Early Experience with a Commercial Hardware Transactional Memory Implementation

Dave Dice
Sun Microsystems Laboratories
dave.dice@sun.com

Yossi Lev
Brown University and
Sun Microsystems Laboratories
levyossi@cs.brown.edu

Mark Moir and Dan Nussbaum
Sun Microsystems Laboratories
{mark.moir,dan.nussbam}@sun.com

Abstract

We report on our experience with the hardware transactional memory (HTM) feature of two pre-production revisions of a new commercial multicore processor. Our experience includes a number of promising results using HTM to improve performance in a variety of contexts, and also identifies some ways in which the feature could be improved to make it even better. We give detailed accounts of our experiences, sharing techniques we used to achieve the results we have, as well as describing challenges we faced in doing so.

Categories and Subject Descriptors C.1.4 [*Hardware*]: Parallel Architectures; D.1.3 [*Software*]: Concurrent Programming—Parallel Programming

General Terms Design, Experimentation, Performance.

Keywords Hardware, transactional memory, synchronization.

1. Introduction

The "multicore revoluton" occuring in the computing industry brings many benefits in reducing costs of power, cooling, administration, and machine room real estate, but it also brings some unprecedented challenges. Developers can no longer hide the cost of new features by relying on next year's processors to run their single-threaded code twice as fast. Instead, for an application to take advantage of advances in technology, it must be able to effectively exploit more cores as they become available. This is often surprisingly difficult.

A key factor is the difficulty of reasoning about many things happening at the same time. The traditional approach to dealing with this problem is to use locks to make certain critical sections of code execute without interference from other cores, but this approach entails difficult tradeoffs: simple approaches quickly develop bottlenecks that prevent the application from taking advantage of additional cores, while more complex ones are error prone and difficult to understand, maintain, and extend.

Transactional Memory (TM) (9) has received a great deal of research attention in recent years as a promising technology for alleviating the difficulty of writing multithreaded code that is scalable, efficient, and correct. The essence of TM is the ability to ensure that multiple memory accesses can be done "atomically", so that the programmer does not have to think about these acceses being interleaved with those of another thread. Using such an approach, the programmer specifies *what* should be done atomically, leaving the system to determine *how* this is achieved. This relieves the programmer of the burden of worrying about locking conventions, deadlock, etc.

TM-related techniques have been proposed in many contexts, ranging from the original "bounded" HTM of Herlihy and Moss (9), to a number of "unbounded" HTM proposals (1; 21; 23), numerous software transactional memory (STM) approaches (7; 8; 15), and some approaches that combine hardware and software in various ways (4; 22; 12; 2).

Proposals for unbounded HTM implementations are incomplete and too complex and risky to appear in commercial processors in the near future. Substantial progress has been made in improving STM designs in recent years, and robust and practical systems are emerging. But implementing TM in software entails significant overhead, and there is growing interest in hardware support to improve its performance.

If programmed directly, bounded HTM implementations impose unreasonable constraints on programmers, who must ensure that a transaction does not access more than a fixed, architecture-specific number of cache lines. However, it has been shown (4; 13; 17) that such implementations can be useful nonetheless by combining them with software that can exploit HTM to improve performance, but does not depend on any particular hardware transaction succeeding.

Because such techniques do not depend on any particular transaction succeeding, they can be used with *best effort*

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.
AC M 978-1-60558-406-5/09/03.

HTM, which differs from bounded HTM in that it may commit transactions that are much larger than a bounded HTM feature would, but it is also not required to guarantee to commit all transactions up to a certain size. This added flexibility considerably simplifies the task of integrating HTM into a commercial processor, because the processor can respond to difficult events and conditions by simply aborting the transaction. Furthermore, because the software is configured to take advantage of whichever HTM transactons that commit successfully, it can automatically adapt to take advantage of future improvements to the HTM feature.

Sun's architects (3) have built a multicore processor—code named *Rock*—that supports a form of best-effort hardware transactional memory. This paper reports on our recent experience experimenting with this HTM feature. The work described in this paper involved two pre-production revisions of the Rock chip: the first, which we'll call R1, was the first revision that included the HTM feature and the second—R2–was the subsequent revision, which included changes made in response to our feedback on the first.

In each case, our first priority was to test that the HTM feature worked as expected and gave correct results. We developed some specific tests to evaluate the functionality, and we have also built consistency checks into all of the benchmarks discussed in this paper. In addition to our testing, Rock's HTM feature has been tested using a variety of tools that generate random transactions and test for correct behavior, for example TSOTool (14). This extensive testing has not revealed any correctness problems.

Next, we experimented with using the HTM feature. We used the benchmarks from our work with the ATMTP simulator (5), and added a couple more, including a real application. As with ATMTP, we achieved some encouraging results but also encountered some challenges. One set of challenges with R1 was that Rock provided identical feedback about why a transaction failed in different situations that required different responses. Based on our feedback, the Rock architects refined feedback about transaction failures in the second revision R2. Our work with R2 began by repeating the evaluation we had done with R1 and examining these changes, which were found to behave as expected.

Because of the timing of our work relative to the various paper deadlines, our work with the second revision is less mature than our work with the first. In particular, our work with R2 was with an early-run chip, which was not rated for full-speed execution. We do not expect our results to change qualitatively when we are able to run on a full-speed chip. There are also a number of areas where we have not yet had time to investigate as fully as we would like.

Roadmap We begin with some background about Rock in Section 2. Then, in Section 3 we describe the tests we used to determine whether transactions succeed and fail in the cases we expect, and also what feedback Rock gives when a transaction fails. In Sections 4 through 8, we present results from our experiments using HTM in a number of contexts.

We use HTM to implement simple operations such as incrementing a counter and a double compare-and-swap (DCAS); we then use the DCAS to reimplement some components of the Java™ concurrency libraries. Next, we experiment with transactional hash tables and red-black trees using a compiler and library supporting Hybrid TM (HyTM) (4) and Phased TM (PhTM) (13); these use HTM to boost performance, but transparently revert to software if unsuccessful. We then experiment with Transactional Lock Elision (TLE), using HTM to execute lock-based critical sections in parallel if they do not conflict; we have experimented with this technique through simple wrappers in C and C++ programs, as well as with standard lock-based Java code using a modified JVM. In particular, in Section 8, we report on our success using Rock's HTM feature to accelerate a parallel Minimum Spanning Forest algorithm due to Kang and Bader (10). Discussion and conclusions are in Sections 9 and 10.

2. Background

Rock (3) is a multicore SPARC® processor that uses aggressive speculation to provide high single-thread performance in addition to high system throughput. For example, on load misses, Rock runs ahead speculatively in order to issue subsequent memory requests early.

Speculation is enabled by a checkpoint architecture: Before speculating, Rock checkpoints the architectural state of the processor. While speculating, Rock ensures that effects of the speculative execution are not observed by other threads (for example, speculative stores are gated in a store buffer until the stores can be safely committed to memory). The hardware can revert back to the previous checkpoint and re-execute from there—perhaps in a more conservative mode—if the speculation turns out to have taken a wrong path, if some hardware resource is exhausted, or if an exception or other uncommon event occurs during speculation.

Rock uses these same mechanisms to implement a best-effort HTM feature: it provides new instructions that allow user code to specify when to begin and end speculation, and ensures that the section of code between these instructions executes atomically and in its entirety, or not at all.

The new instructions are called `chkpt` and `commit`. The `chkpt` instruction provides a pc-relative *fail address*; if the transaction started by the instruction aborts for any reason, control resumes at this fail address, and any instructions executed since the `chkpt` instruction do not take effect. Aborts can be explicitly caused by software, which is important for many of the uses described in this paper. By convention we use the following unconditional trap instruction for this purpose: `ta %xcc, %g0 + 15`.

When a transaction aborts, feedback about the cause of the abort is provided in the CPS (Checkpoint Status) register, which has the same bits as described by Moir et al. (16), plus

three additional bits that are not supported by their simulator. The full set of CPS bits is shown in Table 1, along with examples of reasons the bits might be set. We discuss the CPS register in more detail in the next section.

Each Rock chip has 16 cores, each capable of executing two threads, for a total of 32 threads in the default *Scout Execution* (SE) mode (3). It can also be configured to dedicate the resources of both hardware threads in a core to a single software thread. This allows a more aggressive form of speculation, called *Simultaneous Scout Execution* (SSE), in which one hardware thread can continue to fetch new instructions while the other replays instructions that have been deferred while waiting for a high-latency event such as a cache miss to be resolved. (Rock has a "deferred queue" in which speculatively executed instructions that depend on loads that miss in the cache are held pending the cache fill; if the number of deferred instructions exceeds the size of this queue, the transaction fails.) In addition to providing more parallelism, SSE mode also allows some resources of the two hardware threads to be combined into one larger resource. For example, the store buffer accommodates up to 16 stores in SE mode, but this is increased to 32 in SSE mode.

All data presented in this paper is taken on a single-chip Rock system running in SSE mode; we have briefly explored SE mode and we discuss preliminary observations as appropriate. We have not yet had an opportunity to experiment with a multi-chip system, but we hope to do so soon.

3. Examining the CPS register

Our `cpstest` is designed to confirm our understanding of the circumstances under which transactions abort, and the feedback given by the CPS register when they do. It attempts to execute various unsupported instructions in transactions, as well as synthesizing conditions such as dereferencing a null, invalid, or misaligned pointer, an infinite loop, various trap and conditional trap instructions, etc. Rather than reporting all of the outcomes here, we plan to open source it, so that others may examine it in detail. Below we mention only a few observations of interest.

First, it is important to understand that a failing transaction can set multiple bits in the CPS register, and furthermore that some bits can be set for any of several reasons; Table 1 lists *one example* reason for each bit, and is not intended to be exhaustive. As part of our evaluation of R1, we worked with the Rock architects to compile such an exhaustive list, and together with output from `cpstest`, we identified several cases in which different failure reasons requiring different responses resulted in identical feedback in the CPS register, making it impossible to construct intelligent software for reacting to transaction failures. As a result, changes were made for R2 to disambiguate such cases. The observations below are all current as of R2, and we do not anticipate more changes at this time.

save-restore Rock fails transactions that execute a `save` instruction and subsequently execute a `restore` instruction, setting CPS to 0x8 = INST. This pattern is commonly associated with function calls; we discuss this issue further in Sections 6 and 7.

tlb misses To test the effect of DTLB misses on transactions, we re-`mmap` the memory to be accessed by a transaction before executing it. This has the effect of removing any TLB mappings for that memory. When we load from an address that has no TLB mapping, the transaction fails with CPS set to 0x90 = LD—PREC. When we store to such an address, it fails with CPS set to 0x100 = ST. This is discussed further below. To test the effects of ITLB misses on transactions, we copied code to `mmap`ed memory and then attempted to execute it within a transaction. When there was no ITLB mapping present, the transaction failed setting CPS to 0x10 = PREC.

eviction This test performs a sequence of loads at cache-line stride. The sequence is long enough that the loaded cache lines cannot all reside in the L1 cache together, which means these transactions can never succeed. We usually observe CPS values of 0x80 = LD and 0x40 = SIZ. The former value indicates that the transaction displaced a transactionally marked cache line from the L1 cache. The latter indicates that too many instructions were deferred due to cache misses. This test also occasionally yields a CPS value of 0x1 = EXOG. This happens for example if a context switch happens after the transaction fails and before the thread reads the CPS register.

cache set test This test performs loads to five different addresses that map to the same 4-way L1 cache set. Almost all transactions in this test fail with CPS set to 0x80 = LD (we discuss this further below). We also see occasional instances of EXOG, as discussed above. More interestingly, we also sometimes see the COH bit set in this test. We were puzzled at first by this, as we did not understand how a read-only, single-threaded test could fail due to coherence. It turns out that the COH bit is set when another thread displaces something from the L2 cache that has been read by a transaction; this results in invalidating a transactionally marked line in the L1 cache, and hence the report of "coherence". Even though there were no other threads in the test, the operating system's idle loop running on a hardware strand that shares an L2 cache with the one executing the transaction does cause such invalidations. We changed our test to run "spinner" threads on all idle strands, and the rate of COH aborts in this test dropped almost to zero.

overflow In this test, we performed stores to 33 different cache lines. Because Rock transactions are limited by the size of the store queue, which is 32 entries in the configuration we report on, all such transactions fail. They fail with CPS set to 0x100 = ST if there are no

Mask	Name	Description and example cause
0x001	EXOG	**Exogenous** - Intervening code has run: cps register contents are invalid.
0x002	COH	**Coherence** - Conflicting memory operation.
0x004	TCC	**Trap Instruction** - A trap instruction evaluates to "taken".
0x008	INST	**Unsupported Instruction** - Instruction not supported inside transactions.
0x010	PREC	**Precise Exception** - Execution generated a precise exception.
0x020	ASYNC	**Async** - Received an asynchronous interrupt.
0x040	SIZ	**Size** - Transaction write set exceeded the size of the store queue.
0x080	LD	**Load** - Cache line in read set evicted by transaction.
0x100	ST	**Store** - Data TLB miss on a store.
0x200	CTI	**Control transfer** - Mispredicted branch.
0x400	FP	**Floating point** - Divide instruction.
0x800	UCTI	**Unresolved control transfer** - branch executed without resolving load on which it depends

Table 1. cps register: bit definitions and example failure reasons that set them.

TLB mappings (see above) and with CPS set to 0x140 = ST|SIZ if we "warm" the TLB first. A good way to warm the TLB is to perform a "dummy" compare-and-swap (CAS) to a memory locations on each page that may be accessed by the transaction: we attempt to change the location from zero to zero using CAS. This has the effect of establishing a TLB mapping and making the page writable, but without modifying the data.

coherence This test is similar to the overflow test above, except that we perform only 16 stores, not 33, and therefore the transactions do not fail due to overflowing the store queue, which comprises two banks of 16 entries in the test configuration. All threads store to the same set of locations. Single threaded, almost all transactions succeed, with the usual smattering of EXOG failures. As we increase the number of threads, of course all transactions conflict with each other, and because we make no attempt to back off before retrying in this test, the success rate is very low by the time we have 16 threads. Almost all CPS values are 0x2 = COH. The point of this test was to understand the behavior, not to make it better, so we did not experiment with backoff or other mechanisms to improve the success rate; we left this for the more realistic workloads discussed in the remainder of the paper.

3.1 Discussion

Even after R2 changes to disambiguate some failure cases, it can be challenging in some cases to determine the reason for transaction failure, and to decide how to react. For example, if the ST bit (alone) is set, this may be because the address for a store instruction is unavailable due to an outstanding load miss, or because of a micro-TLB miss (see (3) for more details of the Rock's MMU).

In the first case, retrying may succeed because the cache miss will be resolved. In the latter case, an MMU request is generated by the failing transaction, so the transaction may succeed if retried because a micro-TLB mapping is established from higher levels of the MMU. However, if no mapping for the data in question is available in any level of the MMU, the transaction will fail repeatedly unless software can successfully warm the TLB, as described above.

Thus, the best strategy for a transaction that fails with CPS value ST is to retry a small number of times, and then retry again after performing TLB warmup if feasible in the current context, and to give up otherwise. The optimal value of the "small number" depends on the feasibility and cost of performing TLB warmup in the given context.

One interesting bit in the CPS register is the UCTI bit, which was added as a result of our evaluation of R1. We found that in some cases we were seeing values in the CPS register that indicated failure reasons we thought could not occur in the transactions in question. We eventually realized that it was possible for a transaction to misspeculate by executing a branch that has been mispredicted before the load on which the branch depends is resolved. As a result, software would react to a failure reason that was in some sense invalid. For example, it might conclude that it must give up due to executing an unsupported instruction when in fact it would likely succeed if retried because the load on which the mispredicted branch depended would be resolved by then, so the code with the unsupported instruction would not be executed next time. Therefore, the UCTI bit was added to indicate that a branch was executed when the load on which it depends was not resolved. Software can then retry when it sees UCTI set, hoping that either the transaction will succeed, or at least that feedback about subsequent failures would not be misleading due to misspeculation.

We discuss these and other challenges that have arisen from certain causes of transaction failure and/or feedback software receives about them throughout the paper. Designers of future HTM features should bear in mind not only the quality of feedback about reasons for transaction failure but also how software can react to such failures and feedback.

4. Simple, static transactions

In this section, we briefly summarize our experience with simple, static transactions; more details appear in (6). For such transactions, the code and memory locations to be accessed are known in advance, and we can exploit this information to make transactions highly likely to succeed. For example, we can align transaction code to avoid ITLB misses during the transaction. Similarly, we can warm the DTLB outside the transaction as described in Section 3.

To date, we have experimented with a simple counter and with a DCAS (double compare-and-swap) operation, which generalizes the well known CAS operation to two locations.

For the counter, we compared CAS-based and HTM-based implementations, each with and without backoff (backing off in reaction to CAS failure in the former case, and to the COH bit being set in the latter).

As expected for a highly contended, centralized data structure, all methods showed some degradation in throughput with increasing number of threads. All performed comparably, except the HTM version with no backoff, for which degradation was so severe as to suggest livelock. This is not surprising given Rock's simple "requester wins" conflict resolution policy: requests for transactionally marked cache lines are honored immediately, failing the transaction. We have found simple software backoff mechanisms to be effective to avoid this problem. Nonetheless we think designers of future HTM features should consider whether simple conflict resoluton policies that avoid failing transactions as soon as a conflict arises might result in better and more stable performance under contention.

We also used HTM-based DCAS operations to reimplement two concurrent set implementations from the `java.util.concurrent` library. Our results were fairly similar to those achieved on the ATMTP simulator (5). Briefly, the new algorithms match the performance of the state-of-the-art, carefully hand-crafted implementations in `java.util.concurrent`; see (6) for details. We believe that this approach has strong potential to simplify and improve Java libraries and other libraries.

5. Hash table

The simple hashtable experiment discussed in this section was designed to allow us to evaluate various TM systems under low and high contention. The hash table consists of a large number (2^{17}) of buckets and our experimental test harness allows us to restrict the range of keys inserted into the table. With such a large table, we rarely encounter buckets with multiple keys in them, so we can concentrate on the common simple case. This test has been useful in evaluating the scalability of various STMs, and also supports some interesting observations using Rock's HTM.

The hashtable is implemented in C++, using a compiler and library that can support HyTM (4) and PhTM (13), as well as several STMs. Because the hashtable operations are short and simple, we should be able to get them to succeed as hardware transactions. Both HyTM and PhTM allow an operation to be executed using a hardware transaction, but can also use STM to complete an operation if it cannot succeed in a hardware transaction. All decisions about whether to retry, back off, or switch to using STM are made by the library and are transparent to the programmer, who only writes simple C++ code for the hash table operations.

Figure 1 shows our results from experiments with 50% inserts/50% deletes, for key ranges of (a) 256 and (b) 128,000. In each case, we prepopulate the hash table to contain about half of the keys, and then measure the time taken for the threads to complete 1,000,000 operations each, chosen at random according to the specified operation distributions and key ranges. An "unsuccessful" operation (insert of a value already in the hash table, or delete of one not there) does not modify memory; thus approximately 50% of operations modify memory in this experiment. In this and other similar graphs, `hytm` and `phtm` refer to HyTM and PhTM using our SkySTM algorithm (11) for the STM component, and `stm` is SkySTM with no hardware support. We present results as throughput in total operations per microsecond.

For both scenarios, we observe high hardware transaction success rates for both HyTM and for PhTM (regardless of which STM is used); in fact, almost all operations eventually succeed as hardware transactions, and do not need to revert to using the STM. Figure 1 shows that these methods clearly outperform all software-only methods. For scenario (a), at 16 threads, the two PhTM variants outperform the single lock implementation by a factor of about 54 and the state-of-the-art TL2 STM by a factor of about 4.6. HyTM also does well in this scenario, although it trails the PhTM variants by about a factor of two.

Scenario (b) yields qualitatively similar results to scenario (a), with HyTM and PhTM successfully outperforming all software-only methods (except single threaded, as discussed below). In this case, the quantitative benefit over the software-only methods is less—the two PhTM variants performing "only" about 20 times better than a single lock and about 2.4 times better than TL2 at 16 threads, with HyTM trailing the PhTM variants by only a factor of 1.2 or so. This is because the key range is large enough that the active part of the hash table does not fit in the L1 cache, so all methods suffer the cost of the resulting cache misses, which serves to level the playing field to some extent. We also observed that, with 100% lookup operations with a key range of 256, PhTM outperforms the lock at 16 threads by a factor of about 85 and TL2 by a factor of about 3.4, with HyTM trailing PhTM by a factor of about 1.2 (data not shown).

Examining some of the statistics collected by the PhTM library yielded some interesting observations that give some insight into the reasons for transactions failing on Rock. For example, with the 128,000 key range experiment (scenario (b)), more than half of the hardware transactions are retries,

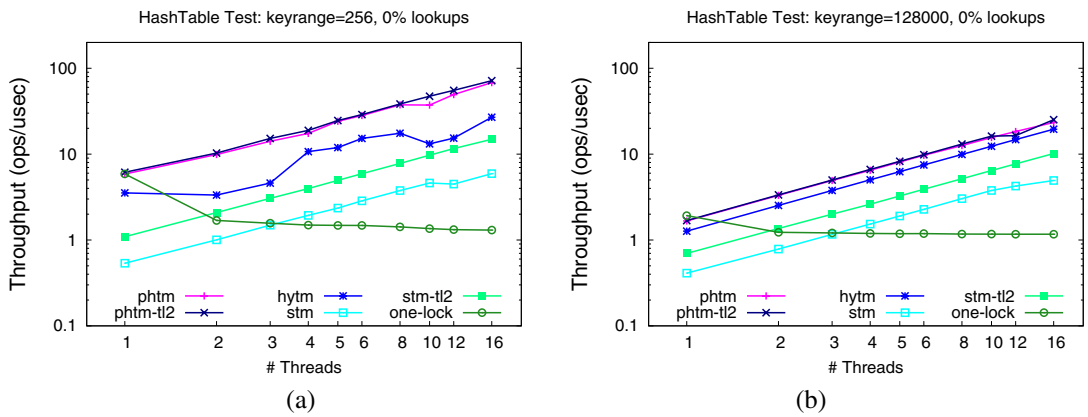

Figure 1. HashTable with 50% inserts, 50% deletes: (a) key range 256 (b) key range 128,000.

even for the single thread case (these retries explain why the single lock outperforms HyTM and PhTM somewhat in the single threaded case). In contrast, with the 256 key range experiment (scenario (a)), only 0.02% of hardware transactions are retries in the single thread case, and even at 16 threads only 16% are retries.

Furthermore, the distribution of CPS values from failed transactions in the 16 thread, 256 key range case is dominated by COH while in the 128,000 key range case it is dominated by ST and CTI. This makes sense because there is more contention in the smaller key range case (resulting in the CPS register being set to COH), and worse locality in the larger one. Poor locality can cause transactions to fail for a variety of reasons, including micro-DTLB mappings that need to be reestablished (resulting in ST), and mispredicted branches (resulting in CTI).

Finally, this experiment and the Red-Black Tree experiment (see Section 6) highlighted the possibility of the code in the fail-retry path interfering with subsequent retry attempts. Issues with cache displacement, TLB displacement and even modifications to branch-predictor state can arise, wherein code in the fail-retry path interferes with subsequent retries, sometimes repeatedly. Transaction failures caused by these issues can be very difficult to diagnose, especially because adding code to record and analyze failure reasons can change the behavior of the subsequent retries, resulting in a severe probe effect. As discussed further in (6), the logic for deciding whether to retry in hardware or fail to software was heavily influenced by these issues, and we hope to improve it further after understanding some remaining issues we have not had time to resolve yet.

6. Red-Black Tree

Next, we report on experiments similar to those in the previous seciton, but using a red-black tree, which is considerably more challenging than a simple hash table for several reasons. First, transactions are longer and access more data, and have more data dependencies. Second, when a red-black tree becomes unbalanced, new insertion operations perform "rotations" to rebalance it, and such rotations can occasionally propagate all the way to the root, resulting in longer transactions that perform more stores. Third, mispredicted branches are much more likely when traversing a tree.

We used an iterative version of the red-black tree (5), so as to avoid recursive function calls, which are likely to cause transactions to fail in Rock. We experimented with various key ranges, and various mixes of operations. In each experiment, we prepopulate the tree to contain about half the keys in the specified key range, and then measure the time required for all threads to perform 1,000,000 operations each on the tree, according to the specified operation distribution; we report results as throughput in total operations per microsecond. Figure 2(a) shows results for the "easy" case of a small tree (128 keys) and 100% lookup operations. Figure 2(b) shows a more challenging case with a larger tree (2048 keys), with 96% lookups, 2% inserts and 2% deletes.

The 100% lookup experiment on the small tree yields excellent results, similar to those shown in the previous section. For example, at 16 threads, PhTM outperforms the single lock by a factor of more than 50. However, as we go to larger trees and/or introduce even a small fraction of operations that modify the tree, our results are significantly less encouraging , as exemplified by the experiment shown in Figure 2(b). While PhTM continues to outperform the single lock in almost every case, in many cases it performs worse than the TL2 STM system (7). A key design principle for PhTM was to be able to compete with the best STM systems in cases in which we are not able to effectively exploit HTM transactions. Although we have not yet done it, it is trivial to make PhTM stop attempting to use hardware transactions, so in principle we should be able to get the benefit of the hardware transactions when there is a benefit, suffering only a negligible overhead when there is not. The challenge is in

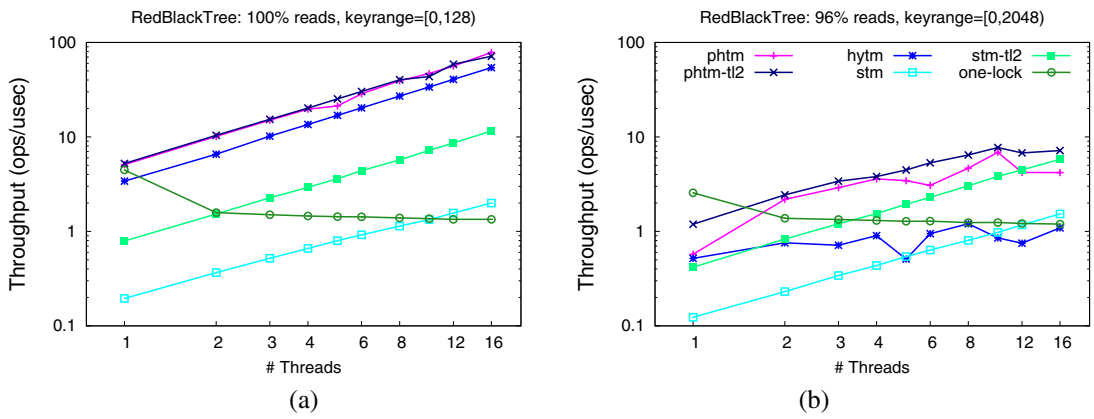

Figure 2. Red-Black Tree. (a) 128 keys, 100% reads (b) 2048 keys, 96% reads, 2% inserts, 2% deletes.

deciding when to stop attempting hardware transactions, but in extreme cases this is easy.

Before giving up on getting any benefit from HTM in such cases, however, we want to understand the behavior better, and explore whether better retry heuristics can help.

As discussed earlier, understanding the reasons for transaction failure can be somewhat challenging. Although the mentioned CPS improvements have alleviated this problem to some extent, it is still possible for different failure reasons to set the same CPS values. Therefore, we are motivated to think about different ways of analyzing and inferring reasons for failures. Below we discuss an initial approach we have taken to understanding our red-black tree data.

6.1 Analyzing Transacation Failures

Significant insight into the reason for a transacation failing can be gained if we know what addresses are read and written by it. We added a mechanism to the PhTM library that allows the user to register a call-back function to be called at the point that a software transaction attempts to commit; furthermore, we configured the library to switch to a software phase in which only the switching thread attempts a software transaction. This gives us the ability to examine the software transaction executed by a thread that has just failed to execute the same operation as a hardware transaction.

We used this mechanism to collect the following information about operations that failed to complete using hardware transactions: Operation name (Get, Insert or Delete); read set size (number of cache lines[1]); maximum number of cache lines mapping to a single L1 cache set; write set size (number of cache lines and number of words); number of words in the write set that map to each bank of the store queue; number of write upgrades (cache lines that were read and then written); and number of stack writes.

[1] In practice, we collected the number of ownership-records covering the read-set. Since each cache-line maps to exactly one ownership-record, and since the size of our ownership-table is very large, we believe that the two are essentially the same.

We profiled single threaded PhTM runs with various tree sizes and operation distributions. Furthermore, because the sequence of operations is deterministic (we fixed the seed for the pseudo random number generator used to choose operations), we could also profile *all* operations using an STM-only run, and use the results of the PhTM runs to eliminate the ones that failed in hardware. This way, we can compare characteristics of transactions that succeed in hardware to those that don't, and look for interesting differences that may give clues about reasons for transaction failures.

Results of Analysis In addition to the experiments described above, we also tried experiments with larger trees (by increasing the key range), and found that many operations fail to complete using hardware transactions, even for single threaded runs with 100% lookup operations. This does not seem too surprising: the transactions read more locations walking down a deeper tree, and thus have a higher chance of failing to fit in the L1 cache.

We used the above-described tools to explore in more depth, and we were surprised to find out that the problem was *not* overflowing of L1 cache sets, nor exceeding the store queue limitation. Even for a $24,000$ element tree, none of the failed operations had a read-set that overflowed any of the L1 cache sets (in fact, it was rare to see more than 2 loads hit the same 4-way cache set). Furthermore, *none* of the transactions exceeded the store queue limitation. Putting this information together with the CPS values of the failed transactions, we concluded that most failures were because too many instructions were deferred due to the high number of cache misses. Indeed, when we then increased the number of times we attempt a hardware transaction before switching to software, we found that we could significantly decrease the number of such failing transactions, because the additional retries served to bring needed data into the cache, thereby reducing the need to defer instructions.

Even though we were able to get the hardware transactions to commit by retrying more times, the additional re-

tries prevented us from achieving better performance than using a software transaction. This suggests we are unlikely to achieve significantly better performance using PhTM for such large red-black trees. Future enhanced HTM implementations may be more successful. It is also interesting to ponder whether different data structures might be more amenable to acceleration with hardware transactions.

Next we explored experiments with larger fractions of Insert and Delete operations. We profiled a run with 15% Insert/15% Remove/70% Get operations on a $1,024$ element tree. Most Insert and Remove operations eventually succeeded in a hardware transaction, and none of those that failed to software did so due to exceeding the store buffer size. Indeed, when checking the failure ratio as a function of the write-set size, we saw no strong correlation between the size of the operation's write-set and the failure ratio.

Putting this together with the CPS data, the most likely explanations for these failures are stores that encounter micro-DTLB misses or a store address that is dependent on an outstanding load miss. Both of these reasons result in a CPS value of ST, which is what we observed in most cases. In ongoing work, we plan to add more features to our profiling tools to help distinguish these cases.

7. TLE

In this section we report on our experience so far using TLE to improve the performance and scalability of lock-based code. The idea behind TLE is to use a hardware transaction to execute a lock's critical section, but without acquiring the lock, so that critical sections can execute in parallel if they do not have any data conflicts.

Rajwar and Goodman (19; 20) proposed an idea that is closely related to TLE, which they called Speculative Lock Elision (SLE). SLE has the advantage of being entirely transparent to software, and thus has the potential to improve the performance and scalability of unmodified legacy code. The downside is that the hardware must decide when to use the technique, introducing the risk that it will actually hurt performance in some cases. Performing lock elision explicitly in software is more flexible: we can use the technique selectively, and we can use different policies and heuristics for backoff and retry in different situations.

Although TLE does not share SLE's advantage of being transparent at the binary level, TLE can still be almost or completely transparent to the programmer. Below we discuss two ways in which we have tested TLE, one in which the programmer replaces lock acquire and release calls with macros, and one in which TLE is made entirely transparent to Java programmers by implementing it in the JVM.

7.1 TLE with C++ STL vector

We repeated the experiment described in (5), which uses simple macro wrappers to apply TLE to an unmodified STL `vector`. This experiment uses a very simplistic policy for deciding when to take the lock: it tries a transaction a fixed number of times before acquiring the lock, and does not use the CPS register to try to make better decisions.

To make a slightly more aggressive experiment, we changed the increment:decrement:read ratio to be 20:20:60, raher than the 10:10:80 used in (5). We also increased from 4 to 20 the number of retries before acquiring the lock because with higher numbers of threads the transactions would fail several times, and would therefore prematurely decide to take the lock. We have not yet conducted a detailed analysis of the reasons for requiring more retries, but we expect to find similar reasons as discussed in Section 5: cache misses lead to transaction failures for various reasons on Rock, but not on ATMTP; because the failed transaction issues a request for the missing cache line, it is likely to be in cache on a subsequent retry. Even using the simplistic policy described above, our results (Figure 3(a)) show excellent scalability using TLE, in contrast to negative scalability without.

7.2 TLE in Java

A particularly interesting opportunity is to use TLE to improve the scalability of existing code, for example by eliding locks introduced by the `synchronized` keyword in the Java programming language. This use of the TLE idea differs from the one described above in several ways.

First, we can be more ambitious in this context because the JIT compiler can use run-time information to heuristically choose to elide locks for critical sections that seem likely to benefit from doing so, and in cases in which lock elision turns out to be ineffective, we can dynamically revert to the original locking code. Furthermore, a TLE-aware JIT compiler could take into account knowledge of the HTM feature when deciding what code to emit, what optimizations to apply, and what code to inline. However, our prototype TLE-enabled JVM attempts TLE for *every* contended critical section, and the JIT compiler does not yet use knowledge of the HTM feature to guide its decisions.

In contrast to the earlier prototype described in (5), our TLE-aware JVM does make use of the CPS register to guide decisions about whether to retry, or backoff and then retry, or give up and acquire the original lock.

For our initial experiments with our modified JVM, we chose two simple collection classes, `Hashtable` and `HashMap`, from the `java.util` library. Both support key-value mappings. `Hashtable` is synchronized; `HashMap` is unsynchronized but can be made thread-safe by a wrapper that performs appropriate locking.

As in our previous work (5), we experimented with a simple read-only benchmark using `Hashtable` (slightly modified to factor out a divide instruction that caused transactions to fail) and `HashMap`. After initialization, all worker threads repeatedly look up objects that are known to be in the mapping. We have also conducted more ambitious tests that include operations that modify the collection. Results

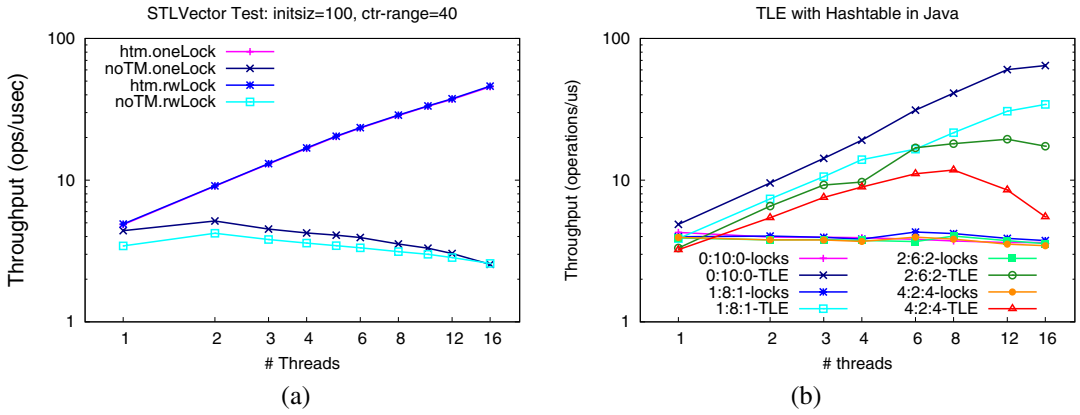

Figure 3. (a) TLE in C++ with STL `vector` (b) TLE in Java with `Hashtable`.

for `Hashtable` are shown in Figure 3; a curve labeled with 2-6-2 indicates 20% `puts`, 60% `gets`, and 20% `removes`.

With 100% `get` operations, TLE is highly successful, and the throughput achieved scales well with the number of threads. As we increase the proportion of operations that modify the `Hashtable`, more transactions fail, the lock is acquired more often, contention increases, and performance diminishes. Nonetheless, even when only 20% of the operations are `gets`, TLE outperforms the lock everywhere except the single threaded case. We hope to improve performance under contention, for example by adaptively throttling concurrency when contention arises.

We also conducted similar experiments for `HashMap`. As before (5), we found that `HashMap` performed similarly to `Hashtable` in the read-only test. When we introduced operations that modify the collection, however, while we still achieve some performance improvement over the lock, so far our results are not as good as for `Hashtable`. We have made some interesting observations in this regard.

We observed good performance with `HashMap` comparable to `Hashtable`, but noticed that later in the same experiment, performance degraded and became comparable to the original lock. After some investigation, we determined that the difference was caused by the JIT compiler changing its decision about how to inline code. At first, it would inline the synchronized collection wrapper together with each of the `HashMap`'s `put`, `get` and `remove` methods. Thus, when the JVM converted the synchronized methods to transactions, the code to be executed was all in the same method.

Later, however, the JIT compiler revisited this decision, and in the case of `put`, instead inlined the synchronized collection wrapper into the worker loop body and then emitted a call to a method that implements `HashMap.put()`. As a result, when the TLE-enabled JVM converts the synchronized method to a transaction, the transaction contains a function call, which—as discussed in Section 3—can often abort transactions in Rock. If the compiler were aware of TLE, it could avoid making such decisions that are detrimental to transaction success.

We also tested `TreeMap` from `java.util.concurrent`, another red-black tree implementation. Again, we achieved good results with small trees and read-only operations, but performance degraded with larger trees and/or more mutation. We have not investigated in detail.

We are of course also interested in exploiting Rock's HTM in more realistic applications than the microbenchmarks discussed so far. As a first step, we have experimented with the VolanoMark™ benchmark (18). With the code for TLE emitted, but with the feature disabled, we observed a 3% slowdown, presumably due to increased register and cache pressure because of the code bloat introduced. When we enabled TLE, it did not slow down the benchmark further, as we had expected, and in fact it regained most of the lost ground, suggesting that it was successful in at least some cases. However, a similar test with an internal benchmark yielded a 20% slowdown, more in line with our expectation that blindly attempting TLE for every contended critical section would severely impact performance in many cases.

This experience reinforces our belief that TLE must be applied *selectively* to be useful in general. We are working towards being able to do so. As part of this work we have built a JVM variant that includes additional synchronization observability and diagnostic infrastructure, with the purpose of exploring an application and characterizing its potential to profit from TLE and understanding which critical sections are amenable to TLE, and the predominant reasons in cases that are not. We hope to report in more detail on our experience with the tool soon.

8. Minimum Spanning Forest algorithm

Kang and Bader (10) present an algorithm that uses transactions to build a Minimum Spanning Forest (MSF) in parallel given an input graph. Their results using an STM for the transactions showed good scalability, but the overhead

of the STM was too much for the parallelization to be profitable. They concluded that HTM support would be needed to achieve any benefit. We report below on our preliminary work using Rock's HTM to accelerate their code.

We begin with a brief high-level description of the aspects of the MSF benchmark most relevant to our work; a more precise and complete description appears in (10). Each thread picks a starting vertex, and grows a minimum spanning tree (MST) from it using Prim's algorithm, maintaining a heap of all edges that connect nodes of its MST with other nodes. When the MSTs of two threads meet on a vertex, the MSTs and the associated heaps are merged; one of the threads continues with the merged MST, and the other starts again from a new vertex.

Kang and Bader made judicious choices regarding the use of transactions, using them where necessary to keep the algorithm simple, but avoiding gratuitous use of transactions where convenient. For example, transactions are used for the addition of new nodes to the MST, and for conflict resolution on such nodes. But new edges are added to the threads' heaps non-transactionally, and when two MSTs are merged, the associated heaps are merged non-transactionally.

Our work focuses on the main transaction in the algorithm, which is the largest one, and accounts for about half of the user-level transactions executed. It takes the following steps when executed by thread T.

- Extract the minimum weighted edge from T's heap, and examine the new vertex v connected by this edge.
- (Case 1) If v does not belong to any MST, add it to T's MST, and remove T's heap from the public space for the purpose of edge addition.
- (Case 2) If v already belongs to T's MST, do nothing.
- If v belongs to the MST of another thread T2:
  - (Case 3) If T2's heap is available in the public space, steal it by removing both T and T2's heaps from the public space for the purpose of merging.
  - (Case 4) Otherwise, move T's heap to the public queue of T2, so that T2 will later merge it once it is done with the local updates for its own heap.

After a short investigation using our SkySTM library, we noticed that the main transaction was unlikely to succeed using Rock's HTM, for two main reasons: first, the transaction becomes too big, mostly because of the heap extract-min operation. Second, the extract-min operation is similar to RBTree, traversing dynamic data that confounds branch prediction. However, we note that in two of the four cases mentioned above (Cases 1 and 3) the transaction ends up removing the heap from the public space, making it unavailable for any other threads. In these cases, it is trivial to avoid extracting the minimum inside the transaction, instead doing it right after the transaction commits and the heap is privately

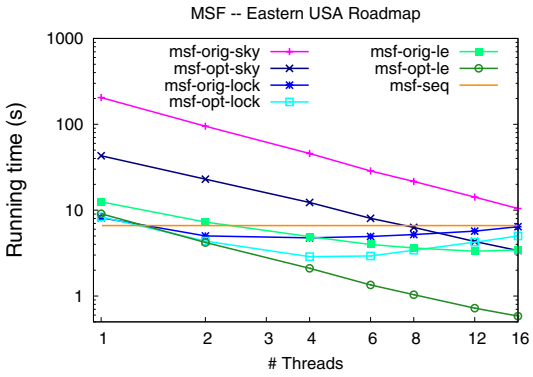

Figure 4. MSF

accessed. Fortunately, Case 1 is by far the most common scenario when executing on sparse graphs, as conflicts are rare.

We therefore created a variant in which we only *examine* the minimum edge in the heap inside the transaction, and then decide, based on the result of the conflict resolution, whether to extract it transactionally (in Cases 2 and 4) or non-transactionally (in Cases 1 and 3). This demonstrates one of the most valuable advantages of transactional programming. Extracting the minimum non-transactionally in *all* cases would significantly complicate the code. Using transactions for synchronization allows us to get the benefits of fine grained synchronization in the easy and common cases where transactions are likely to succeed, without needing to modify the complex and less frequent cases.

8.1 Evaluation results

So far, we have only experimented with protecting all transactions with a single lock, and then using Rock's HTM feature to elide this lock, as described previously. To this end, we evaluated 7 versions of the MSF benchmarks on Rock:

msf-seq a sequential version of the original variant, run single threaded, with the atomic blocks executed with no protection.

msf-{**orig,opt**}**-sky** : the original and new variants of the benchmark, respectively, with the atomic blocks executed as software transactions, using our SkySTM library.

msf-{**orig,opt**}**-le** the original and new variants of the benchmark with the atomic blocks executed using TLE. We try using a hardware transaction until we get 8 failures, where a transaction that fails with the UCTI bit set in the CPS register is only counted as half a failure; then fail to use a single lock.

msf-{**orig,opt**}**-lock** : the original and new variants of the benchmark, respectively, with the atomic blocks executed using a single lock.

To date we have only experimented with the "Eastern Roadmap" data set from the 9th DIMACS Implementation

Challenge (http://www.dis.uniroma1.it/~challenge9), which has 3,598,623 nodes and 8,778,114 edges. Figure 4 shows the results. Note that both axes are log scale, and that the msf-seq runtime is shown across the whole graph for comparison, although it was only run single threaded. Each data point in the graph is the average result of 6 runs, with a standard deviation of less than 3% for all data points.

The single-thread runs with msf-orig-sky and msf-opt-sky pay a 31x and a 6.5x slowdown, respectively, comparing to the sequential version, while the single-thread slowdown with the msf-opt-le version is only 1.37x. We also found that the fraction of user-level transactions that ended up acquiring the lock in single-threaded runs was over 33% with msf-orig-le, and only 0.04% with msf-opt-le. These observations demonstrate the value of our modifications to extract the minimum edge from the heap non-transactionally in some cases, as well as the significant performance improvement gained by executing transactions in hardware.

Both STM versions and msf-opt-le scale linearly up to 16 threads. With 16 threads, the msf-opt-le version outperforms the sequential version by a factor of more than 11, while msf-opt-sky only outperforms it by a factor of 1.95 and msf-orig-sky is unable to improve on the performance of msf-seq.

Finally, note that even though the optimized variant significantly reduces the average size of the main transaction, the transaction is still not small enough to scale well with a single-lock solution: even though msf-opt-lock scales further than msf-orig-lock, they both stop scaling beyond 4 threads. Thus, even though the software-only version that achieves the lowest running time is the msf-opt-lock with 4 threads, this running time is still almost 5 times longer than the best time achieved by the msf-opt-le version.

Finally, we also ran the experiments on Rock configured in SE mode. As we expected, in SE mode, the smaller store buffer caused many transactions to fail (CPS values for failed transactions were dominated by ST|SIZ), even in the optimized variant. Therefore, in the single thread execution of msf-opt-le in SE mode, the fraction of user-level transactions that resorted to acquiring the lock is more than 300 times higher than that of the same run in SST mode. As a result, in SE mode msf-opt-le provides "only" a 9x speedup with 16 threads, and stops scaling at all after this point. Still, even in SE mode, the optimized variant scales better than the original one, and with 16 threads is almost 3x faster than any of the software methods at any number of threads.

9. Discussion

Our investigation to date is very encouraging that we will be able to exploit Rock's HTM feature to improve performance and scalability of existing libraries and applications, in some cases without changes to the source code and in others with only small changes. Furthermore, at a time when transactional programming models are beginning to emerge, we have shown that we can exploit Rock's HTM to enhance performance of software transactional memory.

HTM-aware compilers may be able to make code more amenable to succeeding in hardware transactions. However, it is unlikely that there will be widespread support in commonly used compilers in the near future. Therefore, it is important that HTM is able to execute ordinary code, generated by any compiler. Rock's HTM interface was designed with this requirement in mind.

The difficulty of diagnosing reasons for transaction failures in some cases clearly points to the importance of richer feedback in future HTM features. Apart from further disambiguating different failure reasons, additional information such as program counter of failing instructions, addresses for data conflicts and TLB misses, time spent in failed transaction, etc. would be very useful. Furthermore, eliminating certain restrictions on hardware transactions will make the feature much easier to use in more contexts. The `save-restore` limitation in Rock is a key example.

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. It's not going to, certainly not in the near future. We hope our paper helps set expectations appropriately about what Rock's HTM feature can and cannot achieve. We also emphasize that this is the first step, and that software that uses this feature will automatically benefit from improvements in future HTM features.

There is plenty more work to do, both to maximize the benefit we can extract from Rock's new feature, and to guide the development of future HTM features.

10. Concluding remarks

We have described our experience evaluating the hardware transactional memory feature of Sun's forthcoming Rock multicore processor. This feature has withstood rigorous testing, which has revealed no correctness bugs. Furthermore, we have demonstrated successful use of this feature in a number of contexts. We conclude that Sun's architects have made a groundbreaking step towards sophisticated hardware support for scalable synchronization in multicore systems.

We have discussed techniques we used, challenges we faced, and some ways in which Rock could be improved to be even better. We hope our paper will be useful both to programmers who use Rock's HTM, and to architects designing related features in the future.

Acknowledgments: The work described in this paper would not have been possible without the hard work of many individuals who have been involved in the design and implementation of Rock's HTM feature, and also those involved in the Rock bringup effort. Among others, we are grateful to Richard Barnette, Paul Caprioli, Shailender Chaudhry, Bob Cypher, Tom Dwyer III, Quinn Jacobson, Martin Karlsson, Rohit Kumar, Anders Landin, Wayne Mesard, Priscilla Pon, Marc Tremblay, Babu Turumella, Eddine Walehiane, and

Sherman Yip. We are grateful to Peter Damron, who developed the HyTM/PhTM compiler. We thank David Bader and Seunghwa Kang for providing their MSF code and for useful discussions about it, and Steve Heller for bringing this work to our attention. We are also grateful to our coauthors on the two Transact papers (16; 5) on the ATMTP simulator and techniques developed to exploiting HTM in Rock.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb. 2005.

[2] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.

[3] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *IEEE Micro*, 2009. To appear.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. 12th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[5] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. Transact 2008 workshop. http://research.sun.com/scalable/pubs/TRANSACT2008-ATMTP-Apps.pdf.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*, 2006.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *Proc. 22th Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[10] S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2009. ACM. To appear.

[11] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory, November 2008. Under submission.

[12] Y. Lev and J.-W. Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 197–206, New York, NY, USA, 2008. ACM.

[13] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007. http://research.sun.com/scalable/pubs/TRANSACT2007-PhTM.pdf.

[14] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing implementations of transactional memory. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 134–143, New York, NY, USA, 2006. ACM.

[15] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. In *19th International Symposium on Distributed Computing*, 2005.

[16] M. Moir, K. Moore, and D. Nussbaum. The Adaptive Transactional Memory Test Platform: A tool for experimenting with transactional code for Rock. In *Workshop on Transactional Computiung (Transact)*, 2008. http://research.sun.com/scalable/pubs/TRANSACT2008-ATMTP.pdf.

[17] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 174–185, New York, NY, USA, 2007. ACM.

[18] J. Neffenger. The volano report, May 2003. http://www.volano.com/report/index.html.

[19] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. 34th International Symposium on Microarchitecture*, pages 294–305, Dec. 2001.

[20] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[21] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005.

[22] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[23] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.