

TECHNICAL REPORT

Early Experience with a Commercial Hardware Transactional Memory Implementation

**Dave Dice, Yossi Lev, Mark Moir,
Dan Nussbaum, and Marek Olszewski**



> *Sun Microsystems Laboratories*

Early Experience with a Commercial Hardware Transactional Memory Implementation

Dave Dice, Sun Microsystems Laboratories

Yossi Lev, Brown University and Sun Microsystems Laboratories

Mark Moir, Sun Microsystems Laboratories

Dan Nussbaum, Sun Microsystems Laboratories and

Marek Olszewski, Massachusetts Institute of Technology and Sun Microsystems Laboratories

SMLI TR-2009-180

October 2009

Abstract:

We report on our experience with the hardware transactional memory (HTM) feature of two revisions of a prototype 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. This technical report expands on our ASPLOS paper [9], providing more detail and reporting on additional work conducted since that paper was written.

- [9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *AS-PLoS'09: Proceeding of the 14th international conference on Architectural support for program-ming languages and operating systems*, pages 157–168, New York, NY, USA, 2009. ACM.



Sun Labs
16 Network Circle
Menlo Park, CA 94025

email address:
dave.dice@sun.com
levyossi@cs.brown.edu
mark.moir@sun.com
dan.nussbaum@sun.com
mareko@csail.mit.edu

©2009 Sun Microsystems, Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo, Java, JVM, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries. Information subject to change without notice.

The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

For information regarding the SML Technical Report Series, contact Mary Holzer or Nancy Snyder, Editors-in-Chief <Sun-Labs-techrep-request@sun.com>. All technical reports are available online on our website, <http://research.sun.com/techrep/>.

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
Sun Microsystems Laboratories
mark.moir@sun.com

Dan Nussbaum
Sun Microsystems Laboratories
dan.nussbaum@sun.com

Marek Olszewski
Massachusetts Institute of Technology and
Sun Microsystems Laboratories
mareko@csail.mit.edu

Abstract

We report on our experience with the hardware transactional memory (HTM) feature of two revisions of a prototype 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. This technical report expands on our ASPLOS paper [9], providing more detail and reporting on additional work conducted since that paper was written.

1 Introduction

The “multicore revolution” occurring 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 approaches are error prone and difficult to understand, maintain, and extend.

Transactional Memory (TM) [17] 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 performed

“atomically”, so that the programmer does not have to think about these accesses 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 and so on.

TM-related techniques have been proposed in many contexts, ranging from the original “bounded” HTM of Herlihy and Moss [17], to a number of “unbounded” HTM proposals [2, 35, 41], numerous software transactional memory (STM) approaches [10, 16, 25], and some approaches that combine hardware and software in various ways [19, 6, 37, 21, 3].

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 still entails significant overhead, and there is growing interest in hardware support to improve its performance.

If programmed directly, bounded HTM implementations such as those described in [17] impose unreasonable constraints on programmers, who must ensure that a transaction does not exceed architecture-specific limitations, such as the size of a transactional cache. However, it has been shown [6, 23, 30] 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.

Among other uses, such techniques can be used to support *transactional programming models*, in which programmers express sections of code to be executed atomically. The system is responsible for determining *how* the atomicity is achieved, and it may or may not use special hardware support for this purpose, including an HTM feature. It is important to understand that, in such systems, only the system implementors are aware of specific limitations of the HTM feature, or even its existence; application programmers are not concerned with these details.

Techniques that do not depend on any partic-

ular hardware transaction succeeding can be used with *best-effort* HTM. Best-effort HTM differs from bounded HTM in that it may commit transactions that are much larger than a bounded HTM feature would; furthermore, best-effort HTM is not required to guarantee to commit all transactions of any 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 transactions commit successfully, it can automatically adapt to take advantage of future improvements to the HTM feature.

Sun’s architects [4] 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 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 a 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 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 [24]. This extensive testing has not revealed any correctness problems.

Next, in the ASPLOS paper [9] that this report builds upon, we experimented with the HTM feature using the benchmarks from our work with the Adaptive Transactional Memory Test Platform (ATMTP) simulator [8], 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 input, the Rock architects

refined feedback about transaction failures in the second revision R2.

Since publishing [9], we have continued our work experimenting with Rock. This technical report provides an update on lessons we have learned since, particularly related to the use of “speculation barriers” to avoid aborting transactions. We have also added a new appendix describing in detail the feedback Rock gives when a transaction fails, and how to react to it.

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 10, 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 (in Section 4) and a double compare-and-swap (DCAS). We then use the DCAS to reimplement some components of the JavaTM concurrency libraries (see Section 5). Next, we experiment with transactional hash tables (in Section 6) and red-black trees (in Section 7) using a compiler and library supporting Hybrid TM (HyTM) [6] and Phased TM (PhTM) [23]; these use HTM to boost performance, but transparently revert to software if unsuccessful. In doing so, we found that we could significantly improve success rates for hardware transactions by inserting *speculation barriers* into the code. Our HyTM/PhTM compiler supports inserting such barriers into transactional programs. Section 8 describes a novel source-to-source compiler we developed to support inserting these speculation barriers in other contexts. Next we report on experiments with transactional lock elision (TLE) (see Section 9), which uses 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 Java Virtual Machine (JVMTM). In Section 10, we report on our success using Rock’s HTM feature to accelerate a parallel Minimum Spanning Forest algorithm due to Kang and Bader [18].

Discussion and conclusions are in Sections 11 and 12. Finally, Appendix A contains detailed information about the feedback Rock provides in response to aborted hardware transactions, along with lessons we have learned about how to improve transaction success rates.

2 Background

Rock [4, 5] is a multicore SPARC[®] processor that uses aggressive speculation to provide high single-thread performance in addition to high system throughput. For example, when a load miss occurs, 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). 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, then the hardware can revert back to the previous checkpoint and re-execute from there.

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. See Section 11 for further discussion.

When a transaction aborts, feedback about the cause of the abort is provided in the CPS (Checkpoint Status) register, which is read using the follow-

Mask	Name	Description and example cause
0x001	EXOG	Exogenous - Intervening code has run: <code>cps</code> 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	ASYN	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.

ing instruction: `rd %cps, %<dest reg>`. The bits of the CPS register are shown in Table 1, along with examples of reasons each bit might be set. We discuss the CPS register further in the next section, and a comprehensive description of the CPS register and how to use it appears in Appendix A.

Each Rock chip has 16 *microcores*, each capable of executing two threads, for a total of 32 threads in the default Scout mode [4, 5]. Rock can also be configured to dedicate the resources of both hardware threads in one microcore to a single software thread. This allows a more aggressive form of speculation, called *Simultaneous Speculative Threading* (SST), 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.)

As described in [4], Rock is organized into four *core clusters*, each of which contains four microcores, a single 64-entry, four-way set-associative instruction translation lookaside buffer (*micro-ITLB*), a single 32 kilobyte (kB), four-way set-associative level 1 in-

struction cache (*I1\$*) and two 32 kB four-way set-associative level 1 data caches (*D1\$s*). All four microcores in a core cluster share a single micro-ITLB and a single *I1\$*, and each of the two pairs of microcores in one core cluster shares a single *D1\$*.

In SST mode, each microcore also has a 32-entry, two-way skew-associative data translation lookaside buffer (*micro-DTLB*) and a 32-entry *store buffer*. In Scout mode, the micro-DTLB and store buffer for each microcore are both split in half, with one half being used for each hardware thread.

Rock also includes a single 2 MB eight-way set-associative level 2 (unified data and instruction) cache and a single 4096-entry, eight-way set-associative Unified TLB (*UTLB*), which can be configured to service micro-I/DTLB misses without software intervention.

All data presented in this paper is taken on a single-chip Rock system running in SST mode. We have briefly explored Scout mode and we discuss preliminary observations as appropriate.

3 Examining the CPS register

Our `cpstest` program 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, and so on. Furthermore, various tests are targeted to demonstrate the interactions with various hardware limitations, such as the size and geometry of caches, store buffers, and TLBs. We made extensive use of such tests to confirm and refine our understanding about the causes of transaction failures and the feedback received in each case. A detailed summary of this information is provided in Appendix A. Below we mention only a few observations of interest.

First, a note about the CPS value $0x01 = \text{EXOG}$. This bit is set whenever “exogenous” code has run in between transaction failure and when the CPS register is read, for example, if a clock interrupt occurs. All of the tests described below therefore occasionally see the EXOG bit set.

Second, it is important to understand that a failing transaction can set multiple bits in the CPS register, and that some bits can be set for any of several reasons. Table 1 lists one example reason that each bit may be set, and is not intended to be exhaustive. As part of our evaluation of R1, we worked with the Rock architects to compile 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.

save-restore In some circumstances (see Section 7.3), Rock fails transactions that execute a `save` instruction and subsequently execute a `restore` instruction, setting CPS to $0x8 = \text{INST}$. This pattern is commonly associated with function calls; we discuss this issue further in Sections 7 and 9.

tlb misses To test the effect on transactions of TLB misses, we re-`mmap` the memory to be accessed by a transaction before executing it. In current implementations of the Solaris™ operating system (OS), this has the effect of removing

all micro-DTLB and UTLB mappings for that memory. When we load from an address that has no UTLB mapping, the transaction fails with CPS set to $0x90 = \text{LD|PREC}$. When we store to such an address, it fails with CPS set to $0x100 = \text{ST}$. This is discussed further below.

To test the effects on transactions of UTLB misses for instruction references, we copied a small code fragment to `mmap`d memory and then attempted to execute it within a transaction. When there was no UTLB mapping present for an instruction so copied, the transaction failed, setting CPS to $0x10 = \text{PREC}$.

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 $0x80 = \text{LD}$ and $0x40 = \text{SIZ}$. 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 $0x1 = \text{EXOG}$, as discussed above.

cache set test This test performs loads from five different addresses that map to the same four-way L1 cache set. As usual, we see occasional instances of EXOG, but almost all transactions in this test fail with CPS set to $0x80 = \text{LD}$, as expected.

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 execut-

ing the transaction can 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. These spinner threads execute a tight loop that does not access memory, and are thus more cache-friendly than the system’s default idle loop.

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 (SST), all such transactions fail with CPS set to $0x100 = ST$ if there are no UTLB mappings (see above) and with CPS set to $0x140 = ST|SIZ$ if we “warm” the UTLB first. A good way to warm the UTLB is to perform a “dummy” compare-and-swap (CAS) to a memory location 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 UTLB 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 the above-mentioned changes to disambiguate some failure cases, it can be challenging 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-DTLB miss — see [4] for more details of Rock’s Memory Management Unit (MMU).

In the first case, retrying will normally (eventually) succeed because the cache miss will eventually be resolved without software intervention. In the latter case, an MMU request is generated by the failing transaction, so the transaction may succeed if retried when the needed micro-TLB mapping is already present in the UTLB. However, if no mapping for the data in question exists in the UTLB, the transaction will fail repeatedly unless software can successfully warm the UTLB, as described above.

Thus, the best strategy for a transaction that fails with CPS value ST is to retry a few times to allow enough time for the micro-DTLB to be loaded from the UTLB if an entry is present. If the transaction does not succeed, retry again only after performing UTLB warmup if feasible in the current context, and give up otherwise.

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 a time or two because the load on which the mispredicted branch depends 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. When the UCTI bit is set, retrying immediately in the hope that the cache miss is resolved (so we won't misspeculate in the same way again) is probably the best choice. See Section A.1 for additional details.

We discuss the 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 counter

In our next experiment, we used Rock's HTM to implement a simple counter, comparing its performance against simple implementations using locks and CAS. For the lock-based implementation we used a pthreads mutex. In this experiment, each of n threads performs 1,000,000 increments on the counter, and we measure the time taken for all threads to complete all their operations, and report throughput in total increments per microsecond. Results are shown in Figure 1; this and all other graphs in this paper are presented on log-log scales.

As the "coherence" component of the `cpstest` demonstrated, transactional success rates are very low under heavy contention if no backoff is performed between conflicting transactional attempts. Therefore, results are not shown for the counter implementation that uses HTM with no backoff. The backoff used with the CAS-based and HTM-based implementations is a simple capped exponential backoff, in which backing off is achieved by executing a `membar #Sync` instruction some number of times. The CAS-based implementation backs off on any failure, while the HTM-based one backs off any time it sees the COH bit set in the CPS register.

In the single-thread case, the simple lock outperforms the other methods by a small margin, due to a pthreads optimization used when only one thread exists. With two or more threads, all of the methods that perform synchronization directly on the data

(CAS with and without backoff, and HTM with backoff) outperform the simple lock by a factor of roughly four. We have not yet put any effort into tuning the backoff parameters or mechanisms for the CAS- and HTM-based counters, but with our simple first cut, HTM with backoff and both CAS methods exhibit similar performance.

The low success rates without backoff are due to 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 resolution policies that avoid immediately failing transactions as soon as a conflict arises might result in better and more stable performance under contention.

5 DCAS

The DCAS (double compare-and-swap) operation generalizes the well-known CAS operation to two locations: it can atomically compare two independent memory locations to specified expected values, and if both match, store specified new values to the locations. There has been considerable research into the power of DCAS to support concurrent data structures, especially nonblocking ones [15, 14, 7, 11]. It is straightforward to implement DCAS using simple hardware transactions. However, there are some issues to consider, and as we discuss below, designers of future HTM features are encouraged to think about these issues early in their planning.

5.1 DCAS guarantees

We would like to be able to implement DCAS over HTM in a manner that does not require any alternative software implementation. This has the significant advantage that the implemented DCAS operations will be atomic with respect to ordinary loads and stores. Because the code to implement DCAS over HTM is small, simple, and known, we can exploit our knowledge of the transaction and the HTM

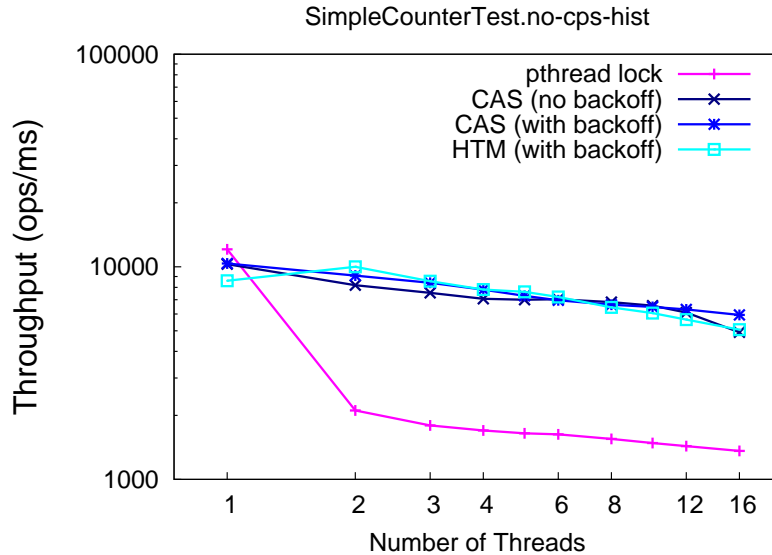


Figure 1: Simple counter benchmark.

feature when implementing DCAS. For example, because the addresses of the memory operands are known, we can perform warmup code to establish UTLB mappings before attempting a transaction (see Section A.4).

There are several challenges associated with implementing a DCAS operation that does not require a software alternative. The first is defining precisely what this means. To require every transaction that implements a DCAS operation to commit successfully seems at odds with a best-effort HTM implementation (and Rock does not do so). However, if transactions used to implement a DCAS operation can fail repeatedly and deterministically, the implementation is not very useful without a software alternative. As far as we know, there is no reason this would happen on Rock, but we have not been successful in getting this to be officially stated, in part because of the difficulty of stating a precise guarantee.

Because of Rock’s simple “requester wins” policy for handling conflicts, two concurrent transactions can cause each other to abort, and in principle this can happen repeatedly. In practice, random-

ized backoff is effective in avoiding the problem, but again, it is difficult to state a precise guarantee. We can also consider making progress guarantees for a DCAS operation only in the absence of contention. An example of such a guarantee would be that, if a transaction is repeatedly attempted in order to effect a given DCAS operation that does not encounter any conflicts with other transactions, then the transaction will eventually succeed. However, because conflicts are detected at cache line granularity, the definition of “conflict” for this purpose would need to include false sharing (i.e., a transaction that implements a DCAS operation failing due to a conflict with another memory access to the same cache line as one of the DCAS operands).

Furthermore, it is possible for a transaction on Rock to fail due to activity of another thread running on a neighboring core, even if that thread does not access any memory in the same cache lines accessed by the DCAS operation. For example, multiple cores share an L1 cache. Therefore, it is possible for one core to repeatedly evict a cache line read by a transaction running on a neighboring core, thus causing it

to fail repeatedly. Again, with randomized backoff, it is very unlikely that this would happen repeatedly, but pathological scenarios cannot be ruled out. It is possible to temporarily disable neighboring cores to avoid such scenarios, but stronger guarantees that would avoid the need for such drastic measures would clearly be desirable. We therefore advise designers of future HTM features to consider whether they can make stronger guarantees for small transactions an early design goal.

At this time, we do not have an officially approved DCAS implementation with a precise progress guarantee. However, the techniques we used have been effective in implementing a DCAS operation that in practice always eventually completes, so we have been able to experiment with using DCAS and other small, static transactions. We describe some of this work below; another use of small and simple transactions is described in [40].

To avoid the need for a software alternative, we must be mindful of several potential pitfalls. First, if a DCAS transaction encounters a UTLB miss, some warmup must be done in order to prevent repeated failures. Because we know the addresses of the operands, avoiding repeated UTLB misses for a data access is straightforward: we perform a dummy CAS on each operand, as described in Section 3. In some cases, we found that it was important not to do so too aggressively, because the CAS invalidates the cache line, and could thus cause subsequent retries to fail, for example due to misspeculation. Section 7.2 discusses some measures for avoiding failure due to misspeculation, but this entails some overhead, and more judicious warmup can largely avoid the problem in the first place.

To avoid UTLB misses from an instruction access during transactional execution, we use an inline assembly `align` directive to ensure that the entire transaction resides on the same page, as described in Section A.3.1.

Other sources of interference from other cores are possible. For example, activity on another core may interfere with shared branch predictor state, resulting in mispredicted branches, which can cause transactions to fail. We have experimented with using bitwise logical operators and conditional moves to avoid

branches and thus avoid mispredicted branches in transactions, and with avoiding storing to an operand when its expected value and new value are equal, to avoid unnecessary conflicts and coherence traffic. We have also explored a number of variants on what is done outside the transaction, particularly in response to transaction failure. For example, we have tested the effects of prefetching the operands before beginning the transaction (or before retrying it), and with various policies for deciding whether to backoff before retrying the transaction.

As expected, we have found that insufficient backoff under contention can result in poor performance, but simple backoff in reaction to certain values of the CPS register, especially the COH bit, is effective. We have also found that it can be beneficial to read the operands outside the transaction — if either one differs from its expected value, the DCAS operation can fail immediately. However, we have found that we should not do this too frequently (for example after every attempt), because these additional loads can cause other DCAS transactions to fail.

Stating and proving a precise and strong guarantee is challenging. In particular, because the L2 cache is shared, an adversary could conceivably interfere with attempts to execute a DCAS transaction forever. Although the techniques described above allow us to implement a DCAS operation that is highly unlikely to fail repeatedly in practice, an ironclad guarantee might require having the software park all other cores in order to absolutely guarantee progress, which would require operating system support.

We have not yet settled on the best recipe for implementing DCAS over Rock transactions, and as suggested above, this is to some extent dependent on requirements of an application. We believe that the techniques described above will suffice in practice, even if no absolute guarantee is made.

5.2 Implementing DCAS

We have experimented with several different ways to implement DCAS in several different contexts. Our description in this section focuses on our experiments with DCAS in Java programs, but most of the points of interest carry over to other contexts.

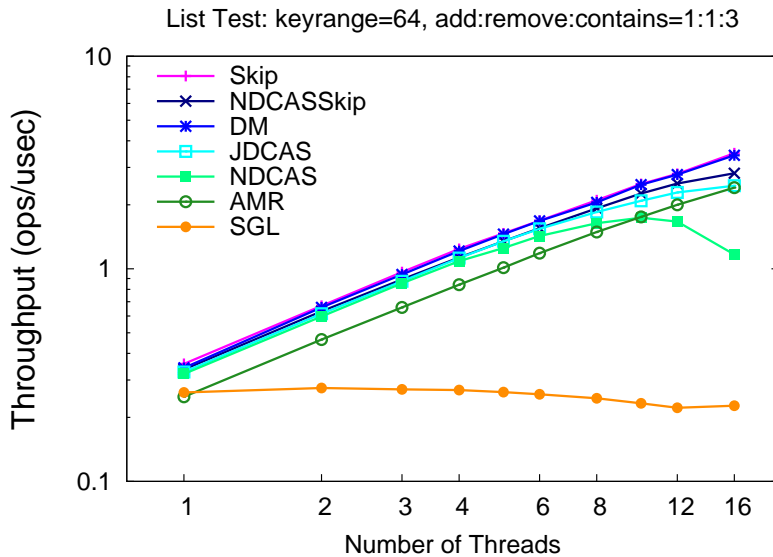


Figure 2: DCAS-based lists.

We used two implementations of DCAS, one (NDCAS) implemented in native assembly code, and the other (JDCAS) implemented in Java code using special `unsafe` methods we incorporated into an experimental Java virtual machine to allow Java programs to use the `chkpt` and `commit` instructions. The latter version posed some challenges due to interactions with the just-in-time (JIT) compiler, and with behind-the-scenes code, such as barriers for garbage collection. Dice, et. al. [8] report in more detail similar experiences they had with their work using their simulator.

5.3 Using DCAS

Although we believe we can still improve on our DCAS implementations, our explorations to date have already allowed us to achieve some interesting results. We used DCAS to construct an ordered, list-based set object that implements the `java.util.Set` interface, which provides `add` and `remove` and `contains` methods on the set, as well as an `iterator` method.

Figure 2 shows the results of an experiment in which each thread repeatedly executes three `contains` methods and one each of `add` and `remove` methods, all with randomly generated integers from 1 to 64. We measure the total number of operations completed in a fixed time period, and report throughput in total operations completed per microsecond.

We first note that the single global lock (SGL) implementation provides *reduced* throughput as the number of threads increases, as expected. Deletion Marker (DM) is the state-of-the-art, CAS-based implementation currently used in the `java.util.concurrent` library. This carefully handcrafted algorithm improves on the previous best Atomic Marked Reference (AMR) algorithm, which is based on Michael’s algorithm [27]. Both the NDCAS- and JDCAS-based implementations perform comparably with DM (except for a degradation for NDCAS at 16 threads, which we have not yet investigated).

Similarly, we implemented a DCAS-based skiplist and compared it to the one currently used in the `java.util.concurrent` library, and again our new implementation performed comparably with the

state-of-the-art one. Results are shown in Figure 2.

We believe it is interesting that we were able to match the performance of the state-of-the-art hand-crafted algorithms, even though we have not yet tuned our DCAS implementation. We believe that this approach has strong potential to simplify and improve Java libraries and other libraries.

6 Hash table

The simple hash table 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 hash table is implemented in C++, using a compiler and library that can support HyTM [6] and PhTM [23], as well as several STMs. Because the hash table operations are short and simple, we should be able to get them to succeed as hardware transactions most of the time. 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 3 shows our results from experiments with lookup:insert:delete ratios of 100:0:0, 60:20:20, and 0:50:50, for key ranges of 256 and 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

*at all*¹; thus approximately 50% of mutator operations modify memory in this experiment. In this and other similar graphs, `hytm` and `phtm` refer to HyTM and PhTM using our SkySTM algorithm [20] for the STM component, and `stm` is SkySTM with no hardware support. We present results as throughput in total operations per microsecond.

For all six scenarios, we observe high hardware transaction success rates for both HyTM and 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 3 shows that these methods clearly outperform all software-only methods.

With a key range of 256 and an operations mix of lookups:inserts:deletes=0:50:50, 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 [10] 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.

The 128,000 key range experiment yields qualitatively similar results to the smaller key range (again with an operations mix of lookups:inserts:deletes=0:50:50), 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 lower — 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.

The results look even better for HTM when we include more lookups in the operations mix. For example, 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.

¹Note difference from “standard” hash table semantics for the insert operation.

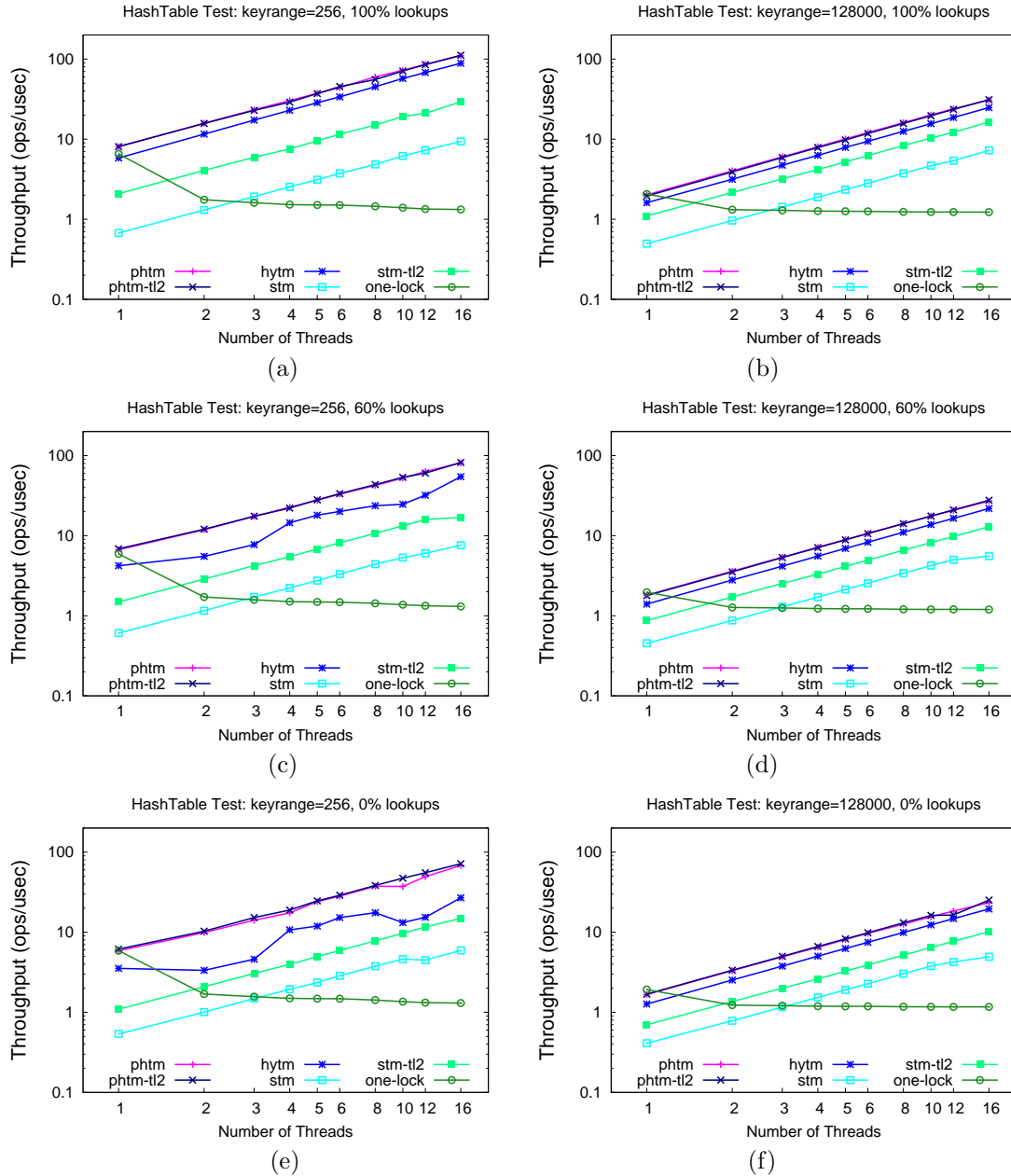


Figure 3: HashTable — Key range: left=256, right=128000. Lookup:Insert:Delete ratios: top 100:0:0, middle 60:20:20, bottom 0:50:50.

Examining some of the statistics collected by the PhTM library yielded interesting observations that give some insight into the reasons for transactions failing on Rock. For example (returning to the lookups:inserts:deletes=0:50:50 operations mix), with the 128,000 key range experiment, more than half of the hardware transactions are retries, even for the single-thread case (these retries explain why the single lock outperforms HyTM and PhTM somewhat in the single-thread case). In contrast, with the 256 key range experiment, 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 7) highlighted the possibility of the code in the fail-retry path interfering with subsequent retry attempts. Issues with cache displacement, micro I- and DTLB 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 Section A.2, 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.

6.1 Post-ASPLOS update

Throughout this report, we compare transactional versions of our micro-benchmarks to versions that

use a single global lock because that is what can be achieved with similar programming effort. Recent user studies [32, 36], while preliminary in nature, lend support to our belief that, in general, programming with atomic blocks is similar to programming with a single lock, while finer-grained use of locks is considerably more difficult.

However, it is not difficult to implement a fine-grained lock-based version of a hash table like the one we have used in this section. Therefore, we performed additional experiments to compare the relative performance of a simple, fine-grained locking hash table implementation to our transactional hash table.

In the experiments described below, for a conservative comparison, we used a simple CAS-based lock with no backoff in preference to the more expensive pthreads lock that we used for the rest of the performance results presented in this paper. (The low level of contention engendered by this test obviated the need to add backoff.)

Results are shown in Figure 4. Note that these results are not directly comparable with those in Figure 3 for a number of reasons, including the following:

- The particulars of the hardware on which the tests were run, including clock speed, were different.
- Because various components of our system have been evolving over time, these experiments use different versions of the compiler and library.
- The numbers in Figure 4 were taken with statistics disabled — all other results in this paper were taken with statistics enable.

This last point is worth elaborating on. Normally, we run our transactional experiments with statistics turned on, usually at a cost of something like 10-20% in performance, in order to gain deeper insight into the behavior of the benchmarks. In Figure 4, since statistics were not available in the fine-grained lock-based version, we ran the other versions without statistics as well, in order to make the comparison fair.

The results in Figure 4 are mixed. At one extreme, in the 100% read, memory bound (keyrange=128K)

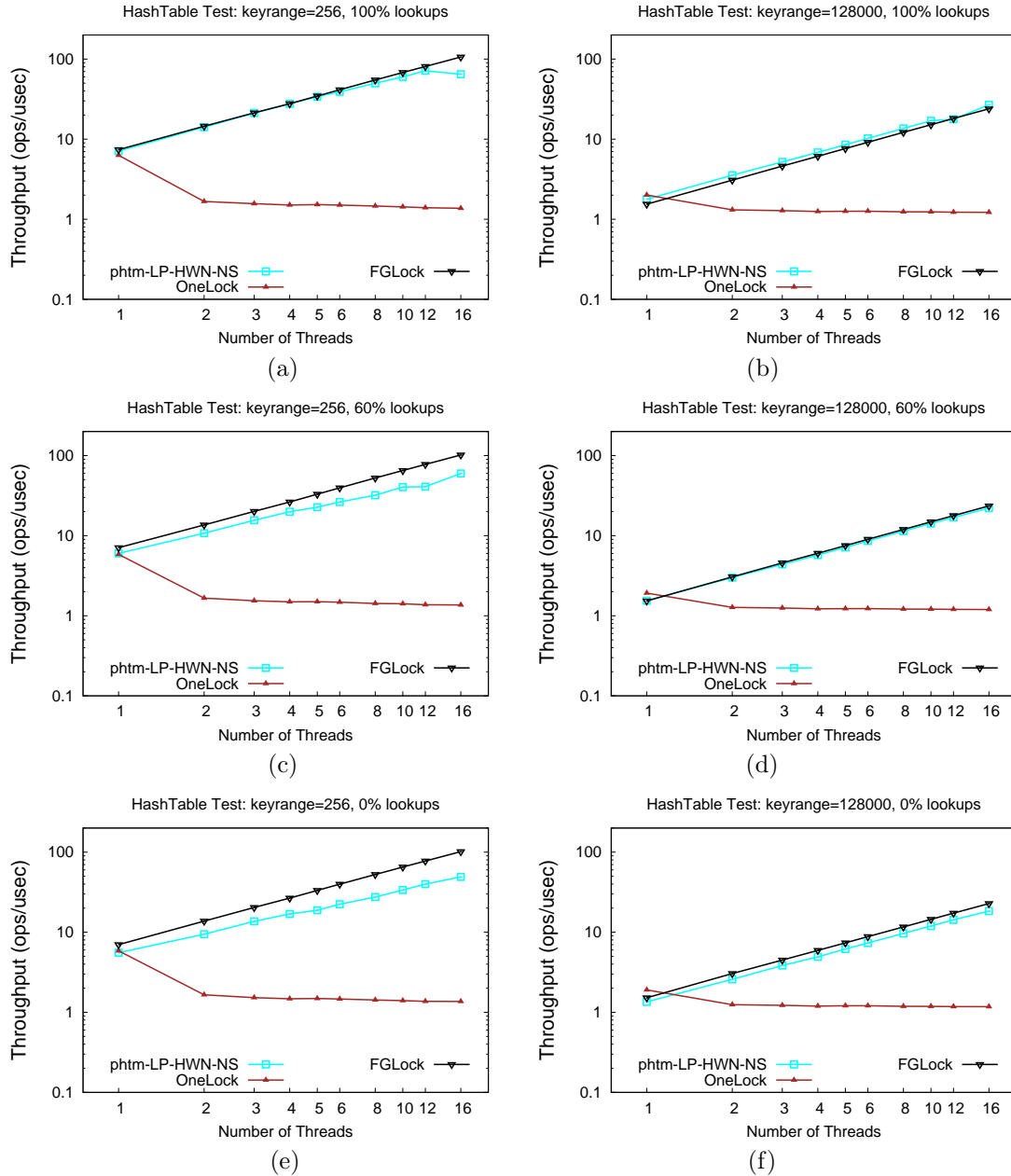


Figure 4: HashTable — comparison with fine-grained lock-based version. Key range: left=256, right=128000. Lookup:Insert:Delete ratios: top 100:0:0, middle 60:20:20, bottom 0:50:50.

case (graph (b) in the figure), the transactional version outperformed the fine-grained lock-based version by about 10%. At the other extreme, in the 0% read, cpu-bound (keyrange=256) case (graph (e) in the figure), the fine-grained lock-based version outperformed the transactional version by between 30% (single-threaded) and a factor of two (16 threads).

Where the transactional version outperformed the fine-grained lock-based version, we believe that the difference in performance stemmed from the difference in type of protocol traffic generated by transactions and by CAS instructions. Specifically, successful hardware transactions read, but do not modify, the metadata (the PhTM mode variable), while lock-based critical sections must always modify the metadata (the lock). Therefore, in the transactional case, metadata can be read shared between transactions, while in the lock-based case each operation must invalidate other cached copies of the metadata and acquire an exclusive copy of the metadata.

Where the fine-grained lock-based version outperformed the transactional version, we think that two factors contributed to the difference:

1. Infrastructure overhead for PhTM.

The code produced by our compiler for a PhTM transaction includes two loops — the first for hardware execution and the second for software execution. There is some overhead associated with these loops. Furthermore, the version of the library for which the numbers in Figure 4 were taken was “one-size-fits-all” — it contains runtime conditionalization to be able to function correctly for STM-only execution, PhTM execution and HTM-only execution. The cost of that conditionalization is considerable — we hope to remove it soon. Finally, there is significant scope for optimizing the PhTM code schema and the code it encapsulates; this has not yet been explored.

2. Costs associated with software execution and the transitions between hardware and software mode.

When a hardware transaction fails (several times), PhTM switches to software mode to com-

plete the transaction. Each such failure represents several (ultimately useless) failed attempts to execute in hardware. In addition to the overhead of switching to software mode, completing a transaction in software is substantially more expensive than if the transaction had succeeded in hardware.

In the single-threaded case, the fraction of transactions that cause a switch to software mode is very small (0.0002%), so these costs don’t impact performance significantly. However, PhTM pays an increasing cost for switching to software mode as the number of threads increases. For a 16-thread run, 0.06% of the transactions decide to switch to software mode, and a total of 1% of the transactions run in software as a result (because a transaction that starts while the system is in software mode will execute in software, even though that transaction did not itself decide to switch to software mode). This is enough to account for the noticeable difference in performance between the fine-grained lock-based version and the transactional version.

These results show that a simple transactional hash table can achieve performance comparable to that of a simple fine-grained lock-based hash table in most cases, occasionally outperforming it slightly, and in some cases performing worse than it by a noticeable margin, while still substantially outperforming the simpler coarse-grained lock implementation.

The additional programming complexity for the simple fine-grained hash table is not prohibitive, but this is the case for many other data structures. For example, fine-grained lock-based implementations of red-black trees have not been known until relatively recently, and are quite complex [13]. In contrast, simple transactional implementations can substantially outperform simple lock-based ones (Section 7).

Even restricting our attention to hash tables, we note that extending such comparisons to hash tables with more features, such as the ability to re-size, brings different additional challenges for both fine-grained locking and for transactional approaches. Further exploration in this direction is a topic of future research.

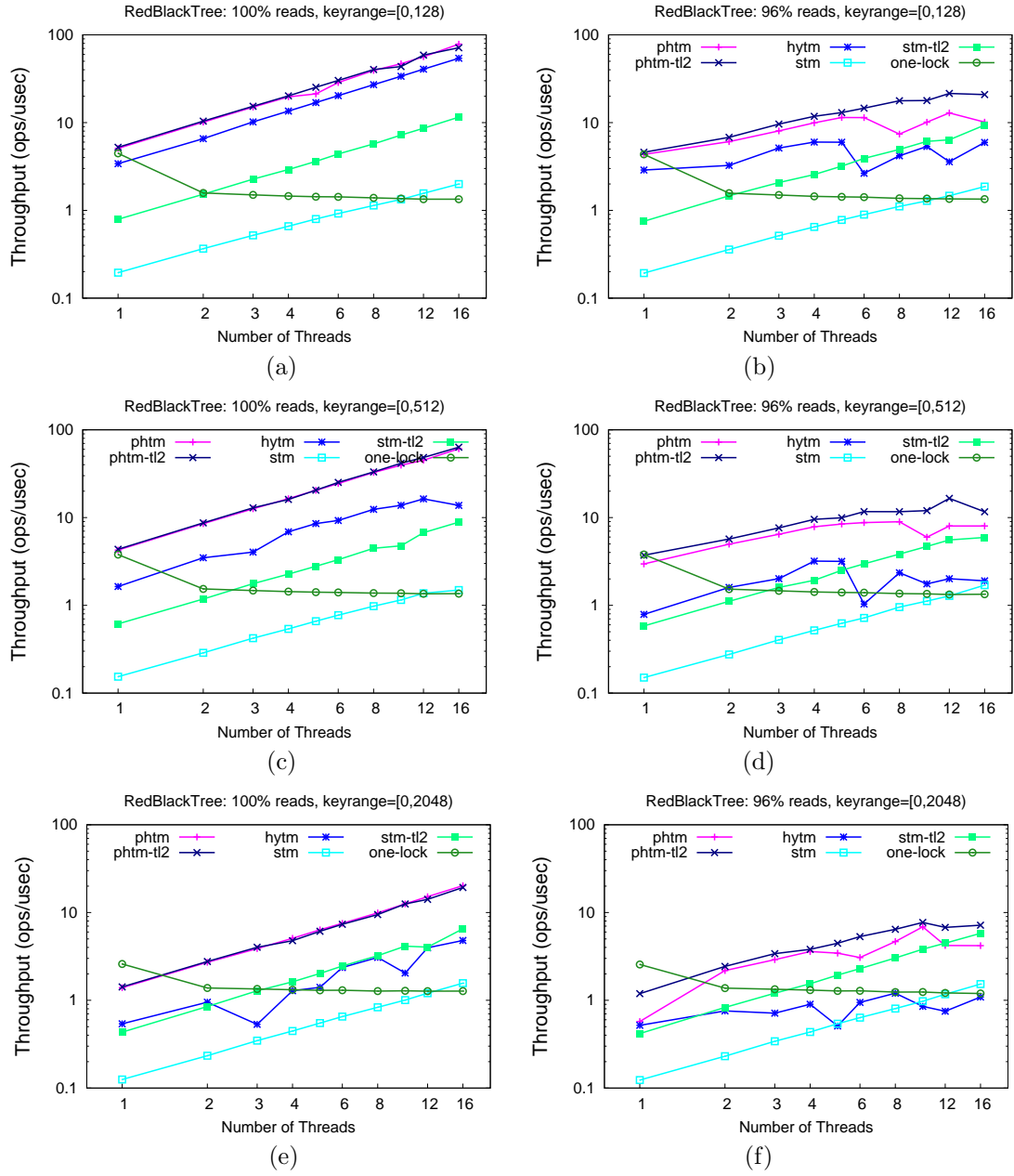


Figure 5: Red-Black Tree — Top: 128 keys, middle: 512 keys, bottom: 2048 keys. Left: Lookup:Insert:Delete=100:0:0; Right: Lookup:Insert:Delete=96:2:2.

7 Red-black tree

Next, we report on experiments similar to those in the previous section, but using a red-black tree, which is considerably more challenging than a simple hash table for several reasons. First, transactions are longer, access more data, and have more data dependencies. Second, when a red-black tree becomes unbalanced, new Insert 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 [8], in order 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 5 shows results for key ranges of 128, 512, and 2048, and for Lookup:Insert:Delete=100:0:0 and Lookup:Insert:Delete=96:2:2 operations mixes.

The 100% Lookup experiment on the smallest tree (key range = 128) 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 5(f).

While PhTM continues to outperform the single lock in almost every case, in a few cases it performs worse than the TL2 STM system [10]. 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 would be straightforward to make PhTM stop attempting to use hardware transactions, so in principle we should be able to

get the benefit of hardware transactions when there is a benefit to be had, suffering only a negligible overhead when there is not. The challenge is in deciding when to stop attempting hardware transactions, but in extreme cases this is easy.

Before giving up on getting more benefit from HTM in such cases, however, we want to better understand the behavior, 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.

7.1 Analyzing transaction failures

Significant insight into the reason for a transaction failing can be gained if we know what addresses it reads and writes. We added a mechanism to the PhTM library that allows the user to register a callback function to be called at the point that a software transaction attempts to commit; furthermore, we modified 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 using a hardware transaction.

We used this mechanism to collect the following information about operations that failed to complete using hardware transactions: Operation name (Lookup, Insert, or Delete); read set size (number of cache lines²); maximum number of cache lines map-

²In practice, we collected the number of orecs covering the read-set. An *orec* (short for “ownership record”) is a chunk of metadata that is used to mediate accesses to the memory locations that map to it. Each cache-line-sized chunk of memory maps to exactly one orec; see [20] for more details. Although multiple such chunks of memory map to each orec, we use a large number of orecs, and therefore we believe that the number of orecs that cover a transaction’s read set closely approximates the size of the transaction’s read set.

ping 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.

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 two 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 attempted a hardware transaction before switching to software, we found that we could significantly decrease the number of such failing transactions, be-

cause 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 retries 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 a Lookup:Insert:Delete=70:15:15 operations mix, on a 1,024 element tree. Most Insert and Delete 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 (see Appendix A), 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.

7.2 Post-ASPLOS update

When the final version of our ASPLOS paper [9] was due, we still had disappointing results in some cases that we did not fully understand, even with only a single thread executing, as described above. Since then, we have refined our understanding significantly, and we've also identified some techniques for improving the success rate. In this section, we describe our experience and the lessons we learned.

When we first got access to an R2 machine, we began by rerunning the previous experiments. When we reran the hashable experiments, we again saw good

performance for 100% Lookups, but the experiment with 50% Inserts and 50% Deletes was no longer giving good results, both for PhTM and HyTM. The red-black tree experiments were worse than before, with almost all hardware transactions failing. We had done a fair amount of work on the runtime library, including updating the code for interpreting and reacting to the CPS register, in anticipation of the CPS enhancements that were implemented in R2. So we were not too surprised to see worse results than we had seen previously, and we began again to investigate the reasons, this time armed with much more complete and detailed knowledge of the CPS register (see Appendix A).

As before, we used the mechanism described in Section 7.1 to profile failing transactions, and as before we did not find any evidence of the transactions exceeding capacity or geometry constraints of the L1 cache or micro-DTLB, nor did we see evidence of the store buffer capacity being exceeded.

So it seemed that we were no closer to understanding the problems than we were before, and in fact the situation was now worse, because we were no longer able to reproduce the good results with the 0% Lookup hash table experiments. This latter fact made it clear that we had some specific problem, and we could no longer dismiss the poor red-black tree results by vaguely hypothesizing that we were approaching the limitations of Rock’s HTM with the larger and more complex data structure.

Continued investigation, with further help from Rock architects, yielded some key insights. First, when we started experimenting with one of the data points that provided worse performance than previously, it miraculously started performing well, just as before. We soon noticed that if we ran this experiment at the command line, it performed well, while minor changes such as running it from a script, or even just redirecting the output to a file, would cause the poor performance to recur. This seemed very odd, until we realized that the slight difference in the environment in which the test was run resulted in some variables being allocated on different pages, and thus mapping differently into the micro-DTLB. As a result of this experience, the presentation for [29] at ASPLOS ’09 described seemed very familiar!

We confirmed that this difference was key to our investigation by allocating some unused space in order to cause a similar shift in addresses, and in this case the performance was poor regardless of whether running at the command line, with or without redirection, or running in a script. We also tried allocating much smaller unused space, so that the relevant variables were shifted enough to map to a different cache set, but not to fall on a different page. With this change, we could again control whether the problem manifested by running with or without output redirection. Together, these observations strongly suggested that our problem was related to micro-DTLB issues. However, when we double checked our analysis of the transactions, we were still unable to explain how the transactions would repeatedly fail to fit in the micro-DTLB.

We soon realized that we had been overlooking an important point that is obvious in hindsight: code executed in the “fail path” (that is, after the failure of one transaction and before the start of the subsequent retry) can interfere with state established by previous attempts, such as cache and micro-DTLB state.

So, we again attempted to capture information about *all* memory accesses, this time taking into account the fail path as well as the transaction itself. This was painstaking work, and several times we identified memory accesses we had previously overlooked. After this settled down, we *still* could not definitively prove that any of the failing transactions we studied could not fit into the micro-DTLB or L1 cache, even taking these additional accesses into account.

However, we did observe that the micro-DTLB was under some pressure, in that there were transactions that might fail the first time due to micro-DTLB conflicts, but should succeed subsequently. In some cases with the red-black tree experiments, just one additional memory access with an unlucky mapping to the micro-DTLB would be sufficient to ensure that the transaction and the fail path could not coexist in the micro-DTLB. But our continued efforts to identify additional memory accesses we had overlooked did not turn up any new ones. Furthermore, aggressive efforts to remove all memory accesses and almost all branches from the fail path failed to eliminate the

problem. This was frustrating!

At some point, the new UCTI bit provided an important clue. We had transactions that were failing repeatedly with the UCTI bit being set, even if retried thousands of times. Previously, we had believed that the UCTI bit should always be transient, because it occurs only when a cache miss occurs during the transaction. We reasoned that unless the transaction and fail path cannot fit together in the L1 cache, this should never happen: we should eventually have no more cache misses and therefore we should no longer see the UCTI bit set.

During yet another “how is this possible?” conversation with the Rock architects, we realized that this thinking was too simplistic. In particular, as described previously, when the UCTI bit is set, this indicates potential misspeculation. Misspeculation can cause memory accesses that are not accounted for by our analysis! This observation helps to explain some of the behaviors we observed.

Below we describe one of the more interesting scenarios that we believe we encountered. But first we need to describe one more interesting observation we made during this work: interference with branch prediction state can also contribute to repeated failure scenarios. In particular, we found we could control the performance of some tests by placing a few no-ops in the code.

We first did this when experimenting with (wrong) hypotheses about what happens if we encounter the commit instruction while certain speculative events are still in progress. Putting a few no-ops before the commit made the transactions succeed. Our friends in the Rock group did not believe these bizarre theories, and suggested that we try putting the no-ops *after* the commit instead. We did so, and it still made the transactions succeed. Because the success of the transactions was being affected by code executed outside of the transaction that did no memory references, the only relevant state we could think of that would be affected by such changes was branch predictor state.

This theory was supported by the observation that if we inlined the CPS triage code (code that reads the CPS register and decides whether to retry), the problem went away in some cases. The inlining caused the

trriage code to be located near the transaction code itself, effectively preventing branch predictor aliasing between the transaction code and the triage code.

With this background, we can describe the most interesting scenario that we believe we encountered. In this case, we observed a red-black tree experiment failing hardware transactions repeatedly, with the CPS register set to UCTI|SIZ. This can occur as follows.

1. During a hardware transaction, a load is performed from a page for which there is no micro-DTLB mapping. This does not cause the transaction to fail: the destination register for the load instruction is set to “not there”, the load is deferred (i.e., placed into the deferred queue for subsequent reexecution when a micro-DTLB mapping has been established and the load miss is resolved), and speculative execution continues.
2. Next, a branch that depends on this register is encountered, and it is mispredicted and executed. Because we are traversing a red-black tree, many subsequently executed instructions depend on the load miss, and are therefore deferred, quickly filling the deferred queue.
3. Therefore, the transaction fails with the CPS register set to UCTI|SIZ. The SIZ bit is set due to the transaction overflowing the deferred queue, and the UCTI bit is set due to it executing a branch that depends on an outstanding cache miss.
4. The code executed during the misspeculation also evicts an entry from the micro-DTLB, setting the stage for repeating this scenario when the transaction is retried.
5. When the transaction fails, a branch is executed in the fail path that happens to alias to the branch predictor state used by the previously mispredicted branch. As a result, the branch predictor “forgets” what it learned about the branch that caused the misspeculation.
6. Now the stage is set to repeat the same pattern: the missing micro-DTLB mapping causes

the branch to be executed before the load on which it depends is resolved, and the branch predictor again mispredicts the branch.

Note that there are a number of possible variations on this theme. First, the above-described micro-DTLB interference might alternatively be caused by a memory access in the fail path, rather than during misspeculation.

Second, more complex examples might involve multiple branches and/or pages, so we could have longer cycles in which consecutive transaction attempts fail for different reasons. In fact, we have observed patterns of repeated failures with alternating CPS values.

Third, when a transaction traverses a dynamic data structure such as a red-black tree, some of the branches are data dependent, and therefore there is no “right” direction for the branch predictor to learn. This implies that we could encounter such scenarios even if we could successfully eliminate *all* memory accesses and branches from the fail path.

Finally, a related set of scenarios exists involving store instructions. When a store instruction targets a page for which there is no micro-DTLB mapping, this causes the transaction fail, setting the ST bit in the CPS register (see Section A.1). A request to establish the required micro-DTLB mapping is generated, and therefore a subsequent retry of the transaction can succeed after this request is fulfilled. However, as before, a memory access in the fail path could potentially interfere with the established mapping, resulting in repeated failures with the ST bit set.

We have identified a number of possible approaches that may help to alleviate pathological situations like those described above, including:

- Inlining the fail path to avoid branch predictor aliasing with transaction code.
- Structuring data to avoid excessive pressure on the micro-DTLB (for example, this investigation caused us to change the mapping of program data to orecs, because our original mapping ensured that if there was a conflict between two data addresses, there would also be a conflict

between the associated orecs, thus exacerbating the problem).

- Reducing memory accesses and especially mispredicted branches in the fail path.

While we have found all of these techniques to be useful in some cases, none of them is guaranteed to completely avoid pathologies in which transactions fail repeatedly even though the CPS feedback includes only failure reasons that should dissipate with several retries. In the next section, we describe a different technique that more reliably avoids the failure of transactions that do not exceed the resource constraints of the HTM feature.

7.3 Avoiding transaction failures by using speculation barriers

Repeated failure patterns such as those described above can be avoided by avoiding misspeculation and avoiding executing stores for which no micro-DTLB mapping is present. Rock includes some special instructions that can be used for this purpose. Specifically, the `brr` (branch if register ready) and `brnr` (branch if register not ready) instructions can be used to change the execution path if a given register is “not there”. The following simple use of the `brnr` instruction shows how we can avoid further speculation until a cache miss is resolved:

```
<load instruction in application>
brnr <destination register for load>, .
nop
```

If the load results in a miss, the `brnr` loop stalls execution until it is resolved, thus avoiding subsequent execution of a branch that depends on the outcome of the load before the load is resolved.

Similarly, the following shows how we can avoid executing a store to a location for which no micro-DTLB mapping exists.

```
<load from address of store below>
brnr <destination register for load>, .
nop
<store instruction from application>
```

This has the effect of performing a load from the location to which we intend to store, and then waiting until the load is completed. If there is no micro-DTLB mapping, the load will establish one (this does not fail the transaction, as a store to a location with no micro-DTLB mapping does; see Section A.1), and we will not proceed to attempt the store until the `brnr` loop confirms that this load has been resolved.

Note that there is no guarantee that load and store instructions instrumented as above will not fail the transaction, for at least two reasons:

- If there is no entry in the UTLB for a given location, then any access to that location will cause the transaction to fail. This can happen due to a *cold start*, i.e., the page has never been accessed before, or because a previous UTLB entry has been lost due to replacement or due to periodic Solaris memory management activities. In such cases, some *remediation* is needed before the location can be accessed in a successful transaction (see Section A.3).
- It is possible that the page has been accessed only for read in the past, in which case the store instrumentation trick described above may establish a micro-DTLB entry with read permission only, in which case the subsequent store instruction will fail the transaction.

Nonetheless, as discussed below (see Section 7.4), we have found that these techniques dramatically increase our ability to commit hardware transactions. In particular, it is interesting to note that it is *not* necessary for all pages to which a transaction stores to have mappings in the micro-DTLB simultaneously, as possible reasons for the ST bit being set seem to suggest (see reason 1 for the ST bit being set in Section A.1).

Using speculation barriers to allow function calls within transactions Our discussions with Rock architects revealed one more opportunity for using speculation barriers to prevent transactions from failing. As described in Section 3, Rock can fail a transaction that executes a `save` instruction and subsequently executes a `restore` instruction, a pattern

commonly associated with function calls. For technical reasons related to the resources Rock has available for checkpointing register state, it turns out that this class of transaction failure can be avoided by ensuring that the transaction is not speculating when it executes the `restore` instruction (at least in SST mode). A speculation barrier similar to those described above can be used for this purpose. Specifically, *any* `br` or `brnr` instruction (regardless of destination register) executed immediately after a `restore` instruction will induce Rock to wait for all speculation to be resolved before continuing. Note that it is essential for the `br` or `brnr` to immediately follow the `restore` in program order.

Because of SPARC branch delay slots and register conventions, some care is required. To illustrate, a typical return sequence, such as:

```
ret
restore
```

could be transformed into a sequence such as:

```
restore
brnr %g0, .
nop
retl
nop
```

Note that in the first sequence, the `restore` instruction is executed in the delay slot of the `ret` instruction. To allow use of a `brnr` instruction after the `restore` instruction, we can no longer put the `restore` in the delay slot of the `ret` instruction. Therefore, the return instruction must come after the `brnr` loop, requiring an additional `nop` to avoid it being executed in the delay slot of the `brnr` before the loop completes. Note also that we use a `retl` instruction for return rather than `ret` as in the first sequence above. This is because the `restore` instruction now executes *before* the return instruction, and therefore the return address is determined using the `%o7` register, rather than the `%i7` register, as it would be if the return instruction were executed before the `restore` instruction. Finally, the above sequence requires one more `nop` instruction after the `retl` instruction because we no longer have any instruction that we can place in the delay slot of the return instruction.

7.4 Results achieved using speculation barriers

In our continued work since publishing [9], several aspects of our environment have changed due to the following factors:

- Ongoing work on the compiler, which has been modified to support transactional language features, as opposed to the rudimentary syntax supported by the previous compiler.
- Ongoing work on the library.
- A new revision of the Rock chip, running at a slightly faster clock speed than the previous one, but still not rated for full speed.

As a result of these changes, our more recent results are not directly comparable to the previously published results. Nonetheless, we have learned more about Rock and applied these lessons while continuing our investigation. Below we discuss our experience.

Recall that as we moved to larger red-black trees and/or introduced even a small fraction of Insert and Delete operations, we started to experience significantly higher failure rates for hardware transactions, and that our detailed investigation strongly suggested that this behavior was not a result of transactions that exceeded hardware resources such as micro-DTLBs, caches, and store buffers.

Once we learned about the above-described speculation barriers, we were able to improve hardware transaction success rates considerably.

Our compiler produces two versions of code to be executed within transactions. The *undecorated* version is “normal” code that can be called outside transactions, and is also used by PhTM to execute hardware transactions. The *decorated* version instruments memory operations with calls to our library in order to facilitate execution with software transactional memory, as well as for detecting conflicts between concurrent hardware and software transactions in HyTM mode.

We did not want to install speculation barriers on every memory access in the undecorated versions, because this would impose considerable overhead both

on regular, nontransactional execution of such code, as well as on hardware transactions that would succeed even without them. So we modified the library to allow for execution of transactions in hardware using the decorated version, even in PhTM. Thus, we could experiment with trying on the undecorated path first — so that transactions that did not require speculation barriers to succeed would execute without additional overhead — but switching to using the decorated path in a hardware transaction if the transaction did not succeed on the undecorated path. If the hardware transaction fails in this case too, we resort to using STM.

This produced a considerable improvement to the success rate of hardware transactions in some workloads, thus significantly reducing how often we need to switch to software mode. Our work with these speculation barriers allowed us to achieve high hardware success transaction rates for significantly more challenging scenarios than previously. In particular, we were able to achieve high success rates on red-black trees with 16,384 keys, while failures were previously frequent with only 2,048 keys. Similarly, we can now increase the fraction of Insert and Delete operations to 10% of the operations and still achieve high success rates, while we were observing high failure rates with just 2% previously.

Our success in using these speculation barriers to eliminate high rates of hardware transaction failures with the ST bit set in the CPS register strongly suggests that these failures were due to speculation (see reason 4 for the ST bit being set in Section A.1), rather than to a micro-DTLB miss (reason 1) as we had previously guessed. This highlights the importance of detailed feedback about failure reasons; we spent a lot of time on the wrong theory because of this ambiguity.

Although we were able to achieve significantly higher success rates, performance did not improve as much as we had hoped, for at least two reasons. First, the overhead imposed by the speculation barriers is considerable, not only because of the additional instructions executed, but also because they eliminate the benefits of speculation, for example the ability to continue executing instructions while waiting for a cache miss to be resolved. As a result, in workloads

such as the large hash table and large red-black tree, in which cache misses are the norm, the cost of successfully committing a transaction using speculation barriers can exceed the cost of executing the transaction using STM, because the STM execution can still benefit from speculation.

Second, in cases in which speculation barriers are needed for success, the attempts we made on the undecorated path (three in our experiments) were simply wasted time. We therefore experimented with not trying at all on the undecorated path, and immediately attempting a hardware transaction using the decorated path and the speculation barriers. As expected, this resulted in considerable performance improvement in some cases. For example, with the red-black benchmark running on a large tree (keyrange=16,384), we saw performance improvements for most thread counts and mutator percentages, ranging from a few percent degradation to a speedup of a factor of 2.5.

However, in other workloads, it resulted in a significant *deterioration* in performance. This is not surprising, because it amounts to disabling speculative execution in transactions; doing so in a transaction that would have succeeded anyway causes a significant performance loss. For example, for a smaller tree (keyrange=128), using only the decorated path results in slowdown of a factor of about four for single thread runs, a factor of about 5.5 for 16-thread runs with no mutation, and a 30% slowdown for 16-thread runs with 2% Inserts and 2% Deletes.

We have not experimented extensively with different policies for determining how often to try in various modes and under which circumstances, but the observations described above clearly indicate that to achieve good performance across a variety of workloads, dynamic policies will be needed that can use speculation barriers when they are needed, and avoid them when they are not. This clearly presents some challenges, and even the best dynamic solution will undoubtedly impose at least some overhead and complexity on the software. Designers of future HTM features might consider whether different execution modes with different levels of speculation aggressiveness for the same code could alleviate this problem.

We have not yet entirely eliminated the possibility of a hardware transaction failing due to speculation. In particular, while our compiler usually instruments loads and stores inside transactions, it does not (yet) instrument memory accesses that are caused by spilling registers onto the stack and subsequently reloading them. An example of the consequences of this limitation arises in the discussion below.

We now turn our attention to the use of speculation barriers for `restore` instructions. Recall that, to achieve the red-black tree results described above, we had to use an iterative red-black tree implementation, because the recursive one with which we started failed due to function calls. When we learned of the possibility of circumventing this restriction, we resurrected the recursive version of the red-black tree code. We also asked our friends in the compiler group to modify the compiler to instrument restore instructions that could be executed in transactions, as described above.

Our “recursive” red-black tree implementation actually uses recursion only when modifying the structure of the tree: traversing the tree — either for a Lookup operation or to determine whether and where an item needs to be inserted or deleted — is done iteratively. Furthermore, when a Delete operation specifies a key that is not present, or when an Insert operation specifies a key that *is* present, no structural modification is made to the tree. Thus, only Inserts that “miss” and Deletes that “hit” in the tree require recursive calls. Without the use of speculation barriers on restore instructions, any time the processor happens to be speculating while executing a restore instruction associated with any such operation, the associated transaction would fail. This explains the poor performance we observed for even a small fraction of such operations using PhTM, because most operations that performed a structural modification to the tree required a switch to software mode.

When we added speculation barriers to the restore instructions, we observed a substantial decrease in hardware transaction failure rates. In particular, in a single-threaded, recursive red-black tree experiment with 128 keys, and 10% Inserts and 10% Deletes, 10% of operations resorted to using software when we did not use speculation barriers after restore instruc-

tions. This 10% almost exactly mirrors the number of operations that perform structural modifications to the tree: Because the tree is initialized to be about half full, and we have equal percentages of Insert and Delete operations, the tree is about half full throughout the experiment. Thus, roughly half of Delete operations do not modify the tree, and the other half delete a node. Similarly, half of Insert operations merely update the value of an already-present node, while the other half insert a node. The operations that insert or delete a node are the ones that make recursive calls, so about 10% of operations make recursive calls in this experiment. Of the 10% of operations that failed to software in the above-mentioned experiment, about 85% of them had the INST bit set on the last failed attempt to execute using a hardware transaction. This is explained by the recursive calls causing hardware transactions to fail, due to the **save-restore** limitation (see Section A.1).

In contrast, when we repeated the experiment with the speculation barriers after **restore** instructions, only about 1% of the operations failed to software. Amongst these, the CPS values of the last failed attempts were dominated by ST|SIZ (due to exceeding the store buffer limitation) and ST (due to uninstrumented stores to the stack, as discussed above).

Thus, using speculation barriers on **restore** instructions enables the use of function calls within transactions, which allows us to use recursive calls, library functions from different compilation units, and so on in transactions.

However, as with those for loads and stores, speculation barriers on restore instructions also impose significant overhead. An example illustrating the cost of such speculation barriers in a different context is presented in Section 9.2.

Furthermore, note that the use of speculation barriers on **restore** instructions does not entirely eliminate the limitation on calling functions. In particular, there is a limited number (eight in Rock) of register windows. Therefore, the maximum call depth is limited by that number. This is illustrated by repeating the single-threaded red-black experiments discussed above with increasing key ranges.

As we increase the key range, the trees become deeper, and therefore more operations make deeper

recursive calls. As mentioned above, the only recursive calls in our recursive red-black tree implementation are made when a node is inserted or deleted, and deep recursive calls result only from rotations that go far back up the tree in rebalancing the tree. Such extensive rotations are rare. Nonetheless, there is a noticeable increase in the number of transactions failing with the PREC bit set in the CPS register. This is what happens when a precise trap occurs, such as a spill-fill trap that occurs when the available register windows are exhausted.

We note that it is possible to ensure that enough register windows can be used without generating a spill-fill trap, up to the limit of the number of register windows. For example, by executing two **save** instructions followed by two **restore** instructions before executing a transaction, we can ensure that a call depth of three (including the current frame) can be achieved without generating a spill trap. Recall that Rock has eight register windows, so this trick only works with up to seven **save-restore** pairs.

Although this technique imposes unnecessary overhead when there are already enough register windows available, it may be useful in constructing specific transactions if it is highly desirable to avoid them failing. We do not use this technique in our HyTM and PhTM systems, because a failure to software will have the side effect of adjusting the register window so that the needed number of stack frames will be available the next time around.

Finally, our compiler allows us to inline the read and write barriers used by hardware transactions to check for conflicts with software transactions, and until recently we have done so, as it has yielded a significant performance benefit. The interaction between these barriers and the surrounding code has become slightly more complex recently, mostly due to (current) technical limitations of our compilation technology.

When we first experimented with HyTM in our new system, we observed that even simple transactions, such as those used by the HashTable benchmark, failed frequently with the ST bit set. After some investigation, we determined that register pressure was causing the problem. Our compiler (currently) has no good way to emit the **brnr** instruc-

tions that are needed for speculation barriers inline with surrounding code. We therefore had to pull the code fragments containing those instructions out into leaf routines. Doing so effectively reduces the size of the register set available to the surrounding code by about a factor of two (the SPARC defines `%o` and `%g` registers to be saved by the caller — any values contained therein can be overwritten by any procedure call). On top of that, inlining HyTM read and write barriers (calls to `CanHardwareRead()` and `CanHardwareWrite()`) proved to add too many temporary variables to the surrounding code, causing registers to be spilled to and restored from the stack. The stores and loads associated with these spills and restores are not currently instrumented with speculation barriers, so transactions can fail due to the address for the stores not being available due to speculation, which is consistent with the ST bit being set (see Section A.1).

We solved this problem (at a small cost) by extracting the HyTM barriers into leaf procedures as well. This has the effect of making the `%o` and `%g` registers available to the HyTM barriers, leaving the `%i` and `%l` registers available for the rest of the code. The current situation is obviously not optimal — once we teach the compiler to produce `brnr` instructions in line, we won't need to use leaf procedures for anything, so except for the very few temporary variables actually used by the HyTM barriers and by the speculation barriers, the entire register set will once again be available for use by the surrounding code.

8 Using hardware transactions to improve concurrent algorithms

Since our ASPLOS publication [9], we have extended our experimentation to accelerating existing concurrent algorithms using Rock's best-effort HTM directly, without using language-level transactions. This approach is applicable, for example, when some steps of a fine-grained, CAS-based algorithm can be combined into a hardware transaction. This approach can reduce synchronization overhead and in

some cases also allows the code in the hardware transaction to be simpler than the corresponding code in the underlying algorithm, because the underlying algorithm must deal with certain interleavings that are not possible when some steps are combined into a transaction. When a hardware transaction is not successful, the original code can be used as a software alternative. In Section 8.2, we describe our experience applying this approach to a scalable non-zero indicator (SNZI) [12] implementation in our SkySTM library [20].

The transactional compiler discussed in the previous sections only supports using STM as the software alternative, and thus is not applicable to the techniques described in this section. Therefore, we wrote the new algorithms directly, using Rock's new HTM instructions via compiler intrinsics and inline assembly. Unfortunately, this meant we could not rely on that compiler's support for inserting speculation barriers (described in Section 7.3).

We therefore developed a source-to-source compiler that supports a number of options for instrumenting transaction code in order to experiment with different ways of making transactions more likely to succeed. This compiler operates at the assembly level, reading and writing UltraSPARC® assembly files, and is used as a final compilation pass after regular compilation. It automatically detects the locations of hardware transactions, and can perform any combination of the following transformations (see Section 7.3 for background):

- Insert TLB-warming instrumentation before store instructions to prevent transactions from failing due to micro-DTLB misses.
- Insert `brnr` speculation barriers either after all load instructions (including loads from the stack), or alternatively, before any instructions that might cause the processor to speculate or fail a transaction (instructions we call *speculative instructions*).
- Insert nontransactional stores³ into each basic block that record the progress of a transaction.

³Rock's nontransactional stores occupy a slot in the store buffer, and thus can cause a transaction to fail by overflowing

This information can then be used to determine common failure locations within a transaction.

- Insert `brnr` speculation barriers after `restore` instructions to allow function calls within transactions.
- Render the control flow graph of a transaction as a PDF.

8.1 Implementation

The compiler is implemented as a Python script, which we hope to open source soon. For many of the listed options, the compiler simply inserts instrumentation before or after instructions of interest, relying only on global liveness analysis to find free registers that it can use for this purpose. Some care is needed when instrumenting instructions in branch delay slots, which usually requires reordering such instructions with respect to the preceding branch.

Additionally, the compiler uses a novel global data flow analysis to trace the flow of all data derived from the result of a load instruction. This analysis is used to identify speculative instructions whose input operands may not be ready at the time the instruction is issued, and to insert speculation barriers before them to prevent them from failing the transaction. This instrumentation is an alternative to inserting barriers after each load instruction. Both approaches prevent the processor from speculating on branch directions. However, the new approach offers three benefits:

- Delaying barriers takes advantage of instruction-level parallelism (between independent load instructions, for example) that would be lost if we placed the barriers after every load.
- Furthermore, the barriers are typically placed farther away from the load instructions, allowing more time for the loads to be satisfied before the barrier is encountered, thus reducing or eliminating the time spent waiting at the barrier.

the store buffer. Therefore, the technique described in this paragraph is limited, though we have found it to be quite useful in some cases.

- Finally, fewer barriers may be executed because an input register to a speculative instruction may be derived from two or more load instructions.

The following code snippets illustrate these points. The following code has two loads followed by a compare and branch:

```
ldx    [%12], %g5
ldx    [%i3+8], %g1
cmp    %g5, %g1
be,pn  xcc,.L77001299
nop
```

The instrumented code contains two speculation barriers, which are inserted after both loads (rather than inserting each barrier immediately after the corresponding load), thereby allowing the two loads to execute in parallel:

```
ldx    [%12], %g5
ldx    [%i3+8], %g1
brnr,pn %g5, .
nop
brnr,pn %g1, .
nop
cmp    %g5, %g1
be,pn  xcc,.L77001299
nop
```

Note that, although the `cmp` instruction is not a speculative instruction, we treat it as one because there is no way to wait until a condition code (which is set by such instructions) is ready. Therefore, we instead wait for the source operand(s) of such instructions to be ready before executing them.

The following code has two loads and compares their sum to 0, which is followed by a branch:

```
ldx    [%12], %g5
ldx    [%i3+8], %g1
add    %g1, %g5, %g2
cmp    %g2, 0
be,pn  xcc,.L77001300
nop
```

The instrumented code requires only one barrier right before the compare instruction, despite the fact that the branch depends on data from two loads:

```

ldx    [%12], %g5
ldx    [i3+8], %g1
add    %g1, %g5, %g2
brnr,pn %g2, .
nop
cmp    %g2, 0
be,pn  xcc,.L77001300
nop

```

As discussed in the next section, these two benefits can significantly lower the cost of the speculation barriers.

The analysis is implemented as an iterative forward direction flow analysis, which repeatedly traces the flow of data derived from each load instruction in a transaction until a fixed state is reached. For each program point, the compiler computes a set of *derivation groups*. Each such group is associated with a unique load instruction and represents the set of registers whose values at the given program point potentially depend on the data loaded by that load instruction.

The analysis uses a union *join* operator that, for every basic block, combines the sets of derivation groups at the ends of the basic block’s predecessors, in order to determine the set of derivation groups for the start of the basic block. If two predecessors have a derivation group associated with the same load instruction in their set, then the contents of the derivation groups are combined. Finally, the *transfer function*, which is used to determine the set of derivation groups reachable at the end of each basic block, examines each instruction in the basic block in the forward direction and performs the following:

- For every load instruction, a new derivation group containing the load’s destination register is added to the set of derivation groups for the subsequent instruction.
- For every speculative instruction, all derivation groups for that program point are examined to see if they contain a register used by the instruction.

If so, for each such register, the instruction is marked as requiring a barrier for that register

(an instruction may require more than one barrier, one for each input register). Subsequently, all derivation groups containing this register are removed from the set of derivation groups (thus, subsequent instructions using any of the derived registers in the same derivation group as this register will not need a second barrier). We note that it is not necessarily the case that, because we wait for one register in a derivation group to be ready, all other registers in that group are also ready. Nonetheless, because all other registers in the group depend on the same load as the register for which a speculation barrier will be emitted, it is likely that they too will soon be ready. We found that this approximation simplified the analysis and resulted in much fewer speculation barriers being emitted, and better performance.

Otherwise, if the instruction uses input registers not present in any of its derivation groups, the instruction is marked as not requiring a barrier. This is necessary because the instruction may have been marked as requiring a speculation barrier during a previous iteration, but this barrier may have become unnecessary due to speculation barriers introduced in subsequent iterations.

- For every other instruction with a destination operand, the compiler removes the destination register from all derivation groups and adds it back to all derivation groups containing any of the source operands. This reflects the fact that the previous value of the destination register is overwritten, so we no longer need to track the set of registers on which this previous value may depend, but the new contents of the register now potentially depend on the results of any load on which the source operands potentially depend.

8.2 Experience

We have experimented with optimizing various parts of the SkySTM library [20] using Rock’s hardware transactions. In this section, we describe our experience with accelerating the SNZI data structure [12], which is used to acquire and release semi-visible read

ownership of SkySTM’s transaction orecs in a scalable manner. We have implemented accelerated versions of both the SNZI Arrive and Depart operations, which first attempt to modify the SNZI tree in one atomic operation using the HTM, and if that fails repeatedly, fall back to the original CAS-based implementations. We found the HTM parts of the new algorithms to be significantly simpler than the originals, although the overall complexity of the code increased since the old version remained. This also requires us to reason about the correctness of having both the original and HTM versions executing concurrently.

We evaluated our work by measuring the cumulative cost of all the Depart operations performed by the SkySTM library when running the red black tree benchmark with 1 and 16 threads on Rock. With a single thread, the new accelerated version ran 27% faster than the original code. When running with 16 threads, the performance gain was a slightly more modest 17% improvement. These encouraging results were achieved even though, in both cases, the transactions failed an average of 1.31 times before either completing successfully or falling back on the software alternative. This led us to believe that more performance could be achieved.

We found that many of the transaction failures were caused when the processor tried to execute branch and store instructions that depended on data that was still outstanding due to a cache miss. Thus, our initial solution was to attempt to prefetch this data before the start of the transaction by hand coding source code that would prefetch relevant parts of the SNZI tree. This improved the single-threaded performance by a further 26%, but had no impact on performance in the 16-threaded experiments. Thus, we used our new source-to-source compiler to insert speculation barriers after every load instruction inside a transaction. This lowered the failure rate to around 4% but significantly increased the single-threaded and 16-threaded execution times, so they were slower than even the original unaccelerated algorithm.

Applying our new data flow analysis, we were able to push down and eliminate over three quarters of the barriers. At 16 threads, the resulting code ran

9% faster than the instrumentation-free prefetched version, with a failure rate of less than 3%, bringing the total improvement over the original code to 27%. Unfortunately, with one thread, the performance of this version was 24% slower than the version with prefetching (though still 30% faster than the original CAS version). This result matches our wider observations about using speculation barriers: while they are very useful for improving transaction success rates, in some cases their high cost may be better avoided by using other techniques such as prefetching, if possible.

9 Transactional lock elision

In this section we report on our experience using transactional lock elision (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 [33, 34] 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. Furthermore, because candidate critical sections for elision are determined by software, possibilities are available for TLE that are not possible with SLE. For example, in a managed runtime such as a JVM, synchronization operations can be inlined into application code, and optimized on a case-by-case basis to improve latency for lightly contended critical

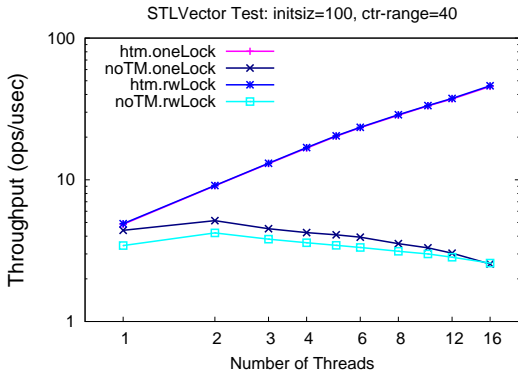


Figure 6: TLE in C++ with STL vector.

sections and throughput for more heavily contended ones. Furthermore, adaptive policies can tailor retry decisions to a given workload. SLE does not have such flexibility because the software is unaware of it.

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.

9.1 TLE with C++ STL vector

We repeated the experiment described in [8], which uses simple macro wrappers to apply TLE to an unmodified STL vector. This experiment uses a 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, rather than the 10:10:80 used in [8]. 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 conducted a detailed analysis of the reasons for requiring more retries, but it is likely due to similar reasons as discussed in Section 6. Specifically,

cache misses lead to transaction failures for various reasons on Rock (but not on the Adaptive Transactional Memory Test Platform (ATMTP) [28, 8]). Because even failing transactions issue cache requests, subsequent retries can succeed when the cache request is fulfilled. Even using the simplistic policy described above, our results (Figure 6) show excellent scalability using TLE, in contrast to negative scalability without.

9.2 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 just-in-time (JIT) compiler can use runtime 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 [8], 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 lock. We describe our implementation in more detail below.

Each TLE attempt starts by initializing a thread-specific `RetryCredit` variable, which is debited by various events, as described below. The thread repeatedly attempts to execute the critical section using a hardware transaction until the `RetryCredit` variable becomes negative. To do so, the thread first executes a `chkpt` instruction to start a hardware transaction and fetches the word that represents

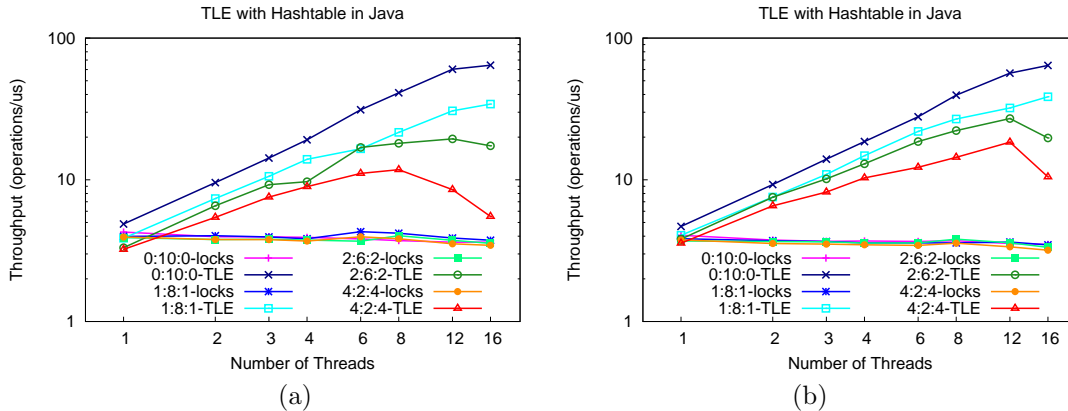


Figure 7: TLE in Java with `Hashtable` — (a) previous results from [9] (b) results with arithmetic backoff and tuning.

the state of the monitor. If the object is unlocked, then the thread tries to execute the critical section transactionally. (In the corresponding `monitorexit` operation, if the critical section was executed transactionally the thread simply executes `commit`, otherwise it invokes the normal `monitorexit` code to release the lock.) If the object was found to be locked, however, the thread commits the transaction (which has thus far not modified any state), and then debits the `RetryCredit` variable by a constant (parameterized for manual tuning and automatic adaptive operation).

If `RetryCredit` becomes negative, the thread reverts to normal mutual exclusion, where, if necessary, the thread blocks in order to yield the processor to other runnable threads. Otherwise the thread delays using a backoff algorithm and then retries the TLE attempt, branching to the code block described above that executes the `chkpt` instruction.

If the hardware transaction aborts, the processor rolls back state and diverts control to the location designated by the `chkpt` instruction, which reads the CPS register, and uses the value read to determine how to react. If the COH bit is set, indicating a conflict, the thread debits the `RetryCredit` variable. If `RetryCredit` remains non-negative, the thread branches to the backoff and retry code, as ex-

plained above. If the COH bit is not set, the thread debits the `RetryCredit` variable, and, if it remains non-negative, retries immediately.

9.3 Experiments with TLE in Java

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 [8], 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 also conducted more ambitious tests that include operations that modify the collection. Results from [9] for `Hashtable` are shown in Figure 7(a); 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.

Since publishing [9], we have experimented further and achieved considerably better results, shown in Figure 7(b), by using arithmetic backoff instead of the exponential backoff used previously, and by further tuning of the initial value of the `RetryCredit` variable and of the amounts debited from it due to coherence and noncoherence failures. While we still observe some degradation in throughput at higher threading levels in the experiments with higher levels of mutation, this is significantly less dramatic than our previous data showed (Figure 7(a)). This underscores the importance of having the flexibility to tune contention management strategies. Furthermore, different locks and different locking sites are likely to achieve best performance with different parameters; our prototype applies a single set of parameters across all locks and sites.

We have also experimented further with using the speculation barriers described in Section 7.2 to mitigate the effects of the `save-restore` limitation mentioned in Section 3. Our JVM supports an option to execute a speculation barrier after each `restore` instruction. To explore the cost of these barriers, we experimented with enabling and disabling this feature, as well as enabling and disabling of inlining. In these experiments, a single thread performed 20% `puts`, 60% `gets`, and 20% `removes`, and the TLE feature was disabled. Thus, these experiments focus only on the performance impact of the speculation barriers, not interactions with transactional success.

With inlining enabled, performing speculation barriers after `restore` instructions had a modest effect on performance: we observed throughput of 3,640 operations/ms without the barriers, and 3,312 operations/ms with them. However, with inlining disabled, the impact of the speculation barriers was much more profound: we observed 3,464 operations/ms without the speculation barriers, but only 2,461 operations/ms with speculation barriers enabled, almost a 30% drop in throughput.

Thus there is a significant cost associated with these speculation barriers. Nonetheless, in some cases the cost is worth paying. For example, in some cases we have found that we can usually achieve better performance without these barriers enabled, but that occasionally an inlining decision would be changed, with the result that all transactions would start failing, causing a severe performance degradation. Inlining decisions are affected by a number of factors, and do not currently take transactions into account at all. Therefore, we can make the code less susceptible to occasional adverse inlining decisions by enabling the speculation barriers after `restore` instructions, at the price of lower performance in the common case in which the “right” inlining decision has been made, so the barrier is not necessary to prevent the transaction failing.

We have also conducted similar experiments as described above for `Hashtable`, using `HashMap`. As before [8], 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 executes a function call to a nonleaf method, which — as discussed in Sec-

tion 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.

9.4 TLE in more realistic applications

We are of course interested in applying Rock’s HTM to improve the performance and scalability of more realistic applications than the microbenchmarks we have used to evaluate our progress to date.

In contrast to our carefully constructed microbenchmarks, real applications are unlikely to be so uniform in their ability to benefit from TLE. Some critical sections will be too large, others will contain instructions not supported in transactions, and others may conflict sufficiently often that TLE provides no benefit. When TLE is attempted and is not successful, it will likely result in performance degradation. Therefore, we believe TLE must be applied *selectively* to be useful in general. We face several significant challenges in our ongoing work towards making the JVM adaptively apply TLE only where it is profitable.

Success with TLE depends on the application code, the code generated by the JIT compiler, capabilities of the hardware, and ability of the JVM to adaptively enable TLE only when it is useful. We are interested both in determining how well we can exploit TLE in the short term given the realities of Rock’s capabilities and limitations on software infrastructure work, as well as how well we could exploit TLE with more sophisticated software support and/or HTM support longer term.

As a first step with more realistic applications, we have experimented with the VolanoMark benchmark [31], a Java benchmark suite that emulates a chat room server. It stresses socket I/O, object allocation and collection, and synchronization (both contended and uncontended).

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 describe this work in more detail below.

9.5 Observability infrastructure for studying TLE

A necessary first step towards exploiting TLE in real applications is identifying applications that potentially benefit from TLE and understanding whether they are likely to do so, given various levels of software and/or hardware support. Moving from microbenchmarks to even moderate-sized realistic benchmarks or applications, it quickly becomes impractical to manually examine compiled code in order to judge its chances of benefiting from TLE.

We have therefore 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. So far, we have introduced instrumentation for several events that a) we can intercept without severely distorting execution up to the interception point and b) may yield useful information that helps us judge TLE feasibility. Below we describe the JVM and discuss some of the observations

we have made with it to date.

Our JVM uses a combination of Solaris™ Dynamic Tracing Framework (DTrace) [26] probes and JVM modifications to gather useful information at various points in a program’s execution. These points include a number of events relevant to synchronization, such as an attempt to acquire a lock that is already held, as well as success and failures of attempted hardware transactions. Many of the events we can monitor are not related to hardware transactions and thus we can use our tool to explore TLE feasibility even on systems without HTM support.

To get useful information about an application’s synchronization behavior, we must avoid excessive instrumentation, which can change the application’s behavior sufficiently that useful conclusions cannot be drawn. Furthermore, we are limited by what events we can conveniently profile. Nonetheless, we have already been able to gain some useful insights using the tool, and we expect to be able to enhance it over time.

The tool and our work with it to date are quite preliminary. Below we share some of what we have learned so far, by using the tool to explore VolanoMark [31] and the Derby database component of SPECjvm2008 [39].

One of our probes fires when an attempt is made to acquire a lock that is already held. When the probe fires, the tool prints a stack trace with various useful information, including address and type of the object being accessed, program counter, thread id, state of the object monitor, and thread id of its owner (if any). The address range of the current stack frame is also printed, allowing us to infer which methods have been inlined.

From this information, we have been able to make some useful observations. First, in some cases we find that the object to be locked is already locked by the thread attempting to lock it (namely, recursive locking). Our simple TLE-enabled JVM does not attempt to remove nested locking attempts, and is thus unable to elide nested locks.

In other cases, we see that the lock is not held. Given that the probe fired because it was locked, we can conclude that this is a case of contention on the lock: it was held when the acquire attempt oc-

curred and because it is no longer held, we can infer that it was locked by another thread. This allows us to focus on an object and method that encountered contention, which may indicate a TLE opportunity. However, when we then dig deeper into what is going on inside the critical section, we often see reasons why TLE is unlikely to be successful.

One example from VolanoMark is a method that invokes the `removeAllElements` method of a `java.util.Vector` object. This library method stores to all elements of the vector, and thus the critical section is likely to perform too many stores to succeed in a Rock transaction.

Another example is an increment of a single counter by many critical sections, which causes conflicts between concurrent attempts to elide such critical sections using TLE. Simple restructuring of the application code may eliminate such effects, but we have not explored this avenue (VolanoMark sources are obfuscated, which makes it difficult to draw the observations above, which generally must be inferred from stack traces, and also to experiment with changes to the application).

Another example is a call to a method that is not inlined, and therefore TLE would ordinarily fail due to the `save-restore` restriction mentioned in Section 3. One common and interesting instance of this is a call to `notify`. In VolanoMark, many critical sections call `notify`, but few of them actually wake up any threads. Therefore, if we could inline the `notify` code at least for this case, we could eliminate one impediment to successful TLE from many critical sections. In Derby we have observed that many critical sections include calls to `notify` that actually do wake up other threads. Making such critical sections amenable to TLE would involve significant JVM work, which we have not attempted to date.

9.6 Ideas for further improving TLE

Our preliminary work described above seems fairly discouraging for TLE in realistic applications, at least given the limitations of Rock’s HTM and of our JVM prototype. However, it is important to realize that this exploration is quite preliminary, and that many of the reasons that transactions are expected to fail

are not fundamental to the approach: they may be addressed by a TLE-aware JIT compiler that knows characteristics of code that can succeed in transactions. Other cases may be successful with enhanced HTM support, for example with less constraints on the number of stores in a transaction and instructions that are supported in transactions.

Furthermore, more sophisticated synchronization mechanisms may improve the effectiveness of TLE. For example, in our current implementation, once the thread has reverted to the traditional mutual exclusion path, it will ultimately acquire the lock in the traditional manner. This strategy can lead to the “lemming effect”, in which other threads revert to the lock because it is held by the first, and this situation persists, even though successful transactional execution is still possible. To avoid this condition, we believe that a “gang wakeup” mechanism, whereby the synchronization subsystem periodically wakes up sets of threads blocked on a lock and causes them to retry transactional entry, may be effective.

In addition, locks that suffer high coherence aborts may revert to either normal mutual exclusion (because of repeated failed retries) or, if they sustain transactional execution, degrade to a throughput level below that obtained with mutual exclusion. In that case, adaptively limiting concurrency on the lock, for example by protecting it with a k -exclusion wrapper construct, might restrict concurrency to a level that TLE is again profitable in comparison to classic mutual exclusion. Using this technique, some set of k threads would be allowed to attempt TLE entry, while the remaining threads would be spinning or blocked. The synchronization system could adapt k to maximize throughput over the lock.

Despite the challenges, we remain hopeful of achieving benefits from TLE in Java on Rock. In the next section, we present a successful experiment using TLE on Rock in C.

10 Minimum Spanning Forest algorithm

Kang and Bader [18] 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 [18]. 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 [20], we noticed that the main transaction was unlikely to succeed using Rock’s HTM, for two reasons: first, the transaction becomes too big, mostly because of the heap `extract-min` operation. Second, the `extract-min` operation is similar to the red-black tree (Section 7) in that it traverses dynamic data, thus confounding 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 straightforward to avoid extracting the minimum inside the transaction, instead doing it right after the transaction commits and the heap is privately 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 nontransactionally (in Cases 1 and 3). This demonstrates one of the most valuable advantages of transactional programming. Extracting the minimum nontransactionally 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 requiring us to modify the complex and less-frequent cases.

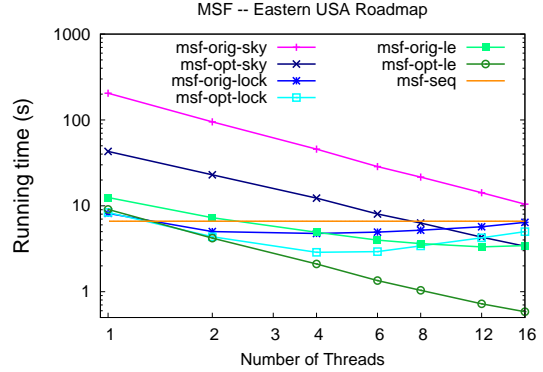


Figure 8: MSF

10.1 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 seven 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 [20].

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 eight failures, where a transaction that fails with the UCTI bit set in the CPS register is only counted as half a failure. If the transaction does not succeed before eight failures counted this way occur, we acquire the 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 (www.dis.uniroma1.it/~challenge9), which has 3,598,623 nodes and 8,778,114 edges. Figure 8 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 six 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, compared 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 greater than 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 nontransactionally in some cases, as well as the significant performance improvement that can be gained by executing transactions in hardware.

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

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 four 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 five times longer than the best time achieved by the `msf-opt-le` version.

Finally, we also ran the experiments on Rock configured in Scout mode. As we expected, in Scout 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 Scout 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 Scout mode `msf-opt-le` provides “only” a 9x speedup with 16 threads, and stops scaling at all after that point. Still, even in Scout mode, the optimized variant scales better than the original one, and with 16 threads is almost three times faster than any of the software methods at any number of threads.

11 Discussion

Our investigation to date has yielded some very encouraging results, suggesting 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.

As discussed below, HTM-aware compilers and tools can improve the likelihood of hardware transactions succeeding by using knowledge of the HTM feature. However, we would like to be able to exploit features like Rock’s HTM in as many contexts as possible, so it is preferable to avoid dependence on platform-specific compiler transformations and optimizations to the extent possible. Therefore, it is important that HTM is able to execute ordinary code generated by any compiler, including preexisting legacy code in some cases. Rock’s HTM interface was designed with this requirement in mind. Specifically, once a transaction is begun (with a `chkpt` instruction), ordinary instructions are executed by the transaction. This allows us to implement simple wrappers, such as described in Section 9.1, that programmers can use in any context to execute hardware transactions. In contrast, some other proposals (e.g., [1]) require special memory instructions for transactions, thereby limiting their use to special compilers

or hand-crafted code.

Despite the encouraging results we have achieved to date, we have also faced a number of challenges. In some cases, it was difficult to diagnose reasons for transaction failures, and difficult to overcome them when we did. In other cases, we had to rely on software-controlled speculation barriers to get transactions to succeed. This requires specific compiler support and the use of Rock-specific features, which limits the applicability of these approaches. It also imposes significant overhead, in some cases so much that we could not achieve a performance improvement over alternative software-only mechanisms.

Our conclusion is that the Rock architects have made a significant step towards sophisticated hardware support for facilitating effective, scalable synchronization in multicore systems, but there is still plenty of room for improvement. In this section, we summarize and discuss aspects that designers of future HTM features should consider. After that, we summarize ways in which software might be improved to better exploit best-effort HTM features in general and/or Rock’s HTM feature in particular.

11.1 Potential hardware improvements for future HTM features

As described in detail in Section A.1, Rock transactions can fail for a number of reasons. Some of these reasons are not fundamental, and are likely to occur in code that might naturally be included in a transaction. While such failures are by definition acceptable in a best-effort HTM, they limit the utility of the feature, and thus it is desirable to avoid such failures to the extent possible.

When transactions do fail, Rock’s CPS register provides valuable feedback as to the reason why (again, see Section A.1). However, it is sometimes quite difficult to definitively determine the reason for a transaction failing, for example because multiple failure reasons result in the same feedback in the CPS register. Even when we can identify the reason for a transaction failing, in some cases it is very difficult to address the problem and cause the transaction to succeed.

The difficulty of diagnosing transaction failures in some cases clearly points to the importance of richer feedback in future HTM features. Apart from further disambiguating different reasons for failure, additional information such as the program counter of failing instructions, addresses for data conflicts and TLB misses, time spent in a failed transaction, and so on 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. For another example, as discussed in Section A.3, it is quite difficult in general to remediate for transactions that fail due to TLB misses. Future HTM features should either avoid failing in such cases, or at least ensure that the TLB miss is resolved when the transaction fails, or provide better feedback to allow software to remediate before retrying.

Some failure reasons are problematic in some contexts, even if they can be easily identified and remediation is easy or unnecessary. For example, cache misses can result in transactions failing in Rock due to misspeculation. Although we can simply retry in such cases, the overhead of repeating the work of failed transactions can in some cases outweigh the benefit of the eventual successful transaction. In other cases, special instrumentation is required to get a transaction to succeed, as discussed in Section 7.3. This instrumentation requires compiler support, and it is difficult for software to determine which accesses need to be instrumented to cause a transaction to succeed. This has led us to conservative approaches that carry significant overhead, sometimes enough that the cost outweighs the benefit. These observations suggest that designers of future HTM features might consider including the ability to execute the same code with varying degrees of aggressiveness with respect to speculation, controlled by hardware or through a software interface.

Designers of future HTM features should take care regarding the interaction of transactions and traps. On one hand, there is no “forward progress” issue with Rock’s approach of aborting the transaction and ignoring the trap, because the HTM is best-effort and thus software must always be prepared with an alternative to execute in the case of repeated fail-

ures. Nonetheless, applying this argument to justify ignoring traps that occur during transactions can significantly undermine the value of the feature. For example, as we discussed previously, in some cases the need for and difficulty of remediation to address TLB misses in transactions can significantly reduce our chances of successfully using the HTM feature. We therefore believe that future HTM features should not simply ignore traps.

An alternative approach [1] is to abort the transaction and then deliver the trap. Some care is required with this approach. Consider a transaction that performs a divide by zero. One might consider it appropriate to deliver the trap as usual. However, given that any changes to memory performed in the transaction before the divide by zero will be rolled back, the effect would actually not be just “as usual”. Furthermore, if the trap occurred due to misspeculation, and could not have happened without misspeculation, there is no bug in the user’s program, and delivering the trap is clearly not the right thing to do. Thus, this issue requires careful treatment.

We think the best approach may be to ensure that information about the trap is made available to software after the transaction fails, so that it can determine which kinds of traps should be delivered in which circumstances. Note that, in a best-effort HTM, software can always be conservative and not deliver the trap when there is any doubt, but excessive conservatism will likely undermine the effectiveness of the feature, as discussed above.

Rock supports some performance counters that can assist with studying the performance characteristics of transactions. We have not experimented extensively with them to date, but we mention them here as a reminder to designers to consider interactions between an HTM and performance mechanisms.

Rock hardware transactions are difficult to debug, because it is not possible to interrupt a transaction without aborting it and rolling back its effects, thus losing any information about what it did. While transactional programs supported using HyTM or PhTM can be debugged by using the alternative software code path [22], this does not help when we wish to debug the code actually executed in a hardware transaction, for example to look for compiler

bugs, or to debug hand-coded transactions. Therefore, designers of future HTM features should consider whether hardware transactions could be executed in a mode that supports debugging them.

As mentioned in Section 8, Rock’s capability to execute nontransactional store instructions inside transactions is rather limited, and in particular, nontransactional stores are subject to the store buffer size limitation. While this feature is very useful for some purposes, other natural uses are not possible due to this limitation.

The freedom that the best-effort approach gives to HTM designers has been key to achieving Rock’s HTM implementation: it allowed the designers to “punt” on cases that were too difficult, or were not considered common enough to merit additional complexity. Furthermore, software that uses it can automatically benefit from future “even better” efforts. However, as discussed in Section 5 and in [40], there are significant advantages to providing guarantees for simple, small transactions such that no software alternative is needed. Based on our experience, we believe designers of future HTM features should consider such guarantees as early design goals.

The remaining ways in which future HTM support could improve over Rock’s are related to the instruction interface.

Programmers should not count on particular limitations causing a transaction to fail, because future improved implementations might not have the same limitation. Therefore, it is important to provide a specific means for aborting transactions that is guaranteed to work for this purpose in future implementations, too. To date, this has not been done for Rock, and we have exploited knowledge of specific limitations to abort transactions, including traps (in particular, by convention we often use the following unconditional trap instruction for this purpose: `ta %xcc, %g0 + 15`). It may be better to use conditional traps in order to avoid unnecessary branches, but we have not experimented extensively with this.

In some cases, it is useful to be able to abort a transaction, while not affecting the same code when run outside a transaction. This can be useful, for example, for early pruning of code paths that are likely to fail the transaction anyway. For this purpose, we

have used `membar #nop`: this instruction fails a transaction that executes it, but is benign on nontransactional execution. If a future implementation were to eliminate this limitation, that would break code that uses this technique. One possibility is to architecturally define `membar #nop` as failing transactions, rather than simply considering it as a limitation of the current implementation.

It is also important that the triage code that reacts to a transaction failure can determine whether it was explicitly aborted, for example because that may mean it should not be retried. Today we depend on knowledge of the CPS register and of the compiled code to infer this, but future designs should ensure that this is possible, preferably in a uniform way. We note that one can also use nontransactional stores to record that the transaction will be explicitly aborted, and to communicate more about the reason for the abort. But this is not ideal, in part because it adds overhead and complexity to the common case. Ideally, we would have an explicit abort that would allow us to communicate some information out of the transaction.

Furthermore, we believe it would be useful to provide a means for code to determine whether it is running in a transaction, for example to elide (unnecessary) `membar` instructions when inside transactions, to avoid failing the transaction.

11.2 Potential software improvements for better exploiting HTM features

Numerous possibilities exist for an HTM-aware compiler to improve the performance of programs that use hardware transactions, especially by increasing the likelihood of transactions succeeding. These possibilities include optimizations, transformations, and code generation choices that are better for transactions in general, such as moving memory accesses later in the transaction to reduce the window of vulnerability to transactions.

An HTM-aware compiler could also be aware of specific limitations of a particular HTM feature. For example, in contexts such as the JVM where run-

time inlining is possible, code can be inlined, even from different modules, in order to circumvent Rock's `save-restore` limitation (Section 3). Even in more static contexts, a Rock-aware compiler could avoid using these instructions for function calls, much like GCC's (now deprecated) `-mflat` option. Note, however, that this approach forces a function to save and restore registers it will use, resulting in additional stores and thus potentially causing a transaction to fail because it exceeds the maximum number of stores that can be executed in a transaction.

As another example, in many cases code can be factored out of a transaction, thus reducing the length of the transaction and possibly removing instructions from the transaction that are likely to cause it to fail. For example, Rock's limitation on executing instructions such as `sdiv` in transactions can sometimes be circumvented by preloading the values to be divided, performing the division outside the transaction, and then simply loading the values again inside the transaction and confirming that they are the same as the preloaded values. Similarly, in code that is known to be executed only in hardware transactions, CAS instructions can be replaced by an equivalent load-compare-store sequence, thus eliminating the use of the CAS instruction, which is not supported in hardware transactions on Rock.

In some cases, counterintuitive code transformations that seem to make code slower can actually improve performance in concurrent code, and in transactional code in particular. One example is "silent store elision": by checking if a value to be stored to a memory location is equal to the value already stored at that location, the store can be avoided. While the load and compare make the code path longer than a simple unconditional store, this transformation can eliminate conflicts between transactions and reduce coherence traffic, both of which can contribute to better throughput.

Similarly, as mentioned in Section 9, in some cases when excessive conflicts between transactions cause excessive coherence traffic and retries, better throughput may be achieved by "throttling" concurrency down to a level where we can successfully exploit parallelism without running into memory bottlenecks, retry storms, and so on.

In our work with Rock, we have concentrated most on understanding and evaluating the hardware, and on providing proof-of-concept demonstrations of various techniques that use the HTM feature. We have done very little optimization, and we have not explored most of the ideas discussed above in great detail. We believe that with improved software support, it will be possible to considerably improve on several of the results we have achieved to date. Nonetheless, some limitations can only be overcome by enhanced hardware support for HTM, and we hope this technical report provides valuable input and guidance for designers of such features.

12 Concluding remarks

We have described our experience evaluating the hardware transactional memory feature of Sun's 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, as well as possible directions for improvement of software that uses Rock's HTM feature. We hope this paper will be useful both to programmers who use Rock's HTM and to architects designing related features in the future.

We often hear claims that people claim that TM will 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 this 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.

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 Virendra Marathe for the fine-grain lock-based hash table microbenchmark. Finally, we are grateful to our coauthors on the two Transact papers [28, 8] on the ATMTTP simulator and techniques developed to exploiting HTM in Rock.

References

- [1] Advanced Micro Devices. Advanced synchronization facility proposed architectural specification, March 2009. Publication # 45432, Revision: 2.1.
- [2] 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.
- [3] 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.
- [4] 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*, 29(2):6–16, 2009.
- [5] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: a novel pipeline architecture implemented in Sun’s Rock processor. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 484–495, New York, NY, USA, 2009. ACM.
- [6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS '06: Proceedings of the 12th Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.
- [7] D. Detlefs, P. Martin, M. Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 2001.
- [8] 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*, 2008. research.sun.com/scalable/pubs/TRANSACT2008-ATMTP-Apps.pdf.
- [9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, New York, NY, USA, 2009. ACM.
- [10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*, 2006.
- [11] S. Doherty, D. Detlefs, L. Groves, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. Steele. DCAS is not a silver bullet for nonblocking synchronization. In *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures*. ACM Press, June 2004.
- [12] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC '07: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2007.
- [13] K. Fraser. *Practical Lock-Freedom*. PhD thesis, Cambridge University Technical Report UCAM-CL-TR-579, Cambridge, England, February 2004.
- [14] M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*. ACM Press, 2002.
- [15] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and system

- design. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 123–136. ACM Press, 1996.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006.
- [20] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Workshop on Transactional Computing (Transact)*, February 2009. research.sun.com/scalable/pubs/TRANSACT2009-ScalableSTMANatomy.pdf.
- [21] 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.
- [22] Y. Lev and M. Moir. Debugging with transactional memory. In *Workshop on Transactional Computing (Transact)*, June 2006. research.sun.com/scalable/pubs/TRANSACT2006-debugging.pdf.
- [23] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007. research.sun.com/scalable/pubs/TRANSACT2007-PhTM.pdf.
- [24] 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.
- [25] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. In *19th International Symposium on Distributed Computing*, 2005.
- [26] R. McDougall, J. Mauro, and B. Gregg. *Solaris(TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
- [27] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *14th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2002.
- [28] 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 Computing (Transact)*, 2008. research.sun.com/scalable/pubs/TRANSACT2008-ATMTP.pdf.
- [29] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming*

- languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.
- [30] 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.
- [31] J. Neffenger. The volano report, May 2003. www.volano.com/report/index.html.
- [32] V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical Report 2009-12, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, September 2009. www.rz.uni-karlsruhe.de/~kb95/papers/pankratius-TMStudy.pdf.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 8th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2009. www.cs.utexas.edu/users/rossbach/pubs/wddd09-rossbach.pdf.
- [37] 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.
- [38] Sparc International, Inc. The SPARC architecture manual, version 8, 1991.
- [39] SPECjvm2008. www.spec.org/jvm2008.
- [40] F. Tabbà, M. Moir, J. R. Goodman, A. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *ACM Symposium on Parallelism in Architectures and Algorithms*, 2009.
- [41] 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.

A Detailed discussion of CPS register

Table 1 in Section 2 provides *example* reasons for each of the CPS bits being set, but it does not list all reasons for each bit, and furthermore does not describe situations in which multiple bits may be set. Below, we provide a detailed description for each CPS bit, enumerating possible reasons for the bit being set, relationships with other CPS bits, and discuss how software should react in each case.

For concreteness, we then present schematic pseudocode illustrating CPS triage logic that decides how to react to each transaction failure. Our purpose here is to present as much information as we can to help programmers understand the CPS register, and to illustrate an approach to using it. We do *not* claim that the presented CPS triage logic is optimal for any particular situation.

A.1 Reasons for CPS bits being set

Below we discuss each of the CPS bits in turn, listing all⁴ reasons the bit might be set, as well as comments on how common various reasons are likely to be, or the circumstances under which they would occur. Based on this input, we provide a conclusion for each bit regarding how to react when a transaction fails and that bit is set.

We emphasize that the best strategy for reacting to transaction failures depends on a number of factors, including the context and application, feasibility of remediating for various conditions (see below), and the cost of the alternative code to be executed if the decision is made to give up trying the transaction. Thus, the following is our attempt to present as clearly and completely as possible the factors to consider, and to give the best advice we can for programmers thinking about how to use the CPS register in deciding how to react to a transaction failure.

We also note that it is not feasible to precisely specify the best course of action for any particular circumstances. For example, different outcomes can result from seemingly identical situations for reasons such as different pseudorandom cache eviction choices, or because of interactions with other code or hardware resources.

For all of these reasons, our advice is somewhat vague in places, for example suggesting “retrying a few times to verify”. In various places, we provide a number just for concreteness, but this should *not* be taken as an opinion that this is the right number.

We begin with the UCTI bit, because if this bit is set, the values of the other bits are less relevant, as explained below.

UCTI This bit is set if a branch instruction is executed during the transaction before a load on which it depends is resolved. Depending on branch prediction, this *may* cause misspeculation, which in turn can cause the transaction to fail for some reason unrelated to the correct execution path. Thus, if UCTI is set, other bits may

⁴This summary is complete to the best of our knowledge, and we have engaged in detail with the Rock architects to achieve as much detail and clarity as we could.

be misleading. For example, there may be an unsupported instruction encountered during misspeculation, in which case the INST bit would be set (see below), which suggests that retrying is not worthwhile. Nonetheless, retrying the transaction is likely to be successful (or at least not fail again for the same reason) because the unresolved load that caused the UCTI bit to be set should (eventually) be resolved.

Note, however, that it is not advisable to blindly retry forever when the UCTI bit is set. We have observed subtle interactions between TLB, cache, and branch prediction state that have resulted in pathologies in which the transaction fails thousands of times with the UCTI bit set. This experience, and how we addressed it, is discussed in Section 7.2.

Conclusion: Retry many times (say 20) but not forever; repeated failures with the UCTI bit set may indicate pathologies which could be investigated and mitigated, perhaps by using speculation barriers, as described in Section 7.3.

Conclusions below for the remaining CPS bits assume that the UCTI bit is not set: if it were, we’d already have made the decision to retry.

EXOG Some events outside transactions do not preserve the contents of the CPS register; such events set the CPS register to EXOG. This can happen due to asynchronous events such as interrupts, as well as synchronous ones such as system calls. Thus, when the CPS register contains EXOG, it does not provide any indication as to why the previous transaction, if any, failed. To maximize the chances of getting a useful indication, we recommend reading the CPS register as soon as possible after a transaction fails, preferably as the first instruction at the fail address specified by a `chkpt` instruction. Note that this does not eliminate the possibility of reading EXOG, as an interrupt may occur before that first instruction is executed. However, it does make it a rare case, making the following conclusion a reasonable one.

Conclusion: Retry.

COH This bit is set when a transactional cache line is lost from the L1 D-cache due to invalidation. It often indicates a conflict with another transaction or other memory access.

COH can be set even though there was no true coherence conflict when a cache line that has been read transactionally is evicted from the L2 cache. Because the L1 cache is included in the L2 cache, such eviction causes an invalidation of the L1 cache line, which is identical to a coherence invalidation due to a conflict from the L1 cache's point of view. Note that this case differs from eviction from the L1 D-cache of a line that has been speculatively read, in which case only the LD bit (and not the COH bit) is set.

Conclusion: Retry. While a small number of immediate retries may be beneficial, repeated retries should be performed only after some back-off.

TCC The TCC bit is set when a trap instruction causes a precise trap.

Conclusion: Don't retry.

INSTR The INSTR bit is set when:

1. A pipe clearing instruction is executed in a transaction. Examples include: `flush`, `cas`, `casx`, `stfsr`, `membar`, and `membar #halt`. Note that the `flush`, `cas`, and `casx` instructions look like load instructions to the pipeline and will therefore set the LD bit as well as the INSTR bit.
2. A `save` instruction causes a spill or clean window trap.
3. A `restore` instruction is executed after a `save` instruction inside a transaction. This cause can be avoided in SST mode by executing a `brnr` instruction immediately after the `restore` instruction (see Section 7.3).

Conclusion: Don't retry.

PREC The description below uses the following abbreviations: UTLB for Unified TLB, uITLB for

micro-ITLB, and uDTLB for micro-DTLB; see [4] for details of Rock's memory architecture.

The PREC bit is set when:

1. An instruction other than a trap instruction reaches the trap stage and causes a precise trap (trap instructions set the TCC bit instead).
2. A load instruction results in a uDTLB or uITLB miss, and no UTLB entry is present, requiring a trap to fill the entry.

Notes:

- To disambiguate uITLB and uDTLB misses for reason 2, only PREC is set for uITLB misses, while the LD bit is set as well for UDTLB misses.
- For both reason 1 and reason 2 when it is a uDTLB miss, just the PREC bit is set, so these cases cannot be distinguished. This is not ideal when trying to diagnose the causes of failures, and also means that we might retry (after remediation, see below) for reason 2 (uDTLB case) when the failure is due to reason 1, in which case the retry is likely to be futile. This illustrates the importance of good feedback about failure reasons in future HTM implementations, but in practice we think this situation is likely to be rare.
- Store instructions that result in uDTLB misses (whether present in the UTLB or not) do *not* set the PREC bit; see details for the ST bit below.

Conclusion: If the LD bit is also set, give up immediately if no remediation for the *data* page accessed is possible, otherwise remediate and retry. If the LD bit is not set, give up immediately if no remediation for the *code* page accessed is possible, otherwise remediate and retry.

Note: "remediation" means performing some action that makes it unlikely that a subsequent transaction attempt will fail again for the same reason as the previous one. We discuss possible

approaches and issues related to remediation in Section A.3.

ASYNC The ASYNC bit is set when:

1. A deferred or disrupting trap occurs.
2. The strand is requested to park or suppress flushes. This occurs, for example, if one of the strands on the chip has not been able to make forward progress for a long time or such a request is initiated by the service processor.
3. An externally initiated reset trap occurs.
4. Certain corner cases are avoided by clearing the pipe; these cases cause the ASYNC bit to be set.

Conclusion: Retry, but if this happens a large number of times, investigate, as it suggests either a pathology related to case 4 above or frequent occurrence of events such as reset traps that should not be so frequent.

SIZ The SIZ bit is set for the following reasons.

1. An overflow of the deferred queue.
2. An overflow of the store queue. This is disambiguated from case 1 by setting the ST bit as well.
3. An attempt to defer five instructions on the same cycle (very rare).
4. An internal bank conflict within the deferred queue (rare).

Conclusion: If the ST bit is set as well (indicating store buffer overflow), retry a few (say three) times to verify, and then give up. Otherwise, retry a moderate number (say six) times.

LD The LD bit is set when:

1. A cache line that has been read transactionally (so its s-bit is set) is evicted from the L1 D-cache.
2. Too many loads lose arbitration to the data cache to fill requests (rare).

3. Data cache data or tag parity error (very rare).
4. An L2 miss is detected in a transaction (Scout mode). In SST mode, an L2 miss never (directly) causes a transaction to fail.
5. Certain forward progress corner cases are avoided by a pipe clear that causes the LD bit to be set (very rare).
6. A load-like pipe-clearing instruction is executed (namely, `flush`, `cas`, or `casx`). In this case, the INSTR bit is set as well.
7. Multi-RAW: A load to the same word as two or more previous stores that the processor was unable to merge will cause the transaction to fail. Note that there is no reason to retry in this case, as the transaction is very likely to fail again. In this case, the ST bit is set as well.
8. A load experiences a UTLB miss. Note that there is no need to retry here without remediation, as it will never succeed. In this case, the PREC bit is set as well.

Conclusion: If it appears to be a UTLB miss related to a data load (that is, the PREC bit is also set), give up immediately if remediation is not possible, otherwise remediate and retry. If it appears to be a multi-RAW access (that is, the ST bit is also set) or a pipe-clearing instruction (that is, the INSTR bit is also set), retry a few times to verify, and then give up. Otherwise, if in SST mode, the failure is probably due to reason 1, in which case retrying a few times may be worthwhile because different decisions made by the pseudo-LRU eviction policy may allow success. In Scout mode, it is again worth retrying in this case, as the failure may be due to reason 4, and the transaction may be successful after the L2 cache miss is resolved. Due to the latency of resolving an L2 cache miss, it makes sense to retry more times for this case in Scout mode than in SST mode.

ST The ST bit is set when:

1. A store experiences a uDTLB miss.

2. The data register (`rd`) for a nontransactional store [4] instruction depends on a load that resulted in a cache miss that has not yet been resolved.
3. A data cache parity error (very rare).
4. One of the source registers used to determine the address for a store instruction (`rs1` and/or `rs2`) depends on a load that resulted in a cache miss that has not yet been resolved.
5. Store-buffer overflow. The `SIZ` bit is set as well in this case; see above.
6. Multi-RAW. The `LD` bit is set as well in this case; see above.

Conclusion: If it appears to be a store-buffer overflow (that is, the `SIZ` bit is set as well), retry a few times to verify, and then give up. If it appears to be a multi-RAW access (that is, the `LD` bit is set as well), retry a few times to verify, and then give up. Otherwise, one of reasons 1–4 applies. Retry a fair number (say six) times, perhaps remediating for reason 1 if possible. Note that transactions fail due to reason 1 regardless of whether an entry is present in the UTLB. However, the failure causes a request for a uDTLB entry to be established. Thus, a subsequent retry may succeed if there is a UTLB mapping present, even if no remediation is performed.

CTI The CTI bit is set when:

1. A branch instruction that has been placed in the deferred queue (see Section 2) is found to be mispredicted.
2. A branch in the delay slot of another branch is executed.
3. A helperized instruction (that is, an instruction that is implemented in microcode; for example, a block load) is executed in an annulled delay slot.

Conclusion: Retry a fairly large number (say 20) times because repeated attempts should re-

duce the number of mispredicted branches because of state retained in cache and branch predictors during failed attempts. (Note that reasons 2 and 3 should never occur in code produced by most modern compilers, so retrying makes sense in these cases even though a transaction that failed for either of these reasons would likely do so forever.)

FP The FP bit is set when:

1. Long latency instructions such as `fdiv` and `fsqrt` fail transactions.
2. A part of a floating point operation’s operand is dependent on an outstanding load miss. This occurs if there is a single-precision load miss followed by a double-precision fp operation where half of the source register is dependent and the other half is not (rare).
3. A RDGSR operation when the `MASK` or `ALIGN` field in the GSR register is dependent on an outstanding load (only appears in specialized code).
4. A RDY operation when the `Y` register is dependent on an outstanding load (deprecated).

Conclusion: Don’t retry.

A.2 Reacting to the CPS register

Figure 9 illustrates how software might react to repeated failures of a hardware transaction. Again, this is schematic and is intended only to illustrate a reasonable approach to reacting to the CPS register. Different choices may be appropriate for different circumstances. In particular, the choice of values for constants such as `nonCoherenceFailureLimit` and `coherenceFailureLimit` depend on the cost of the software alternative relative to the likelihood of success with more retries.

The pseudocode in Figure 9 illustrates CPS triage logic to be executed immediately after a transaction failure. Ideally the `rd %cps, %<dest reg>` instruction that is represented by the `read_cps_register()`

```

cpsVal = read_cps_register()

numFailures++

if ( cpsVal & ( CPS_UCTI | CPS_CTI | CPS_ASYNC | CPS_EXOG ) )
    giveUpIfExceeds( numFailures, nonCoherenceFailureLimit )
    < retry >

if ( ( cpsVal & CPS_PREC ) && ( cpsVal & CPS_LD ) )
    < remediate for data load TLB miss or give up >

if ( cpsVal & ( CPS_FP | CPS_INST | CPS_TCC ) )
    < give up >

if ( cpsVal & CPS_PREC )
    < remediate for instruction fetch TLB miss or give up >

if ( cpsVal & CPS_COH )
    giveUpIfExceeds( numFailures, coherenceFailureLimit )
    < back off >
    < retry >

if ( cpsVal & CPS_ST )
    if ( cpsVal & ( CPS_LD | CPS_SIZ ) )
        < give up >

// possibilities here are: ST, LD, SIZ

if ( cpsVal & ST )
    < remediate for data store TLB miss or give up >

giveUpIfExceeds( numFailures, nonCoherenceFailureLimit )
< retry >

```

Figure 9: Pseudocode illustrating use of CPS register to decide how to react to a transaction failure. `giveUpIfExceeds(n,lim)` is shorthand for `if (n > lim) <give up>`.

function call is the *first* instruction executed at the fail address specified by the `chkpt` instruction that started the transaction.

It is generally not advisable to retry unconditionally, regardless of the reported failure reason. We use the `numFailures` counter to avoid doing so; it is initialized to zero before the first attempt of a transaction (not shown) and records the number of retries.

After incrementing `numFailures`, we first check for conditions in which the transaction should be retried immediately, provided we have not yet exceeded `nonCoherenceFailureLimit` attempts. Specifically, if any of the UCTI, CTI, ASYNC, or EXOG bits is set, we retry immediately.

We then examine the value read from the CPS register further, and take appropriate action. The pseudocode should be mostly self explanatory given the descriptions in Section A.1, together with the following elaborations.

For failure reasons where some form of remediation may be helpful, we simply state “remediate or give up”. This glosses over several issues. First, in some cases remediation is not practical, for example because certain information is not available (see Section A.3 for a detailed discussion). In such cases, there is no alternative but to give up.

Second, even if remediation is possible, there is no guarantee in general that the transaction will eventually succeed, so if a case for which remediation is suggested arises repeatedly, we should eventually give up.

Finally, when the ST bit (and no other bit) is set, this may be due to a uDTLB miss even if there is an entry in the UTLB, in which case subsequent attempts may succeed even without remediation. Therefore, the simple “remediate or give up” response suggested by the pseudocode for this case may be too pessimistic; it may be worth retrying a few times before giving up.

The pseudocode shown in Figure 9 is illustrative and is not intended to be taken too literally. In some cases, we have found it critical to keep memory accesses and conditional branches in the CPS triage code to a minimum in order to avoid pathologies such as the one described in Section 7.2. For this reason, our CPS triage code is factored to allow inlining of

the common cases while other cases are not inlined. Furthermore, to avoid unnecessary memory accesses, we maintain the `numFailures` counter in a register.

Finally, slightly different orders for examining the CPS bits may be appropriate in some cases. For example, it may be preferable for immediate retries in response to seeing only the ST bit set (before remediating or giving up, as discussed above) to come before other cases for which immediate retry is less useful.

A.3 Notes about remediation

In this section we discuss what kind of remediation may be helpful when the CPS register indicates that remediation may be required. In such cases, we have some hints about what kind of remediation may be required. Specifically, we can distinguish between data loads, data stores, and instruction fetches, as indicated in Figure 9. In each case, a UTLB mapping must be established that will prevent the transaction from failing again for the same reason when retried. Thus, in each case, an address is required (the data address for an offending data load or store, or the code address for an offending instruction load).

Below we discuss simple solutions for some specific cases and possible approaches for other cases. Unfortunately, however, because the required address is not immediately available in general, there is no simple and general solution for achieving the required remediation. Therefore, designers of future HTM features should carefully consider not only what information might be needed to diagnose the cause of a transaction failure, but also (if applicable) what information might be needed to facilitate remediation that can avoid aborting the transaction again for the same reason.

A.3.1 Remediating TLB misses for code pages

To avoid repeating a transaction failure due to a TLB miss for code, it is necessary to establish a mapping with “execute” permission in the UTLB for the code page in question. For this purpose, it is not sufficient to simply access a memory location on the page (for example using the dummy CAS technique discussed

earlier) because this does not establish a mapping with execute permission. Instead, it is necessary to actually execute an instruction on the page for which a mapping is desired.

A straightforward approach exists for transactions that fit in a single page and do not call any functions that may reside on other pages. The idea is to use an `align` directive to ensure that the transaction (and any functions it calls, if applicable) fits within a single page. This way, before the `chkpt` instruction is executed, there will be an ITLB mapping for this page, and all code within the transaction resides on this page, so the transaction will not fail due to the absence of an ITLB mapping for another page.

We use this technique for our DCAS implementation, for example (see Section 5). Note that, while the simplest way to achieve this for small functions is to align them at the beginning of a page, this prevents different aligned transactions from residing on the same page, and thus can increase ITLB pressure. Furthermore, such alignment patterns can lead to excessive biasing towards specific I-cache entries.

For transactions that are larger and/or span more than one code page, the situation is more challenging. The main reason is that, when a transaction fails due to a TLB miss when attempting to load code, although we know the reason the transaction failed, we do not know *which* code page it attempted to access. Below we discuss some possible approaches for dealing with this problem, none of which we consider entirely satisfactory (hence our position that future HTM implementations should provide better feedback in some cases, see Section 11.1). First we discuss how to remediate for a code page (assuming that we have the address for which we know or suspect remediation is needed).

Establishing a TLB mapping for a given code page Absent special compiler support, we have no control over what code is on the page, so we are presented with the problem of executing at least one instruction on a given page, without affecting the program's behavior. This is straightforward if there is a known side-effect-free function on the target page: we can simply call the function. But in general this

is not the case. The following trick based on SPARC delay slots allows us to execute a single instruction on the target page. For now, let us assume that there is a `nop` instruction at a known address on the target page. Then calling the `ForceExecutable` function (Figure 10) with the known address achieves the desired result.

The key to understanding why this trick works is understanding the behavior of instructions in the *delay slot* of a branch in SPARC [38]. Briefly, in the above example, the `jmp1` instruction causes control to transfer to the specified address (in register `%o0`) *after* the `ba` instruction that immediately follows it (i.e., is in its delay slot) is executed. Correspondingly, the `ba` instruction causes control to transfer to the `1:` label *after* the first instruction at the target of the `jmp1`, namely the target `nop` instruction. Thus, the net effect is that the `jmp1` instruction causes the target `nop` instruction to be executed, but the `ba` instruction brings control back to the `ForceExecutable` function shown above, ensuring that additional instructions after the `nop` are not executed.

In practice, normally compiled code usually contains many `nop` instructions, so it will often be easy to find one. However, with only slightly more complexity, other instructions can be used too. For example, a store instruction can be used by changing its target address to point to some memory location that can be changed without having any effect on the program's behavior, and restoring the register's previous contents after the store has been executed. Similarly loads and other instructions can be used by saving and later restoring the register(s) affected by the instruction. Given this range of options, it would be very rare to be unable to find any suitable instruction on a code page, and indeed one is likely to be found near the beginning of the page.

A cleaner solution would be for the operating system and/or hypervisor to provide an interface for establishing the required mapping. But, while the above-described mechanism is far from elegant, it is sufficient to establish a TLB mapping with execute permission for a code page, given its address.

```

    !! Signature: void ForceExecutable (void * InstructionAddress)
    .text
    .align 64
    .type ForceExecutable, #function
    .global ForceExecutable
ForceExecutable:
    !! Refer to the SPARC(R) V8 manual and specifically the section that
    !! describes the handling of DCTI couples.
    jmpl    %o0,%g0
    ba     1f
    nop
1:  retl
    nop
    .size ForceExecutable,.-ForceExecutable

```

Figure 10: Assembly routine that can be used to establish a TLB mapping with execute permission for the code page containing a specified address.

Identifying code pages for remediation The remaining challenge is determining *which* code page a transaction attempted to access before failing because of the lack of a TLB mapping for the page. In general, this is a hard problem, because after a Rock transaction fails, we have no indication of which instruction it was executing, the contents of the registers, and so on.

In some cases it is possible to parse the instruction stream to statically determine (a conservative superset of) the set of pages reachable within a transaction. With compiler support, this information could be made available without runtime parsing of the code.

Note however that this is not a general solution, for at least two reasons. First, the transaction may call a function through a function pointer whose contents cannot be statically determined. Furthermore, the set of reachable pages is necessarily conservative in some cases because we cannot statically determine which path through the transaction’s code will be taken, so for completeness we must consider all possible paths. Such conservatism may be expensive. For example, this discussion ignores the issues of geometry, capacity, and replacement policy of the second-level unified TLB. It is possible that a conservative

superset of the reachable pages cannot coexist in the UTLB, while an accurate set would.

Most (but not all) compilers today place each function on a contiguous set of pages, and one can determine the size of the function. In such cases, the set of pages that contain code of the function itself can be determined. But this still leaves the problem of determining the code page that contains a function called within a transaction.

Another possible approach is to use a nontransactional store before a function call to record the address of the function to be called. This way, if the transaction fails indicating a TLB miss on a code page, the address of the most recent function the transaction attempted to call is available, so a TLB mapping can be established as described above before retrying. This approach has several drawbacks. First, it imposes additional overhead on every function call. In general, this may not be excessive. However, because nontransactional stores occupy slots in the store buffer in Rock, we would not want to unconditionally write the address of every function called. Making such stores conditional in order to record addresses of a called function only if it has not been seen previously adds more overhead and more complexity. Furthermore, because each attempt to ex-

ecute a transaction can discover only one call to a function that starts on an unmapped code page, the number of attempts required is at least proportional to the number of calls to functions that begin on code pages that do not have existing TLB mappings.

Finally, we note that, depending on context, simply giving up and using the software alternative may be sufficient to achieve the UTLB warmup needed to allow subsequent transactions to avoid failing due to TLB misses for the same code page. For example, the TLE technique discussed in Section 9 uses the same code path for the critical section as is used inside the transaction that attempts to elide it. Thus, if a TLB mapping for a code page executed in such a code path is not present, we can fall back on the lock, and executing the critical section (nontransactionally because the lock is held) would establish a TLB mapping with execute permission for any code pages executed within the critical section. While this incurs the cost of executing with the lock in the first instance, it is likely to facilitate successful subsequent transactional execution.

Similarly, some implementations of transactional programming languages (e.g., PhTM [23]) can use the same code path for hardware transactional execution as for nontransactional execution in some cases.

To summarize, while some effective techniques for achieving TLB warmup for various cases and contexts do exist, no solution is general, simple, and cheap, so future HTM features should provide better feedback about the reasons for transaction failure to avoid this problem.

A.4 Remediating TLB misses for data pages

The issues are similar when a transaction fails due to the absence of a TLB mapping for data as they are for dealing with the same problem for code (see above). In particular, we can determine the type of access that caused the failure, but we do not get specific information about which instruction caused it, what the register contents were, and so on. However, there are differences, as discussed below.

Establishing a TLB mapping for a given data page First, it is easier to establish a TLB mapping for a given data page than it is for a given code page, because we do not require a mapping with execute permission. Therefore, simply loading from a page is sufficient to establish a mapping that will avoid repeating a failure due to a load-related TLB miss.

For a store-related TLB miss, loading from the page is not sufficient, because we need to establish a mapping with write permission. Our favorite way to deal with this problem is to use a “dummy CAS”, that is, a CAS instruction that provides the same expected and new values. This way, regardless of whether the CAS succeeds or fails, memory is not modified and so program behavior is not affected. We note that this trick exploits knowledge of the implementation of CAS instruction, specifically that it requires write permission for the page it is accessing, even in the degenerate case in which it will never modify the page. Again, a cleaner solution would be for the operating system and/or hypervisor to provide an interface to achieve this purpose.

In practice, we often handle remediation for load-related and store-related misses in the same way, namely by using the dummy CAS technique, because it is likely that the transaction will subsequently modify the page from which it has loaded, so by requesting write permission even for load-related failures, we avoid a second failure for this page if it is subsequently written. We note that this approach could result in the CAS instruction faulting in case a read-only page is accessed. This can be avoided by using the `ASI_PRIMARY_NOFAULT` address space for the CAS instruction.

Identifying data pages for remediation As for code pages, we have the challenge of determining which data page(s) may require remediation after a transaction fails due to the absence of a TLB mapping for a data page. A similar range of options can be considered as discussed above for code, such as compiler assistance, and using nontransactional stores to record addresses of data accesses. Again, in some circumstances, using the nontransactional software alternative may establish TLB mappings that

prevent subsequent transactions from failing for the same reason.

These techniques are subject to similar disadvantages as discussed above for code pages, so again we conclude that there is no simple and cheap solution that applies for all situations, and therefore future HTM features should provide better information to assist software in performing appropriate remediation, if applicable.

About the Authors

Dave Dice currently works at Sun Microsystems on threading and synchronization in the Java Virtual Machine. Before joining Sun Microsystems, he was a co-founder of Praxsys Technologies. You can learn more about Dave Dice on his blog, <http://blogs.sun.com/dave> or on twitter @daviddice.

Yossi Lev received a B.Sc. degree in Computer Science and Math from Tel Aviv University, Israel in 1999. After a few years in the Israeli High-tech industry, he returned to academia and received an M.Sc. degree also in Computer Science from Tel Aviv University in 2004. He is currently a PhD. student at Brown University, and an intern in the Scalable Synchronization Research Group in Sun Microsystems Laboratories. His current research interests include design and implementation of scalable concurrent data-structures and infrastructures, and various aspects of the transactional memory infrastructure in concurrent software design.

Mark Moir is a Distinguished Engineer and is the Principal Investigator of the Scalable Synchronization Research Group at Sun Labs. Moir received the B.Sc. (Hons.) degree in Computer Science from Victoria University of Wellington, New Zealand in 1988, and the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill, USA in 1996. From August 1996 until June 2000, he was an assistant professor in the Department of Computer Science at the University of Pittsburgh, after which he joined Sun Labs.

Dan Nussbaum is a member of the Scalable Synchronization Research Group at Sun Labs. His current work is focused on software and hardware-software hybrid implementations of Transactional memory. Dan has also worked on simulation software, to aid the group's TM research and other architectural work. Dan was previously a Founder at Curl Corporation, where he was the lead architect for Curl's runtime libraries and thread/process systems. Before that, Dan worked in the financial community, at the online division of D.E. Shaw and Co. In 1993, Dan received a PhD in Computer Science from MIT, where he did research in multiprocessor architecture and multiprocessor-oriented graphics algorithms.

Marek Olszewski received both his B.A.Sc. (with honours) and M.A.Sc. degrees in Computer Engineering from the Edward S. Rogers Sr. Department of Electrical and Computer Engineering at the University of Toronto, Canada in 2005 and 2007. Subsequently, he joined the CSAIL laboratory at the Massachusetts Institute of Technology, USA, where he is currently a 3rd year Ph.D. Candidate working under the supervision of Saman Amarasinghe in the Commit Group. During the summers of 2008 and 2009, Marek interned at the Scalable Synchronization Research Group in Sun Microsystems Laboratories.



Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025



Early Experience with a Commercial Hardware Transactional Memory Implementation

D. Dice, Y. Lev, M. Moir, D. Nussbaum, M. Olszewski

SMLI TR-2009-180