For parallel applications, STMs typically result in visible slowdowns of 2x or more.

Analysis

Pure software transactional memory comes with an unavoidable level of reduced performance. No optimization technique is likely to ever allow it to perform as well as traditional fine-grained locking, and in some cases STM solutions actually perform worse than simply using a global coarse-grained lock around each atomic block. Even if STM scales better than traditional locking, it may never make up for the underlying overhead added by any software transactional memory implementation, as shown by the Haskell benchmarks.

The optimization challenge is that while one set of strategies works well for algorithm A, the same set of strategies works poorly for algorithm B. Real-world applications unfortunately want to use transactional memory to execute algorithm A and

algorithm B and algorithm C, all at the same time. Complex applications executing a disparate mix of transactions will not be easy to manage. Even if compilers are enhanced to statically determine that a transaction is read only or runtime systems can determine which transactions have a high abort rate, the properties of specific transactions may vary over time. As the runtime state of an application changes, so too can the transactional read and write operations performed by each transaction. The innate complexity of real-world applications and the heterogeneous use of hardware platforms prevent a one-size-fits-all solution.

Hardware transactional memory may avoid the performance problems associated with software only TM, but the required level of flexibility is likely to exceed what is cost-effective to implement using only hardware. Software transactional memory may be a bridge between fine-grained locking and some future hybrid form of hardware-accelerated transactional memory. STM allows experimental implementations to be built today without being constrained by any single hardware design, but purely software transactional memory may always be overshadowed by hardware transactional memory. The fundamental advantage of hardware TM is that using hardware-level operations reduces the amount of software-level overhead. No matter how well STM may be optimized, HTM can improve on it.

HYBRID SOLUTION

Transactional memory systems implemented using either purely software or purely hardware solutions must deal with different challenges. A hybrid design uses a limited set of new hardware features to accelerate software transactional memory, switches between hardware and software mode, or manages to execute both hardware and

software transactions concurrently. This section examines how the newest hybrid designs are more appealing than a software only solution.

Observation

Early hybrid transactional memory systems HyTM [13], RTM [54], and PhTM [36] would only switch between software only and hardware only transactions, as necessary to support large transactions that overflow the hardware transaction log. The switch had to wait for all currently executing transactions to complete, which hurt performance. Newer designs are able to execute both types of transactions simultaneously, thanks to either a novel software algorithm like RingSTM [59] or strong atomicity provided by a novel hardware change like USTM [5].

Transactional memory is an active research area, and novel techniques with significant improvements continue to be discovered. It will be interesting to see how the newest hardware TM research integrates with the newest software TM research, for instance the hardware-based dependency analysis techniques of DATM [52] may be further enhanced by improved software-based scheduler integration like CAR-STM [16]. It is impossible to predict what new innovation will be found next year to trump the cleverest discovery of this year.

Transactional memory designs are becoming increasingly complex. FLEXible Transactional Memory (FlexTM) [58] combines four different techniques from previous transactional memory research to implement a new hybrid hardware-assisted transactional memory system. FlexTM uses bloom filtering to track access, uses versioning and explicit abort techniques from RTM, and introduces Conflict Summary Tables (CST) to track read and write conflicts between processing threads. FlexTM dynamically changes its behavior, for instance between optimistic and pessimistic

conflict detection, based on current runtime information. Seven different benchmarks compared FlexTM against previous TM implementations and coarse-grained locking.

> [58] On a variety of benchmarks, FlexTM outperformed both pure and hardware-accelerated STM systems. It imposed minimal overheads at lower thread levels (single thread latency comparable to CGL) and attained ~ 5× more throughput than RSTM and TL2 at all thread levels.

It has been also suggested that purely hardware transactional memory systems are impractical, and that there is a complexity threshold that must not be exceeded.

> [36] Recently, numerous proposals for "unbounded" HTM have appeared in the literature in an effort to overcome the shortcomings of bounded and best effort HTM designs. However, all of them entail substantially more complexity than the much simpler best effort implementations, and therefore it will be much more difficult to integrate unbounded HTM solutions into commercial processors in the near future. We believe that simple best effort HTM support can be implemented much sooner. However, if used directly, best effort HTM support can impose unreasonable constraints on programmers, for example requiring them to think about the number and distribution of cache lines accessed by transactions or other architecture-specific implementation details. These tradeoffs are a large part of the reason HTM has not been embraced by the computer industry to date.

Analysis

Recent research has focused on improving conflict management decisions as well as tighter integration with the operating system: TM and the scheduler, TM and the virtual memory manager, and TM interacting with traditional mutually-exclusive locks. It seems unlikely that these complex optimizations will ever be implemented purely in hardware, but rather some type of hybrid design using hardware elements for maximum performance and software components to oversee the hardware transactions, resolve conflicts, and make intelligent just-in-time runtime adjustments.

I am optimistic that hardware-hybrid transactional memory will eventually be embraced, but the reason why has little to do with the concept of transactional memory and much more to do with the fact that transactional memory provides an opportunity to

improve multi-processor utilization through moderate hardware changes. Modern computer architecture already uses tricks to extract more performance without increasing the clock-rate: instruction level parallelism with longer instruction pipelines and out-of-order execution; thread level parallelism running multiple threads on one processor core; data parallelism to execute one operation across multiple registers or memory locations concurrently. Adding hardware hooks to enable transactional memory would not be a revolutionary design jump, but rather the next logical evolutionary optimization.

It is unfortunate that the latest Sun server roadmap [44] no longer includes the Rock UltraSparc processor, but Azul Systems has been building niche-market hardware that supports speculative lock elision for five years. It is only a matter of time before we begin to see mainstream hardware support for transactional memory systems.

STRONG ATOMICITY

Transactional memory designs must specify guarantees for the atomicity of transactions. This section examines the weaker atomicity guarantees sometimes specified by software transactional memory implementations in order to improve performance.

Observation

Allowing memory changes outside of an atomic block can be a challenge for transactional memory. Weak isolation allows memory operations outside of a transaction to view partial results of atomic transactions. To avoid this issue and ensure strong isolation (string atomicity) between atomic blocks, any memory locations accessed from inside a transaction may require that non-transactional access be instrumented to check for conflicts. Most TM implementations that provide strong isolation for transactions do so by adding overhead to the non-transactional code.

> [42] HTM systems support strong isolation, which implies that transactional blocks are isolated from non-transactional accesses. There is also a consistent ordering between committed transactions and non-transactional accesses. In contrast, high-performance STM systems do not support strong isolation because it requires read and write barriers in non-transactional code and leads to additional runtime overhead. As a result, STM systems may produce incorrect or unpredictable results even for simple parallel programs that would work correctly with lock-based synchronization.

The UFO STM [5] introduces User-fault-on (UFO) memory protection to provide user controllable read-barriers and write-barriers. When a process performs reads or writes that violate this memory protection, a UFO memory-fault occurs. Hardware transactions flag memory with read or write protection so that conflicting concurrent software transactions will fault, run a special handler, and can either delay or abort the software transaction as necessary. This provides strongly-atomic transactions with little overhead added to hardware transactions and almost no overhead added to software transactions. This is a general-purpose memory protection mechanism useful for other applications as well, for example concurrent garbage-collection or self-modifying code.

Analysis

The only way for software transactional memory to provide strong atomicity to existing non-transactional applications (or libraries) is to instrument executable code. Regardless of the efficiency, this always adds some amount of software overhead. Hardware transactional memory can monitor memory access and ensure strong atomicity without overhead, using novel techniques such as UFO. For transactional memory to gain acceptance it needs to support legacy code and provide strong atomicity between new transactional memory operations and legacy non-transactional code. This makes hardware or hybrid solutions a better choice for transactional memory.

LOCKING

Transactional memory can provide software with atomic blocks, but it will not simply eliminate the need for locking. This section reviews the concurrent locking problems transactional memory solves and the issues that still require explicit locking.

Observation

Methods or modules using mutually-exclusive locking to protect critical sections may not be easily combined into new compound methods or modules, as this usually requires access to the internal locking and introduces opportunities for deadlock. The larger the software system, the more modules are combined with each other and the larger the risk. If a software language supports composability, then individual atomic operations can be combined into larger atomic actions. For transactional memory systems, this done by nesting transactions such that the combined sequence of atomic operations is executed as a single atomic operation. Traditional locks do not support this nesting behavior without introducing inefficient coarse-grained locking.

A good example [34] is a hash-table implementation that supports parallel insert and delete. Adding or removing entries in the hash-table occurs atomically, but moving an entry from one hash-table to another requires additional synchronization. Either coarse-grained locking must be used to protect every access to these two hash-tables or the internal locking must be exposed. New lockTable and unlockTable methods would need to be called on both tables in a globally consistent order, to atomically move values between hash-tables without causing a deadlock.

Lock-Aware Transactional Memory (LATM) [24] allows the uses of traditional shared mutually-exclusive locks both inside of transactions (LiT) and outside of transactions (LoT). Previous work supports non-transactional locking by converting locks

into transactions, but this allows deadlocks to occur. Any detected deadlocks may be broken, but this allows the possibility of an incorrect program state. LATM requires the programmer to declare which transactions may cause locking conflicts, using two different levels of granularity. The specific list of locks that may conflict can be provided, or simply a flag is used to indicate that any lock may conflict. Because transactions may abort, LATM requires that transactions only acquire locks. All locks acquired by a transaction are automatically released after the transaction commits.

Transactional memory also prevents I/O operations from being performed from inside of a transaction. Those operations must be done outside of an atomic block, as you cannot rollback an I/O operation once an Ethernet packet is sent or a new value is written to the hard disk. Recent work [24] provides correct efficient integration of traditional mutually-exclusive locks with memory transactions, but transactional memory cannot eliminate the need for careful locking around I/O operations.

JudoSTM [45] uses dynamic binary rewriting to instrument existing code and libraries at runtime, using privileged transactions to allow kernel system calls, external libraries, or I/O operations to be performed. Unfortunately, these privileged transactions must still be executed serially; kernel calls cannot be undone when a transaction aborts.

Analysis

Course-grained locking may be the straightforward way to ensure that a program executes correctly, but poor performance usually makes it impractical. Fine-grained locking reduces the performance penalty but often results in higher complexity and more obscure design defects. Medium-grained locking around specific hotspots is often used as a compromise. Transactional memory intends to provide the full performance of fine-grained locking while only requiring the complexity of course-grained locking. You can

think of it as a black box that performs automatic speculative fine-grained locking around the individual memory locations accessed by any critical section. Concrete explicit locking around critical sections is effectively replaced by implicit locking around atomic blocks, performed by transactional memory systems only when necessary to resolve conflicts.

Transactional memory provides new strategies for optimizing specific concurrency problems, but it should only be considered a complex technique for efficient automatic optimization of critical sections. Locks are used for two reasons: to ensure atomicity of critical sections and for coordination between concurrent processes. Software stills need a way to efficiently suspend, notify, fork, join, and block multiple threads of execution. To effectively coordinate a pool of threads you still need to interact with the scheduler. Recent work [16] shows that significant performance gains are possible by tightly integrating transactional memory with the thread scheduler.

Transactional memory does not eliminate the need for traditional synchronization techniques like semaphore-based mutually exclusive locking. Atomic critical sections are only one tool necessary to synchronize concurrent processes. Do not be fooled into thinking that Dijkstra's semaphore will be retired anytime soon.

SIMPLICITY

Some transactional memory solutions require software designers to provide additional information regarding transactional memory behaviors, beyond simply marking critical sections. This section consolidates quotes from transactional memory research and explores the idea that simplicity is more important than performance.

Observation

Transactional memory performance continues to improve using novel optimization techniques but often these require additional work by the programmer in the form of additional meta-data markup. Regardless of the amount of work required, this goes against the basic assumption that transactional memory frees developers from the complexity of managing concurrent transactions.

> [52] Several proposed extensions to the TM programming model can be used to achieve higher performance, including privatization, early release, escape actions, open and closed nesting, Galois classes, transactional boosting and abstract nested transactions. These techniques all fundamentally affect the programming model, increase programmer effort, and increase program complexity as the price for better performance. They differ in their degree of applicability and the difficulty of reasoning involved, as well as the amount of additional compromises they force on their users. For example, using escape actions to implement a counter requires the programmer to also write a compensation block, which is a significant programmer burden. Moreover, semantics may be weakened when using this approach (e.g. a counter implemented this way is no longer monotonically increasing). In Galois and transactional boosting, the programmer needs to provide inverse operations for the concurrent data structures, which might be difficult (e.g., k-d tree), as well as define commutativity relationships between the various operations.
>
> [34] Shared data declarations, however, shift the burden of correctness from the TM system back to a programmer. Accidentally omitting a declaration can cause a data race, which is a problem that transactions should eliminate. Again, program analysis can alleviate the burden. For example, a compiler can use escape analysis or a type system to conservatively identify data that cannot be shared with another thread.
>
> [35] Allowing programmers to disable privatization where it is not needed is undoubtedly better than requiring them to provide annotations where it is needed. However, with such severe overhead, programmers will be motivated to disable privatization aggressively. Apart from the additional burden this places on the programmer, it is inevitable that programmers will incorrectly disable privatization, or perhaps subsequent code changes will render a previously correct usage incorrect, again resulting in subtle nondeterministic bugs. We are therefore motivated to find the best implicit privatization mechanisms possible to avoid the need for explicit programmer annotations.

Analysis

For transactional memory to be adopted it should perform as well or better than coarse-grained or medium-grained locking without requiring the software developer to provide additional meta-data or other complexities. Smarter compile-time and runtime analysis should be able to provide much of the information required to perform advanced optimizations. Techniques that still require additional developer markup should be avoided, as the primary selling-point of the transactional memory paradigm is reduced complexity.

The performance gain or loss due to transactional memory may not turn out to be as important as how well it can eliminate complexity. Automatic garbage collection can be less efficient than painstakingly manual management of the heap, but the benefit of never dealing with pointers often outweighs any lost efficiency. There is great value in not tracking down which components lock what data structures and in what order; or why your application deadlocks at startup in the customer's environment, but never during internal long-running testing.

NEW ALGORITHMS

This section examines a few novel algorithms developed by transactional memory research and the possible long term effects. Some techniques which improve software transactional memory performance today could actually make hardware or hybrid transactional memory more appealing going forward.

Observation

The proliferation of parallel core CPUs has caused a renewed interest in using non-blocking algorithms optimized for highly concurrent software systems. Modern non-

blocking hash tables scale almost linearly [40] [56] [12] up to hundreds of parallel processors, and offer a drop-in replacement for lock-based concurrent hash tables that perform poorly as the number of parallel threads increases.

The recent software transactional memory design SkySTM [35] also included the Scalable NonZero Indicator (SNZI) algorithm [18] which provides shared counters using a hierarchical data structure that scales well for a large number of concurrent threads. This counter is not a replacement for traditional counters, but rather removed certain counter properties (for example, the actual count value) to work better with transactional memory. The SNZI algorithm performed between 2.5 to 8 times slower than a simple CAS counter when using a single thread, but SNZI always performed better than a simple CAS counter with 3 or more threads, and 550 times better with 48 concurrent threads.

Analysis

Traditional mutually-exclusive locking may be optimized or even circumvented to support newer highly-parallel hardware environments. What if the cache-coherency changes being designed for HTM can be leveraged to improve the performance of traditional locking? Transactional memory research may inadvertently uncover techniques that enable traditional mutually-exclusive locking to scale more linearly. These techniques may also help improve the performance of existing non-blocking algorithms and data structures, which already somewhat alleviate the need for switching to transactional memory.

Software transactional memory is challenged by the additional overhead required to track memory read and write operations and detect conflicts, which inevitably results in lower performance than using hardware transactional memory. As standardized data structures are improved to work more efficiently with a large number of concurrent

processors, the benefits of software transactional memory are diminished. If the performance gap between software transactional memory and non-transactional solutions remains the same, it will be an uphill battle uphill battle for a purely software solution to gain wide-spread acceptance.

Conclusion

NO SILVER BULLET

Software transactional memory can help reduce "accidental difficulties" [8] and alleviate some of the innate complexity surrounding concurrent programming, but transactional memory is not the panacea it was initially purported to be. It seems to be the newest "silver bullet" in software development. Transactional memory improves performance and/or reduces complexity in many circumstances, but it does not eliminate the complexity of building software to utilize multiple processors. Like the silver bullets that came before it, the paradigm of transactional memory will provide incremental but moderate improvements that help software developers to better manage the ever-increasing complexity of software development.

Software only implementations appear less desirable than a hardware design, but there is currently no widely-available commercial support for hardware transactional memory. CPU designs are more likely to provide support for transactional memory as the number of parallel processors per chip continues to increase. When hardware support finally arrives, it will most likely be a simple design requiring the least drastic changes to the underlying hardware architecture. Mainstream software development may take interest late enough to avoid software transactional memory altogether, and leapfrog directly to the choice of using hybrid hardware-accelerated transactional memory or not.

Appendices

APPENDIX A - IMPLEMENTATIONS

Summaries of noteworthy transactional memory implementations are presented below, ordered chronologically and with more details provided for newer designs. The early transactional memory designs were clearly hardware-only or software-only, but the more recent hybrid designs build on previous work from both groups.

HTM

Although earlier work used hardware to support database transactions, the term Transactional Memory was first introduced [29] in 1993. It describes the set of logical primitives and the architecture changes required to implement a cache coherency protocol that supports Hardware Transactional Memory. HTM performance results were generated using a simulator modified to implement the HTM instruction set changes. It's interesting to note that the original intent of transactional memory was focused on the hardware layer rather than at the software layer.

RSTM (C++)

The Rochester Software Transactional Memory [53] has evolved over time based on the work of an early paper [39] that describes various algorithms used to synchronize concurrent processes without using spin-locks. The RSTM library works with C++ programs using the standard (UNIX) pthreads library.

Oklahoma

The Oklahoma Update [60] takes its name from a song in the famous Oklahoma musical named: All Er Nuthin'. It documents the use of a reservation system that enables

atomic reads and writes of memory which is synchronized across multiple processors. This early paper (1993) is frequently referenced by newer transactional memory research.

SLE/TLR/OTC

Speculative Lock Elision [49] and Transactional Lock Removal [50] and explore the ways locks may be optimized or even completely elided under certain conditions in highly concurrent systems.

Optimistic Thread Concurrency [4] describes how Azul Systems use a transactional-style speculative technique to optimistically execute synchronized blocks and roll-back if interference is detected. This enables running thousands of threads without an excessive amount of lock contention. Azul Systems makes computational appliances based on the Vega CPU. The current Vega 3 processor provides 54 hardware cores and the Vega series 7300 system uses between 4 and 16 processors, for a total of up to a total of 864 hardware threads.

DSTM

Dynamic Software Transactional Memory [31] implements pure STM using Java and includes some benchmark results using a SunFire system with 72 hardware threads. It provides STM primitives similar to the Glasgow Haskell Compiler, but has only limited support for nested transactions. DSTM is somewhat limited by the JVM, but makes use of hardware commands like compare-and-swap.

TCC

Transactional Memory Concurrency and Consistency [26] proposes changes to the underlying hardware cache coherence API such that all memory actions are transactional. This research details the proposed ISA hardware changes that enable the hardware portion of transactional memory to be implemented.

OSTM

Object-based Software Transactional Memory [21] [22] is an STM implementation scoped to individual objects. The authors also proposed Word-based Software Transactional Memory (WSTM) and Multi-word Compare and Swap (MCAS). In later papers, one of the original authors (Fraser, at Cambridge) renamed OSTM to be simply Fraser's STM (FSTM). This work is purely software-only, with no support for hardware optimizations.

UTM/LTM

Unbounded Transactional Memory and Large Transactional Memory [1] from MIT, details the development of a STM implementation which handles transactions that exceed the size of the hardware cache. LTM limits the size of transactions to the actual physical memory size, while UTM supports virtual memory sized transactions.

Two performance tests were conducted. First a GNU version of the JVM was recompiled to replace locks with atomic blocks and an then an existing benchmark suite was run to measure performance improvements of STM over locking. A multiprocessor simulator was used to evaluate the behavior with up to 32 cores. Second, a special version of Linux, running in user-mode under a host Linux OS, was instrumented and the kernel was built, as a computationally intensive test, to evaluate the transactions that occur inside the kernel: percentage of time spent inside a lock, average number of hardware cache-lines required per transaction, maximum transaction size. Detailed test results confirm the assumption that transactional memory provides better performance for an environment with a large number of parallel CPUs.

SXM (C#)

Microsoft Research provides [30] a software transactional memory library for .NET managed code, but it has not been updated since 2005.

Glasgow Haskell Compiler (GHC)

The benefits to software design are illustrated by the work done using the Haskell implementation [27] of STM. The Haskell language is a purely functional language with strong typing that provides a good framework for building STM software. Enabling composability of atomic actions is a major benefit of using STM for large complex software systems, and this aspect is illustrated well [48] by the work done using Haskell STM. Additional research [28] shows how enforcement of data invariants at the end of every transaction can be added, not to improve performance but rather to provide safety.

Haskell's STM introduces two user-controlled actions to STM: retry and orElse. The retry action allows programs to deliberately abort a transaction and cause it to rerun. Haskell delays the aborted transaction until another transaction commits a write that conflicts with the read set of the aborted transaction. This prevents wasted work, since the aborted transaction will abort again and again until it reads different values from memory. The orElse is used to specify an action (transaction) to execute when a transaction aborts. This allows for more complex algorithms and special handling of transactions which may have a high abort rate.

Recent research [47] benchmarked Haskell's STM (GHC) implementation on an SGI Altix 4700 system with 64 dual-core SGI processors using a set of 10 different tests, ranging from very simple to medium complexity transactions. The results show that most tests reached their maximum speedup when using only 8 processors[1]. Adding additional

[1] STMlite embraces this STM limit by optimizing for 8 processors or less.

processors makes the normalized per-processor execution time goes up as more processors are added. This goes against the theory that STM performance gains from a large number of processors will overshadow the extra overhead seen when running with only one or even a few processors. Details are provided regarding abort rates, commit times, the percentage of wasted work, and a detailed analysis of the memory cache behavior seen while running. Among other things, the authors suggest hardware mechanisms to help speed up Haskell's STM performance.

VTM

Virtual Transactional Memory [51] implements TM such that data is allowed to overflow out of the hardware and into a virtual structure in order to allow large transactions as well as context switches and page faults. Details show that VTM uses Bloom filters, has seven distinct states each transaction can pass through, and the virtual memory is specific to a single running process rather than the entire system. VTM is an improvement over UTM, which does not support context switching and is apparently less efficient at detecting conflicts.

ASTM (Java)

Adaptive Software Transactional Memory [37] is an object-based STM system for Java, which improves on OSTM by implementing adaptive behaviors that control how and when the locks are acquired. Conflicting transactions can be detected earlier, which leads to faster aborts. Speculative object modifications can be tracked as either a list of changes or a complete duplicate object. The system adapts its behavior at runtime based on the current workload.

HybridTM

Hybrid Transactional Memory [33] from (University of Michigan and Intel Labs) should not be confused with HyTM (from Harvard, Brown, and Sun). This work provides an alternate implementation to the DSTM API, which introduces a HTM mode used dynamically for executing memory transactions. Three attempts are made using the hardware mode, before falling back to the software-only mode. A multiprocessor simulator is used for performance results, and show that the hardware-mode provides improvement over the pure-software model. It is unfortunate that the authors of this paper chose a name similar to HyTM.

HyTM

Hybrid Transactional Memory [13] implements transactional memory to operate using either software or hardware modes. A CPU simulator was used to compare expected performance with HTM available, and a 24-core SunFire server and a standard benchmark suite was used to investigate the resulting STM performance. In addition, the BerkeleyDB engine was converted from using locks to memory transactions. The authors found source code comments indicating that the BerkeleyDB authors had attempted fine-grain locking, but found it too complex and abandoned the idea. The single shared-lock used by BerkeleyDB was well suited for conversion to STM.

HyTM outperforms fine-grained locking using software-only mode, and HTM acceleration is expected to provide even better performance. The authors hope to establish a standard API that can be used today and enable HTM acceleration in the future, when chip-makers introduce the necessary hardware changes.

XTM

The eXtended Transactional Memory system [11] is similar to VTM, but with less required hardware changes and support for nested transactions. Unlike most transactional memory systems, XTM does not simply flatten nested transactions when they occur.

XTM is a pure software implementation that support fully virtualized memory transactions, while XTM-g and XTM-e both leverage minor hardware support to gain additional performance benefits. Transactions that overflow the hardware cache size in XTM are aborted and rerun using the virtual-memory space. XTM-g allows a more efficient gradual overflow from the hardware cache into the virtual memory. XTM-e uses an eviction log buffer to track changes (read or write) by cache-line rather than by virtual memory page; this helps reduce false sharing.

Details on performance testing and results show that XTM-e and XTM-g both provide performance comparable to VTM, but with much less hardware complexity. Software only XTM is slower, but works regardless of the underlying hardware architecture.

PTM

Unbounded Page-Based Transactional Memory [10] is a transactional memory system that improves on VTM. The PTM design is integrated into the operating system's memory manager so that transactions can be shared across multiple processes on the same system, not just within a single application. It uses shadow blocks and read/write bit vectors to track speculative changes and takes advantage of the hardware cache coherence protocol to optimize transactions that fit inside the processor's cache.

HASTM

Hardware accelerated software transactional memory [55] implements a hybrid STM by introducing a few new hardware instructions. Most transactions can be validated (conflict free) using hardware, but software checking is used as a fall-back in special situations. Notable features include support for managed runtime environments which includes using object-level granularity for detecting transaction conflicts, support for garbage-collection interruptions without aborting transactions, consistent performance across a variety of transaction sizes, and flexible support for nested transactions.

LogTM

Log-based Transactional Memory [43] is a transactional memory design similar to UTM, LTM, and VTM. The most notable feature is the use of eager version management so that commits are faster, but aborts and rollbacks are much slower. This is the best strategy when most transactions commit, and only a few transactions abort. LogTM does not currently support context switching within a transaction, while UTM and VTM do. Nested transactions are subsumed by the top-most parent transaction. A simulation framework was modified to support the necessary MOESI (cache coherency) protocol changes, and benchmarks results were gathered to determine the LogTM performance improvement.

TL2

The Transactional Locking 2 [14] algorithm provides software transactional memory using a global version-clock. Variations were created for both per-object and per-stripe tracking of changes, and an interesting versioned-counter algorithm was introduced. Some benchmark results are included comparing TL2 against algorithms from Fraser, Ennals, Hanke, as well as the previous TL algorithm. Benchmarks were

performed using the Red-Black tree implementation (TreeMap) provided with Java 1.6. Source code was released and TL2 has subsequently appeared in more extensive benchmark comparisons.

McRT-STM

Multi-Core RunTime Software Transactional Memory [57] is provided as part of the McRT package, which also includes a scheduler, a flexible memory manager, and supports OpenMP, Pthreads, and ORP. McRT primarily provides a detailed look at how different STM strategies affect performance. Transactional reads may either acquire locks for each read or track version numbers updated by writers, but the authors found that "... read versioning performs an order of magnitude better than reader locking." Writes may be handled by either saving them up until commit time (buffering) or writing out immediately to main memory and keeping an undo log. The authors found that "... undo logging performs better than write buffering." Conflict detection can be managed by tracking either entire objects or cache-line sized blocks of memory. Results were mixed, as cache-line locking performs better than object locking in some cases, but not others.

Benchmarks were run using only basic data structures on an IBM 16-processor (Xenon) Linux system, and results are compared against the performance of using a single coarse-grained lock. McRT-STM outperforms coarse-grained locking in some cases, for example binary search trees, hash-tables, and unsorted linked lists, but not for more complex structures like B-trees or sorted linked lists.

PhTM

Phased Transactional Memory [36] improves on the Hybrid Transactional Memory (HyTM) system developed by the same authors. PhTM provides a hybrid transactional memory implementation that dynamically switches between modes based

on the transactional workload. Modes are defined for hardware-only, software-only, hybrid, and a sequential mode optimized for single-threaded execution. When the mode changes, new transactions are delayed and in-progress transactions are allowed to complete to prevent two different types of transactions from executing at the same time. This eliminates the need to coordinate conflict checking between different transaction types, and allows HTM transactions to execute without ever checking for conflicts with STM transactions, and vice-versa. Coordination between multiple styles of transactional memory would increase complexity and overhead. PhTM was constructed using LogTM for the hardware mode and TL2 for the software mode. Performance evaluation used an existing LogTM simulator built with GEMS/Simics. BerkeleyDB and a red-black tree were run as benchmarks.

RTM and RTM-Lite

The University of Rochester developed an integrated hardware-software approach [54] called RTM (along with RTM-Lite) using C/C++ and based on a previous software-only implementation called RSTM. A hardware enhancement called alert-on-update (AOU) allows cache coherency events to be relayed back to the CPU such that contention management can be handled by software rather than hardware. This enables advanced conflict management behaviors to be implemented which would be impractical to accomplish at the hardware level. For example, RTM can delay a read-write transaction to allow a concurrent read-only transaction to complete first, rather than simply aborting one of the two transactions. A second hardware modification called Programmable-data-isolation (PDI) provides a way to control when changes to cached data is propagated to other processes. Normally this happens immediately, to maintain cache coherency. By delaying notifications, RTM is able to optimize performance of some operations. RTM

implements both AOU and PDI, but RTM-Lite only includes AOU. An optimization called fast-path is included to bypasses almost all transactional overhead when single-threaded execution is detected.

Because HTM requires hardware support, all benchmarks were performed on a simulated 16-way SPARC system, using the GEMS/Simics infrastructure. Primitive data structures (HashTable, RBTree, LinkedList, LFUCache, RandomGraph) were used for performance testing. RTM was shown to be an improvement over RSTM, and coarse-grained locking (CGL) was shown as a baseline for comparison. The results were as expected: CGL generally performed best with a single thread of execution, RTM generally performed faster than RTM-Lite, and RTM-Lite generally performed faster than RSTM. One test, involving large transaction sizes, shows RTM and RTM-Lite performing at only 20%-35% the throughput of CGL, regardless of the number of concurrent threads of execution.

SigTM

Signature-accelerated transactional memory [42] is a hybrid TM system that uses hardware signatures to track the read and write sets for transactions and provides strong isolation. Benchmarks show that SigTM outperforms STM implementations by 30% to 280% while trailing HTM by only 10%. New hardware operations are introduced to provide Bloom filter operations that track memory read and write sets, but no changes to the hardware memory caches are necessary. Benchmarks test how different Bloom filter sizes affect the performance, and fount that a small read set size performs poorly while the size of the write set does not affect performance. “SigTM was inspired by the Bulk HTM that first proposed the use of signatures for conflict detection.”

JudoSTM

The Judo dynamic binary rewriting (DBR) system was used to implement JudoSTM [45] for x86 processors. Judo is a similar to DynamoRIO, which was acquired by VMWare in 2007 and released as Google Open Code in 2009. DBR rewrites native X86 code on-the-fly and allows executing compiled code to be instrumented at runtime. This is frequently used for debugging, but used by JudoSTM to provide software transactional memory without requiring any changes to the underlying application. JudoSTM uses optimistic concurrency (invisible readers) and value comparison to check for conflicts at commit time. Privileged transactions allow for kernel system calls, external libraries, or I/O operations to be performed. Legacy exclusive locking is effectively elided and JudoSTM relies on value comparison to decide when to abort and rerun transactions.

JudoSTM uses both coarse-grained and fine-grained locking to manage transactional commits. A subset of each shared commit lock includes a version-counter used by read-only transactions to commit without acquiring any locks. Fine-grained locking uses an 8192-region hash to associate locks with memory locations. To avoid waiting for each transaction to sort the fine-grained locks and acquire them in order to prevent deadlock, the spin-locks timeout and cause transaction to abort. Because JudoSTM does not insert a check for every memory read, but instead instruments branch edges to validate the read set. This prevents infinite loops which could be caused by reading old or inconsistent values. Because the entire binary is rewritten during execution, the standard malloc library works and privileged transactions allow system calls to extend the active heap-size. The standard libc malloc performed poorly, and was replaced with the highly optimized Hoard memory allocator.

A four-processor Xenon system was used to compare JudoSTM against RSTM, coarse-grained-locking, and fine-grained locking. Benchmark tests consisted of a counter, a linked-list, a hash-table, and a red-black tree. JudoSTM and RSTM have similar performance results for hash-tables, but JudoSTM appears to scale better for a linked-list test. The red-black tree test shows JudoSTM approaching the performance of coarse-grained locking when four processors are used, while RSTM is two times slower. Course-grained locking outperforms JudoSTM and RSTM in most cases; log-scale graphs are used to show the fairly wide performance gap. JudoSTM shows that dynamic binary rewriting is an effective technique for implementing software transactional memeory, but the final result shows only marginal improvements over RSTM for tests using basic data structures.

TinySTM

The TinySTM [20] provides a lightweight transactional memory implementation using locks to protect shared memory locations. The Lazy Snapshot Algorithm, developed in previous work by the authors, is used to manage updates. Although very similar to and inspired by TL2, TinySTM differs by using early encounter time locking to detect conflicts faster and to allow read-only transactions to commit safely along side read-update transactions. Hierarchical locking is introduced to optimize the speed of validating large reads sets during commits. This technique is effective when read-set size is large and the number of competing concurrent writes is small.

Benchmarks were run using the Red-black tree from STAMP as well as a sorted linked-list provided by the authors, on an 8-core (Xenon) Linux system. TinySTM was tested using two transaction strategies: write-back (track updates in a log and write at commit) and write-through (change main memory directly, but keep an undo log).

Results showed that TinySTM performed better than TL2, but only under certain conditions, such as accessing a linked list that generates a large write set. Additional benchmark testing showed the effect of dynamically tuning the runtime parameters used by TinySTM: the size of each memory location locked, the total number of locks tracked, the "amount of shift" used by the hash-function mapping memory locations to a lock, and the number of levels used by hierarchical locking. Three-dimensional graphs show how performance varies for different values and combinations, and different two different sweet-spots exist for both the red-black tree and linked-list benchmarks. An automatic tuning strategy was implemented to adjust parameters over time based on recent performance, leading to a set of values comparable to what was found through manual testing. The auto-tuning is appealing because there seems to be no ideal set of parameters to provide the best performance across all workloads.

TokenTM

The TokenTM [7] implementation provides unbounded hardware transactional memory based on LogTM's per-thread logging, but introduces a new conflict detection algorithm based on tokens. These tokens allow individual threads to modify token state more efficiently and decrease overhead for large transactions without having an adverse affect on small transactions. This is an improvement over using Bloom filters or other hash-link signatures to summarize the actual read or write sets, which causes some amount of false conflicts. TokenTM uses a double-entry bookkeeping system where tokens are subtracted from a table representing memory blocks and added to the per-thread transaction log. Each block starts with a fixed number of tokens, read access requires only one token, and write access requires all of the tokens. A fast token release

variation of TokenTM is developed to allow small transactions to release multiple tokens with only a single hardware operation.

The GEMS/Simics simulator was used to emulate a 32-core SPARC system, and benchmark tests were taken from both SPLASH and STAMP. The smaller SPLASH tests show that TokenTM performs as well as LogTM for small transactions. Longer transactions provided by the STAMP tests show that TokenTM can perform better than LogTM. In the Delaunay STAMP test, TokenTM performed 5.7 times better than LogTM. "TokenTM is most valuable if either the large read/write sets of Delaunay transactions become common or designers prize robust performance that is insensitive to signature design."

RingSTM

The RingSTM [59] uses a novel ring shaped data structure to provide software transactional memory with less per transactional overhead than TL2 as well as efficient privatization. Rather than using a set of locks representing each memory area locked by a transaction, RingSTM uses three different sized Bloom filters to track and communicate write sets. The algorithm greatly reduces commit time overhead. Conflict checking is accomplished in a fixed O(1) time rather than O(N) time, tied to the number of blocks. Read-only transactions proceed without performing any writes to shared data structure and write transactions only block other transactions while pushing delayed writes out to main memory. This requires O(W) time rather than O(R+W) as required by TL2 or most other STMs, where W and R represent the total number of write or read operations by a transaction. The ring-based algorithm provides support for multiple transaction priority levels to allow one processor to jump in front of the others, and supports inevitable transactions that allow a single transaction to claim exclusive control and perform in-

place memory writes without any STM overhead. Three different variations of RingSTM were developed, including a ring size of one that only allows a single writer at a time and a ring that allows out-of-order writes, but testing shows that these two variations provide no performance benefits over the full RingSTM algorithm. Privitization is provided "for free" since transactions are committed based on their location in the ring and all reads use eager conflict detection based on the Bloom filters.

Performance testing was performed on a Sun Niagara platform, with 8-cores providing a total of 32 hardware threads. A test using red-black trees shows that TL2 outperforms RingSTM with more than 6 threads when using a small (32 bit) Bloom filter, and more than 12 threads when using a large (8196 bit) Bloom filter. This plus other test results indicate that small Bloom filters result in too many false positives to be effective. A test using a random graph shows that RingSTM outperforms TL2 by 33% to 50% for large read sets and moderate write sets. Testing using a hash-table shows that both TL2 and RingSTM are unable to scale past 10 or 20 threads due to very high contention for shared global elements. The authors express interest in using dynamic tuning and hardware acceleration to further improve the performance of RingSTM.

USTM

The UFO STM [5] uses a novel approach to build a strongly-atomic hybrid transactional memory system. User-fault-on (UFO) is a new type of memory protection, providing user controllable read-barriers and write-barriers. When a process performs reads or writes that violate this memory protection, a UFO memory-fault occurs. Hardware transactions flag memory with read or write protection so that conflicting concurrent software transactions will fault, run a special handler, and can either delay or abort the software transaction. This provides strongly-atomic transactions with little

overhead added to hardware transactions and almost no overhead added to software transactions. It is a general-purpose mechanism useful for other applications as well, for example concurrent garbage-collection or self-modifying code. Benchmarks are performed using STAMP to compare USTM against versions of HyTM, PhTM, and TL2. Results show that USTM performs as well as LogTM in almost all cases and includes analysis of how different failover rates between HTM and STM affect performance.

FlexTM

FLEXible Transactional Memory [58] combines four different techniques from previous transactional memory research to implement a new hybrid hardware-assisted transactional memory system. FlexTM uses bloom filtering to track access, uses versioning and explicit abort techniques from RTM, and introduces Conflict Summary Tables (CST) to track read and write conflicts between processing threads. FlexTM dynamically changes its behavior, for instance between optimistic and pessimistic conflict detection, based on current runtime information. Seven different benchmarks were used to compare FlexTM against RSTM, TL2, RTM, and coarse-grained locking, but not fine-grained locking. The GEMS/Simics simulator is used to run the benchmarks and to provide the new (simulated) hardware extensions required by FlexTM, and test results show that FlexTM outperforms previous STM designs.

> [58] On a variety of benchmarks, FlexTM outperformed both pure and hardware-accelerated STM systems. It imposed minimal overheads at lower thread levels (single thread latency comparable to CGL) and attained ~ 5× more throughput than RSTM and TL2 at all thread levels.

CAR-STM

The Collision Avoidance and Resolution STM [16] manages conflicting transactions by re-executing aborted transactions in such a way that future conflicts are

avoided. This is accomplished by forming dependency rules at runtime used by the process scheduler to serialize those transactions that otherwise might be in conflict. This is a novel technique, although one prior work used a single queue to execute transactions once high-contention was detected. CAR-STM uses one queue per processor, and has two different style for tracking conflicting transactions. The Basic serializing contention manager (BSCM) resolves conflicting transactions by assigning the aborted victim transaction to the queue of the conflicting transaction's processor. The permanent serializing contention manager (PSCM) keeps a "permanent" record of conflicts, which may included subordinate transactions, and prevents the conflict from ever occurring again.

CAR-STM was added to RSTM and testing using STMBench7. PSCM requires more bookkeeping, and BSCM is shown to be more efficient in the benchmark results. Compared to RSTM, CAR-STM shows a dramatic reduction in execution times, running on an 8-core (Xenon) Linux system, with hyper-threading disabled. It should be noted that the number of threads exceeded the number of physical cores, which puts the baseline RSTM implementation at a disadvantage. CAR-STM with between 2 and 32 cores show a speed-up of between 1.7 and 36 compared to RSTM. The standard deviation for RSTM was also much higher than for CAR-STM. This is explained as RSTM having a high probability of a live-look caused by the OS scheduler, while CAR-STM manages the scheduler to effectively avoid conflicts. "This work suggests that making the operating-system scheduler transaction-aware may yield significant performance gains."

DATM

Dependence Aware Transactional Memory [52] improves the handling of conflicting memory transactions by sharing of data values between processors using an

enhanced cache coherency protocol. DATM allows transactions to commit as long as they are conflict serializable and reduces the number of transactional aborts or subsequent retries. The new Forward Receive MSI (FRMSI) cache coherency protocol is explained in detail. The author's previous work around TxLinux is used for a benchmark baseline comparison as well as a starting point for implementing FRMSI.

The DATM tracks dependencies between transactions based on read or write events and in some cases forwards new (changed) values between transactions. The DATM contention manager only aborts transaction determined to not be conflict serializable based on current dependencies, rather than simply checking for any writes which may or may not be a problem; some conflicts can occur without affecting the serializability of committing transactions.

Benchmarks were done on the Simics machine simulator using STAMP, one test specific to TxLinux, and a shared counter micro-benchmark. DATM shows improvement over TxLinux's MetaTM in almost all cases. Vacation showed 20% improvement, and bayes showed 30% improvements. The new cache coherency protocol changes enable improved HTM performance in some specific cases without hurting performance, but as with any hardware transactional memory system, there is a question of whether these gains are worth the required hardware changes.

VPTM

Value-Prediction Transactional Memory [46] optimizes the performance of LogTM by predicting memory values based on recently observed values. Value prediction is based on the "stride" of changes observed for a small set of memory locations, chosen based on the contention rate between transactions. A memory location is actually a full cache-line, and the directory of predicted values is a fixed size of around

5 or 10. Predicted values are handed out speculatively to processors, and validated at commit time. Using predictions may increase the number of aborts by allowing deadlocks to occur, depending on the write order that occurs before and after a prediction, but these transactions would otherwise be aborting, so this does not hurt overall performance. This makes VPTM similar to DATM, which is also sensitive to the position and ordering of write operations within memory transactions.

Testing was done using value prediction directory sizes from 1 to 5, using the STAMP benchmark suite and the GEMS/Simics simulator. This suite was chosen because several tests have transactions that conflict on a small set of unique addresses, which makes them well suited for value prediction. LogTM was used as a baseline, and VPTM showed minor improvements for most tests. The raytrace test has 15 unique conflicting addresses and showed a 100% speedup over LogTM. Transactions in the labyrinth test conflict over only 2 unique memory addresses and showed a remarkable 1000% speedup over LogTM. Clearly VPTM is effective for certain workloads and it seems to be otherwise innocuous, ignoring the fact that it requires significant hardware-level changes.

STMlite

STMlite [38] explores software transactional memory optimized to support automatic parallelization of algorithms using 2-8 threads. This limits scalability, but allows STMlite to make a few novel optimizations. Lock contention among transactions is avoided by using a single dedicated thread as a transaction commit manager (TCM). Workers add executed transaction information into a precommit log that is validated by the TCM. Bloom-style signatures are uses to efficiently track write sets and validate read sets, and a global clock value is increments to allow read-only transactions to occur

without involving the TCM. After a commit is approved by the TCM, worker threads perform the actual memory writes in parallel with each other. The loop parallelization framework automatically converts DOALL loops into a set of parallel transactions. Single-thread execution is used for recursive loops, and all the threads for one loop must complete before another can begin.

Performance was measured using the STAMP benchmarks on an 8-core UltraSparc system and compared against TL2 as a baseline, to measure improvement. STMlite "… achieves about 2.5x and 3.1x speedup over TL2 with 8 cores, which is quite close to the speedup achieved by previous hybrid schemes" for the STAMP vacation test, which has long transactions. The authors point out that the Sparc processor shares a single floating point unit between all 8 cores, which causes other STAMP tests to show minimal improvements over TL2.

Parallelization performance was tested using SPECfp benchmarks, and compared against both TL2 and simulated HTM results. Transactional load and store calls were replaced with standard ones to provide simulated best-case hardware transactional memory performance. STMlite outperforms TL2 by up to 3x in some cases, and in most cases show similar or slightly improved performance. The simulated HTM results tended to be between 1x and 2x better than STMlite.

SwissTM

SwissTM [17] is a software transactional memory implementation focused on improving contention manager performance. SwissTM is compared in detail with RSTM, TinySTM, and TL2 to clarify the changes made and how those changes affect runtime performance. Because the focus of this work is on the final performance, a large set of benchmarks were performed using STMBench7, STAMP, Lee-TM, and a standard red-

black tree. Rather than choose either early or late detection of conflicts, SwissTM detects read-write conflicts late and write-write conflicts early. This allows reading processes time to possibly commit before the writer is affected, rather than immediately aborting. Writer-writer conflicts trigger an immediate abort rather than allow processes to continue working only to fail later at commit time. Results showed that SwissTM outperforms other STMs, with a larger improvement for a read-dominated workload, as read-write conflicts are detected later and more efficiently. SwissTM is not strongly atomic, so non-transactional processes can not safely read and write into the transactional memory space concurrently. SwissTM is also not privatization safe; object references taken (moved into process-local private space) during a transaction may still be seen by other transactions. Privitization requires a quiescence strategy, where committed threads wait for other active transactions to either abort or commit.

LATM

Lock-Aware Transactional Memory [24] allows the uses of traditional shared mutually-exclusive locks both inside of transactions (LiT) and outside of transactions (LoT). Previous work supports non-transactional locking by converting locks into transactions, but this allows deadlocks to occur. Any detected deadlocks may be broken, but this allows the possibility of an incorrect program state. LATM requires the programmer to declare which transactions may cause locking conflicts, using two different levels of granularity. The specific list of locks which may conflict can be provided, or simply a flag is used to indicate that any lock may conflict. Because transactions may abort, LATM requires that transactions only acquire locks. All locks acquired by a transaction are automatically released after the transaction commits.

Benchmark results were acquired on a Sun Fire T2000 providing 32 concurrent hardware threads, using basic hash-table and linked-list tests. With a large number of concurrent transactions, LATM performed better than using full lock protection. The level of programmer-specified lock-conflict granularity performed differently between small data-structures and larger data-structures, which reflects the amount of associated overhead. The performance numbers are not as interesting as the simple fact that LATM allows transactional memory and non-transactional code to correctly sharing a set of mutually-exclusive locks.

SkySTM

SkySTM [35] is a purely software transactional memory system that introduces new mechanisms for providing privatization efficiently. Prior STM designs provide privatization but do not scale well due to high levels of contention among global data structures. SkySTM utilizes several variations of the scalable nonzero indicator (SNZI) algorithm to greatly reduce the number of writes to global data structures and thereby reduce the amount of contention between parallel processors. This allows privatization to be provided that scales linearly up to hundreds of parallel processors.

SkySTM is benchmarked against TL2 [14] using a Sun T5440 server and four UltraSparc T2 Plus processors with a total of 32 cores and 256 hardware threads but four different shared L2 memory caches. "… while the machine supports up to 256 hardware threads, inter-thread communication overhead significantly when running more than 64 threads, as then not all threads share the same L2 cache." HashTable with a 50% read-write ratio is used to compare SkySTM against a version of TL2 modified to support privatization. "SkySTM scales almost linearly up to 256 threads, while TL2-GV4 scales only while all threads are on the same chip (i.e., up to 64 threads); beyond this point,

throughput decreases." Another optimized version of TL2 is shown to in fact scale better than SkySTM, but it does not provide privatization.

APPENDIX B - BENCHMARKS

This appendix describes the benchmark suites of interest to transactional memory.

SPLASH-2

Stanford Parallel Applications for SHared memory version two [61] was released in 1995 as an enhancement to the original SPLASH released in 1992. The newest version includes eight applications that perform ray-tracing, FFT, radix sorting, and particle system simulations. The tests are complex but scale well when running with transactional memory and multiple processors. SPLASH-2 applications have been criticized as being too parallel to be effective as a benchmark for transactional memory systems. The SPLASH-2 applications are optimized with fine-grained locking and present very little conflicts between transactions or other unusual behavior that might challenge a hybrid conflict manager or overflow a hardware based transaction log. This benchmark has been effectively superseded by the new STAMP benchmark, also from Stanford.

STAMP

The Stanford Transactional Applications for Multi-Processing [41] consists of 8 different applications designed to test the overall performance of transactional memory systems. Tests include algorithms pertinent to data mining, gene sequencing, network intrusion detection, graphs, pattern matching, and mesh network refinement. The labyrinth test is similar to the Lee-TM algorithm, but with three dimensions rather than two. The vacation test simulates customer reservations with a travel agency service. STAMP is written in C for maximum portability and there are a total of thirty options available for adjusting the runtime behavior of individual tests. Tests are categorized by

transaction length, size of the read and write set, percentage of time spent inside transactions, and the contention rate.

Baseline benchmarks were provided for six different TM implementations. Lazy-HTM emulates the TCC architecture with late detection of conflicts, while Eager-HTM follows the LogTM design with early conflict detection. Lazy-STM is a port of TL2 that was modified and also used for Eager-STM. Lazy-Hybrid follows the SigTM system that uses signatures to track read and write sets, and a modified version is used for Eager-Hybrid. To ensure a fair comparison, all benchmarks were run on the same simulator, which included support all four HTM implementations.

In most cases the hardware TM performed better than the software TM, with the hybrid designs falling somewhere in between. Unusually, HTM performed poorly on the labyrinth test due the very large dataset causing a transaction overflow. STM performed best on the bayes algorithm test, because STM tracked memory operations using a finer granularity than HTM. This benchmark suite makes it clear just how differently transactional workloads behave across a variety of implementations. This benchmark suite covers a breadth of different algorithms as well as a depth, thanks to the adjustable runtime parameters.

STMBench7

An existing object-oriented database benchmark “007” was modified to create STMBench7 [25] for use as a transactional memory benchmark, with versions available for both C++ and Java. The workload takes an object graph representing a collection of documents and performs a series of operations on that graph using transactions. The benchmark lets you adjust parameters to create work loads that are read-dominated,

write-dominated, or balanced. The length of traversals can also be adjusted, and initial testing showed longer transactions to be problematic.

> [25] Our simple ASTM-based implementation performs very poorly when long traversals are enabled—a single execution of traversal T1, for example, could last as much as half an hour (with a single thread, on the 2-cpu machine; as compared to about 1.5 s for locking).

This poor performance may make ASTM look bad, but it makes STMBench7 look good. This benchmark mimics real-world actions and has already exposed poor performance on one transactional memory implementation, so we can expect it to cause similar problems for other transactional memory systems.

WormBench

The WormBench benchmark [63] is a parameterized workload written in C#, and evaluated using an existing STM built for Bartok which compiles code to run on Microsoft's .NET Common Language Runtime (CLR). The benchmark is inspired by the classic snake game, where players steer virtual snakes around a field and the length of each snake changes over time. The goal of WormBench was to provide a single benchmark that provides different transactional characteristics based on a tunable set of parameters: how large is the field, how many worms, starting length, head size, and the set of movements performed by each worm. As a proof of concept, parameters were developed to match the runtime metrics of the STAMP genome benchmark test. WormBench provides an adjustable non-trivial benchmark suitable for testing STMs running in the Microsoft CLR environment.

Lee-TM

The Lee-TM benchmark suite [2] is based on Lee's Routing Algorithm, which automatically produces an efficient interconnect mapping between components on a

circuit board. Component locations as well as physical obstructions are provided to multiple processes. Each process searches for a path to connect components according to a layout design and adds the found path to the routing grid. Paths may not cross other paths and the shortest path found is usually preferred.

This C++ benchmark includes five different implementations of Lee's Algorithm: sequential and coarse-grained are provided as baselines. Medium-grained partitions the circuit grid into blocks for locking. A transactional version uses the medium-grained partitioning with atomic blocks, and early-release is added for the optimized transactional version. Early-release means that after a path is discovered, some data is removed from the read set before the transaction is committed. A large area may be searched to find a routing path, but only cells forming the final path need to be part of the committing memory transaction. This helps reduce the conflict between parallel transactions by reducing the size of the read set that must be verified before a commit can occur.

Three sets of layouts plans are provided for testing. *Simple* consists of 841 short routes between an evenly placed grid of components. *Main* consists of 1506 routes and *Mem* consists of 3101 routes, both representing a microcontroller. Benchmarks using RSTM2 show poor performance for the two complex layouts; both coarse-grained and medium-grained non-transactional locking had better performance using up to 8 processors. A verifier is included to check that a circuit layout is valid and ensures that the transactional memory implementation is working correctly.

> [2] Unoptimized transactional execution was, in the best case, four times slower than medium-grain locking. This result highlights the need for complex benchmarks to stress TM systems. ... The analysis identified contention management as a target of future research to make better decisions that result in less wasted work, and thus better performance.

Additional work by the same authors [3] compares benchmarks of Lee-TM against STAMP, both running on DSTM2. Detailed analysis shows that Lee-TM has a unique runtime behavior when compared to STAMP. For example, most STAMP tests show a very consistent rate of commits, but Lee-TM is shown to have an erratic instantaneous commit rate. STAMP tests show higher rates of aborts for tests that spend a larger percentage of time inside transactions, while Lee-TM spends almost all of it's time inside transactions with an unexpectedly low abort rate. Lee-TM appears both unique and effective as a complex TM benchmark.

APPENDIX C - REAL-WORLD APPLICATIONS

Several reports have been published which document how transactional memory was benchmarked using specific real-world applications rather than test suites.

ApacheSTM

Eran et al. [19] used Intel's experimental STM compiler to convert a portion of the Apache HTTP server to use software transactional memory. The Apache mod_cache module provides a hash-table shared between worker threads to cache recently delivered web pages, using LRU and a size-based greedy algorithm to decide which entries are ejected when the cache is full. Benchmarks were performed using an 8-processor quad-core Opteron system, for a total of 32 hardware cores. Two systems were used, one running the modified version of Apache and the second running Siege, an automated web testing client. Results show that when the workload is such that cached data is not used, the STM overhead is high and performance is worse than the original non-transactional implementation. When the workload can be served from the cache, STM performs only marginally better than the non-transactional version.

QuakeTM

An existing multiplayer game server was parallelized [23] using OpenMP and an extended version of McRT-STM. The authors did not intend to improve performance, but simply to show that it is possible to parallelize a large legacy software system using only the coarse-grained required by atomic blocks and transactional memory. The Lee-TM benchmark is 800 lines of code and the STMBench7 suite is 5000 lines of code. The QuakeTM server provides a larger more complex workload, with 27,600 lines of code

and “irregular parallelism and long transactions contained within eight different atomic blocks with large read and write sets.”

Performance measurements were run on a 4-processor dual-core Xenon Linux system, providing a total of 8 hardware cores. Results show that the STM overhead when executing a single thread was measured to be between 2.4 and 4.5 times that of the original sequential server using a single mutually-exclusive lock. The single lock technique does not scale with multiple threads, but transactional QuakeTM does scale as more processors are added. Unfortunately, even with 8 hardware threads of execution, QuakeTM performance results are still more than two times slower than the non-scaling sequential global-lock version. The authors point out that the very large transaction size causes a high percentage of work to be wasted due to aborted transactions. Measurements found the mean amount of data read by a transaction is 5.1KB, but reaches up to 1.7MB in some cases. “This leads us to conclude that a coarse-grained approach is not a viable option for the current STM systems. Moreover, we have shown that the read and write set sizes are significant, which could impose serious problems for hardware TM systems.”

The Rock

Sun developed a multicore SPARC processor, code-named Rock, which provides best-effort hardware transactional memory. Testing was done [15] using simulators as well as two pre-production versions of the chip. Hardware instructions are provided to start and stop transactions as well as specify a handler that is called when a failure occurs. The Rock CPU provides 16 cores that can be set to operate in two different modes. The Scout Execution (SE) mode uses two threads per core, for 32 hardware threads of execution total and memory transaction store logs able to hold up to 16 writes. Simultaneous Scout Execution (SSE) mode provides only 16 hardware threads, but the

store log is able to hold up to 32 writes. All testing was done using a single-CPU system and focused on the SSE mode.

Benchmarks were run on hash-tables and red-black trees using HyTM and PhTM to test hardware transactional memory, TL2 to represent STM performance, and a single coarse-grained lock as a baseline. These tests performed short and simple operations, to ensure that transactions could execute without overflowing the relatively limited hardware buffers. In contract to the single coarse-grained lock, each transactional memory implementation scaled up evenly on the hash-table test. With a small key range, PhTM throughput was a factor of 54 times higher than the single lock, 4 times higher than TL2, and 2 times higher than HyTM. With a key range 500 times larger, PhTM performed 20 times better the single lock, 2.4 times better than TL2, and only 1.2 times better than HyTM. Testing small red-black trees showed similar results, but larger red-black trees caused PhTM to perform worse than TL2. Detailed analysis explains that certain larger data structures are simply more suited for execution by STM rather that PhTM. Vector, hash-table, and a minimum spanning forest algorithm were also tested using transactional lock elision (TLE), which speculatively elides locks at runtime with rollback-retires when conflicts occur.

TxLinux 2.4

The Linux 2.4 kernel was converted to use transactional memory [32] and benchmarked against the standard Linux 2.6 kernel. The TM implementation is based on MetaTM, previously developed by the authors when they converted the Linux 2.6 kernel to use transactional memory. The Linux 2.6 kernel has more complex fine-grained locking, but the Linux 2.4 kernel uses coarse-grained locking in much the same way real-world applications are expected to be able to utilize transactional memory.

> [32] Although bad for performance, coarse-grained locking requires less programming effort to resolve complicated issues such as deadlocks and determining which locks are required to modify particular data structures. With TxLinux 2.4, we try to use HTM to turn the coarse-grained locking vice into a virtue. If coarse locks guard distinct data, then HTM should be able to achieve the synchronization performance of 2.6 without the complexity associated with fine-grained locking.

Only a minimal set of HTM optimization features are provided (simulated) to show that "more baroque" designs are not necessary to achieve sufficient performance benefits. When transactions overflow in TxLinux they are executed using STM, with some special handling to ensure that proper kernel locking is maintained. For overflows that occur in user-space transactions, software retries must block any other concurrent hardware transactions from committing, in order to ensure consistency between HTM and STM transactions. This means that a high overflow rate could result in poor performance.

Benchmarks were taken from previous TxLinux 2.6 work, and consisted of several non-transactional user-mode applications, designed to thoroughly exercise the kernel synchronization hot-spots. A subset of STAMP benchmarks were also used to evaluate the overflow performance of user programs. The Simics simulator is used to run the tests, with either 8, 16, or 32 virtual x86 processors. None of the benchmarks cause an overflow rate greater than one percent, and most results show that TxLinux 2.4 executes tests faster than the standard non-transactional Linux 2.4 kernel. Benchmarks were also run on the simulator using TL2 to compare pure STM performance and show that TxLinux 2.4 performs better than TL2 but worse than the newer Linux 2.6 kernel.

References

[1] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. 2005. Unbounded Transactional Memory. In *Proceedings of the 11th international Symposium on High-Performance Computer Architecture* (February 12-16, 2005). IEEE Computer Society, Washington, DC, 316-327.

[2] Ansari, M., Jarvis, K., Kotselidis, C., Lujan, M., Kirkham, C., Watson, I. Profiling Transactional Memory Applications, in *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, pp. 11-20, 2009.

[3] Ansari, M., Kotselidis, C., Watson, I., Kirkham, C., Luján, M., and Jarvis, K. 2008. Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory. In *Proceedings of the 8th international Conference on Algorithms and Architectures For Parallel Processing* (Agia Napa, Cyprus, June 9-11, 2008). A. G. Bourgeois and S. Q. Zheng, Eds. Lecture Notes in Computer Science, vol. 5022. Springer-Verlag, Berlin, Heidelberg, 196-207.

[4] Azul Systems, Inc. (January 2006) Optimistic Thread Concurrency. White paper.

[5] Baugh, L., Neelakantam, N., and Zilles, C. 2008. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. *SIGARCH Comput. Archit. News* 36, 3 (Jun. 2008), 115-126.

[6] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. ACM Symposium on OS Principles*, Big Sky, MT, USA, Oct. 2009.

[7] Bobba, J., Goyal, N., Hill, M. D., Swift, M. M., and Wood, D. A. 2008. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proceedings of the 35th international Symposium on Computer Architecture* (June 21-25, 2008). IEEE Computer Society, Washington, DC, 127-138.

[8] Brooks, F. P. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* 20, 4 (Apr. 1987), 10-19.

[9] Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S. 2008. Software transactional memory: why is it only a research toy? *Commun. ACM* 51, 11 (Nov. 2008), 40-46.

[10] Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Van Biesbrouck, M., Pokam, G., Calder, B., and Colavin, O. 2006. Unbounded page-based transactional memory. *ACM SIGPLAN Notices*. 41, 11 (Nov. 2006), 347-358.

[11] Chung, J., Minh, C. C., McDonald, A., Skare, T., Chafi, H., Carlstrom, B. D., Kozyrakis, C., and Olukotun, K. 2006. Tradeoffs in transactional memory virtualization. *ACM SIGPLAN Notices*. 41, 11 (Nov. 2006), 371-381.

[12] Click, C. A lock-free hashtable. JavaOne Conference. May 2007.

[13] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. 2006. Hybrid transactional memory. *ACM SIGPLAN Notices.* 41, 11 (Nov. 2006), 336-346.

[14] Dice, D., Shalev, O., and Shavit, N. 2006. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing* (DISC 06), Stockholm, Sweden, pp.194—208.

[15] Dice, D., Lev, Y., Moir, M., and Nussbaum, D. 2009. Early experience with a commercial hardware transactional memory implementation. In Proceeding of the 14th international Conference on Architectural Support for Programming Languages and Operating Systems (Washington, DC, USA, March 7-11, 2009). ACM, New York, NY, 157-168.

[16] Dolev, S., Hendler, D., and Suissa, A. 2008. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing* (Toronto, Canada, August 18-21, 2008). ACM, New York, NY, 125-134.

[17] Dragojević, A., Guerraoui, R., and Kapalka, M. 2009. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 15-21, 2009). ACM, New York, NY, 155-165.

[18] Ellen, F., Lev, Y., Luchangco, V., and Moir, M. 2007. SNZI: scalable NonZero indicators. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA, August 12-15, 2007). ACM, New York, NY, 13-22.

[19] Eran, H., Lutzky, O., Guz, Z., and Keidar, I. 2009. Transactifying Apache's cache module. In *Proceedings of SYSTOR 2009: the Israeli Experimental Systems Conference* (Haifa, Israel). SYSTOR '09. ACM, New York, NY, 1-9.

[20] Felber, P., Fetzer, C., and Riegel, T. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming (Salt Lake City, UT, USA, February 20-23, 2008). ACM, New York, NY, 237-246.

[21] Fraser, K. Practical lock-freedom. PhD thesis, University of Cambridge Computer Laboratory, Feb. 2004. Also published as UCAM-CL-TR-579.

[22] Fraser, K. and Harris, T. 2007. Concurrent programming without locks. In *ACM Trans. Comput. Syst.* 25, 2 (May. 2007), 5.

[23] Gajinov, V., Zyulkyarov, F., Unsal, O. S., Cristal, A., Ayguade, E., Harris, T., and Valero, M. 2009. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international Conference on Supercomputing* (Yorktown Heights, NY, USA, June 8-12, 2009). ACM, New York, NY, 126-135.

[24] Gottschlich, J. E., Siek, J. G., Vachharajani, M., Winkler, D. Y., and Connors, D. A. 2009. An efficient lock-aware transactional memory implementation. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy, July 6-6, 2009). ACM, New York, NY, 10-17..

[25] Guerraoui, R., Kapalka, M., and Vitek, J. 2007. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 315-324.

[26] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. 2004. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual international Symposium on Computer Architecture* (München, Germany, June 19-23, 2004). IEEE Computer Society, Washington, DC, 102.

[27] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. 2005. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA, June 15-17, 2005). ACM, New York, NY, 48-60.

[28] Harris, T., Peyton-Jones, S. Transactional memory with data invariants. In Proc. The First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT‘06), Ottawa, Canada, Jun. 2006.

[29] Herlihy, M. and Moss, J. E. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual international Symposium on Computer Architecture* (San Diego, California, United States, May 16-19, 1993). ACM, New York, NY, 289-300.

[30] Herlihy, M. SXM: C# Software Transactional Memory. Unpublished manuscript, Brown University, May 2005.

[31] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing* (Boston, Massachusetts, July 13-16, 2003). ACM, New York, NY, 92-101.

[32] Hofmann, O. S., Rossbach, C. J., and Witchel, E. 2009. Maximum benefit from a minimal HTM. In *Proceeding of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems* (Washington, DC, USA, March 7-11, 2009). ACM, New York, NY, 145-156.

[33] Kumar, S., Chu, M., Hughes, C. J., Kundu, P., and Nguyen, A. 2006. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA, March 29-31, 2006). ACM, New York, NY, 209-220.

[34] Larus, J. and Rajwar, R. 2007 Transactional Memory (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers.

[35] Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D. and Olszewski, M. Anatomy of a scalable software transactional memory. In *TRANSACT '09: Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[36] Lev, Y., Moir, M., and Nussbaum, D. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT-II)*, 2007.

[37] Marathe, V. J., Scherer, W. N., and Scott, M. L. Adaptive software transactional memory. Technical report 868. Computer Science Department, University of Rochester, 2005.

[38] Mehrara, M., Hao, J., Hsu, P., and Mahlke, S. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 15-21, 2009). ACM, New York, NY, 166-176.

[39] Mellor-Crummey, J. M. and Scott, M. L., Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, in *ACM Trans. on Computer Systems*, Feb 1991.

[40] Michael, M. M. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada, August 10-13, 2002). ACM, New York, NY, 73-82.

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

[42] Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual international Symposium on Computer Architecture* (San Diego, CA, USA, June 9-13, 2007). ACM, New York, NY, 69-80.

[43] Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., and Wood, D.A. LogTM: Log-based transactional memory. In *Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[44] Morgan, T. P. (2009, Sep 11). Sun's Sparc server roadmap revealed. Retrieved from The Register website:
http://www.theregister.co.uk/2009/09/11/sun_sparc_roadmap_revealed/

[45] Olszewski, M., Cutler, J., and Steffan, J. G. 2007. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th international Conference on Parallel Architecture and Compilation Techniques* (September 15-19, 2007). IEEE Computer Society, Washington, DC, 365-375.

[46] Pant, S. M. and Byrd, G. T. 2009. Extending concurrency of transactional memory programs by using value prediction. In *Proceedings of the 6th ACM Conference on Computing Frontiers* (Ischia, Italy, May 18-20, 2009). ACM, New York, NY, 11-20.

[47] Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M. 2008. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers* (Ischia, Italy, May 5-7, 2008). ACM, New York, NY, 67-78.

[48] Peyton Jones, S., Beautiful Concurrency, in *Beautiful Code*, edited by Andy Oram, Greg Wilson, O'Reilly, 2007. ISBN 0-596-51004-7.

[49] Rajwar, R. and Goodman, J. R. 2001. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE international Symposium on Microarchitecture* (Austin, Texas, December 1-5, 2001). IEEE Computer Society, Washington, DC, 294-305.

[50] Rajwar, R. and Goodman, J. R. 2002. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international Conference on Architectural Support For Programming Languages and Operating Systems* (San Jose, California, October 5-9, 2002). ACM, New York, NY, 5-17.

[51] Rajwar, R.; Herlihy, M.; Lai, K., Virtualizing transactional memory, In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pp. 494-505, 4-8 June 2005.

[52] Ramadan, H. E., Rossbach, C. J., and Witchel, E. 2008. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 2008 41st IEEE/ACM international Symposium on Microarchitecture* (November 8-12, 2008). IEEE Computer Society, Washington, DC, 246-257.

[53] Rochester Software Transactional Memory, RSTM. http://www.cs.rochester.edu/research/synchronization/rstm/

[54] Shriraman, A., Spear, M. F., Hossain, H., Marathe, V. J., Dwarkadas, S., and Scott, M. L. 2007. An integrated hardware-software approach to flexible transactional memory. *SIGARCH Comput. Archit. News* 35, 2 (Jun. 2007), 104-115.

[55] Saha, B., Adl-Tabatabai, A., and Jacobson, Q. 2006. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th Annual IEEE/ACM international Symposium on Microarchitecture* (December 9-13, 2006). IEEE Computer Society, Washington, DC, 185-196.

[56] Shalev, O. and Shavit, N. 2006. Split-ordered lists: Lock-free extensible hash tables. J. ACM 53, 3 (May. 2006), 379-405.

[57] Saha, B., Adl-Tabatabai, A., Hudson, R. L., Minh, C., and Hertzberg, B. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, New York, USA, March 29-31, 2006). ACM, New York, NY, 187-197.

[58] Shriraman, A., Dwarkadas, S., and Scott, M. L. 2008. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th international Symposium on Computer Architecture* (June 21-25, 2008). IEEE Computer Society, Washington, DC, 139-150.

[59] Spear, M. F., Michael, M. M., and von Praun, C. 2008. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany, June 14-16, 2008). ACM, New York, NY, 275-284.

[60] Stone, J. M., Stone, H. S., Heidelberger, P., and Turek, J. 1993. Multiple Reservations and the Oklahoma Update. In *Parallel & Distributed Technology: Systems & Applications*, IEEE, vol.1, no.4, pp.58-71, Nov 1993.

[61] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual international Symposium on Computer Architecture* (S. Margherita Ligure, Italy, June 22 - 24, 1995). ISCA '95. ACM, New York, NY, 24-36.

[62] Yen, L.; Bobba, J.; Marty, M.R.; Moore, K.E.; Volos, H.; Hill, M.D.; Swift, M.M.; Wood, D.A., "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," High Performance Computer Architecture, 2007. IEEE 13th International Symposium on, pp.261-272, 10-14 Feb. 2007.

[63] Zyulkyarov, F., Cristal, A., Cvijic, S., Ayguade, E., Valero, M., Unsal, O., and Harris, T. 2008. WormBench: a configurable workload for evaluating transactional memory systems. In *Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture* (Toronto, Canada, October 26 - 26, 2008). MEDEA '08, vol. 310. ACM, New York, NY, 61-68.

Vita

Thomas Hughes attended UT Austin between 1990 and 1991, but dropped out and spent more than a decade working for various small software companies. He eventually returned to school and received his Bachelors of Science in Software Engineering from UT Dallas in 2005. While still working, he entered the UT Austin Masters of Science and Engineering program for Software Engineering in 2008.

Permanent email: tfh2000@gmail.com

This report was typed by the author.