Applications of the Adaptive Transactional Memory Test Platform *

Dave Dice* Maurice Herlihy† Doug Lea‡ Yossi Lev*† Victor Luchangco*
Wayne Mesard* Mark Moir* Kevin Moore* Dan Nussbaum*

*Sun Microsystems †Brown University ‡SUNY Oswego

dave.dice@sun.com mph@cs.brown.edu dl@cs.oswego.edu yosef.lev@sun.com victor.luchangco@sun.com
wayne.mesard@sun.com mark.moir@sun.com kevin.moore@sun.com daniel.nussbaum@sun.com

Abstract

Sun has recently announced it will support a form of "best effort" hardware transactional memory in its forthcoming multicore processor, code-named "Rock". In this paper, we report on early results and experience from an exploration of software mechanisms that exploit this feature in a variety of contexts including: explicit transactional programming in the C++ and Java™ programming languages, explicit lock elision in C++ and implicit lock elision in Java, and use in Java concurrency libraries. This work has been conducted using our *Adaptive Transactional Memory Test Platform*, which we plan to open source soon to allow other researchers to explore the use of Rock's HTM support.

1. Introduction

Sun recently announced [27, 28] that its forthcoming multicore processor—code-named *Rock*—will support a form of *best-effort* hardware transactional memory (HTM). Best-effort HTM does not guarantee to support transactions of any size and duration, and thus is free to simply abort transactions that exceed on-chip resources for HTM or encounter difficult events or situations.

Rock's HTM feature is an important but modest first step in integrating HTM support into a mainstream commercial multicore processor. We are therefore interested in understanding what mechanisms and workloads can effectively exploit it, as well as how its limitations impact its usefulness in other situations, whether we can effectively work around them, and what future modifications could make HTM support more effective for more situations.

We have begun to explore a variety of software techniques for exploiting Rock's HTM feature. In this paper, we report on our early progress in this exploration. A key enabler for this work is a simulator developed recently by our group called the *Adaptive Transactional Memory Test Platform* (ATMTP) [18]. ATMTP supports the same HTM interface as Rock and approximates the circumstances under which transactions will succeed and fail in Rock.

Although ATMTP is *not* intended to accurately model Rock's implementation or performance, it provides a useful platform for developing and testing code that will behave similarly on Rock. ATMTP will soon be made available so other researchers and developers can experiment with code that will exploit Rock's HTM feature. We hope the results and experience presented in this paper will provide a valuable starting point for others.

In this paper, we describe our initial work in this direction for the following contexts:

- Implementing a red-black tree using our Hybrid Transactional Memory compiler and library [2].
- Using HTM to simplify the implementation of and speed up the obstruction-free "factory" for DSTM2 [9].
- Implementing nonblocking list and skip list algorithms for the Java™ concurrency libraries using a simple NCAS (N-location compare-and-swap) interface, which is implemented using HTM.
- Explicit use of HTM to elide mutual exclusion locks used to implement a thread-safe vector data structure using a non-thread-safe vector from the C++ Standard Template Library (STL).
- Explicit use of HTM to elide mutual exclusion locks in the libc library.
- Modifying a Java Virtual Machine (JVM) to automatically elide synchronization from synchronized blocks transparently to the programmer.

We have achieved encouraging results, but we have also identified a number of issues that prevent us from achieving results as good as we initially hoped. In some cases we have already worked around the issues; in other cases we have some ideas but further investigation is needed. All of our explorations are in preliminary stages, and for some we simply report current status without any particular result or conclusion.

Our work has also served to identify a few simulator bugs, and has motivated improvements to ATMTP to facilitate the kind of work we are doing. Some of these improvements have already been made and have assisted in achieving the results described in this paper; others are future work. The simulator will soon be open sourced, (see [25] and [29] for details and updates) so others will also have the opportunity to experiment with improving the functionality of the simulator, in addition to using it for their own research.

The rest of this paper is organized as follows: Section 2 presents brief background on Rock and ATMTP. In Section 3 we describe our work to date using ATMTP to explore a number of potential uses for Rock's HTM support, and report on status of our explorations, preliminary results achieved, issues identified, and interesting observations made during our preliminary work. We conclude in Section 4.

2. The Adaptive Transactional Memory Test Platform

In this section, we briefly describe salient features and limitations of the HTM feature supported by ATMTP. A more detailed descrip-

tion, as well as a discussion of Rock and the relationship between Rock and ATMTP can be found in a companion paper [18]. Although ATMTP aims to provide a useful development platform for code that will behave similarly on Rock, we emphasize that it is *not* intended as a completely accurate model of Rock; indeed, as we do not yet have final silicon for Rock, it cannot be. In the remainder of this paper, we restrict our attention to the HTM feature as modeled by ATMTP.

2.1 Instructions and interface

ATMTP supports a standard SPARC® instruction set architecture, plus two new instructions `chkpt` and `commit` and a new *checkpoint status* (`cps`) register. A transaction is started using the `chkpt` instruction, so called because it *checkpoints* the register file so that it can be restored if the transaction fails. The `chkpt` instruction specifies a PC-relative *fail address*, where control is transferred if the transaction fails for any reason.

If the transaction executes a `commit` instruction, the code executed within the transaction is executed atomically and in isolation, and control continues past the `commit` instruction.

If the transaction fails, it appears as if the `chkpt` instruction were an unconditional branch to the fail address, and none of the code within the transaction were executed, with one exception: the contents of the `cps` register may change to indicate the reason for the transaction failure.

2.2 Reasons for transaction failure

Transactions in ATMTP can fail for a number of reasons. Here we briefly describe those most relevant to this paper; a more detailed and complete description appears in [18].

Conflict: A location read by the transaction is modified by another thread, or a location written by the transaction is accessed (read or modified) by another thread.

Cache geometry: The cache lines read by a transaction fail to fit in cache because too many of them map to the same cache set.

Resource exhaustion: The transaction exceeds some resource limit, for example, on the number of stores that can be performed in a transaction.

Exception: An exception occurrs during the transaction's execution. A relevant example for this paper is an exception caused by a TLB miss.

"Difficult" instruction: The transaction executes an instruction that ATMTP does not allow during transactional execution. A example relevant to this paper is the `sdivx` instruction.

Function call: The transaction fails due to a function call. (Rock is expected to have limitations that make transactions that call functions likely to fail; ATMTP approximates these limitations. Inlining can provide some relief from this cause of failure.)

2.3 System Model

ATMTP simulates a single-chip CMP system with a private L1 cache per core and a single shared L2 cache. In this paper, we model single-chip CMP systems with between 2 and 32 processors, in which processor cores and caches are connected by a low-latency fully-connected on-chip network.

3. Using ATMTP to explore HTM-based mechanisms

In this section, we describe several techniques for using Rock's best-effort HTM, and describe our progress in implementing and testing these techniques using ATMTP. Specifically, we repeat earlier experiments on hybrid transactional memory run on a previous simulator of "generic" HTM support [2, 13], this time running them using ATMTP to study differences introduced by Rock's specific features and limitations (Section 3.1). We also use HTM to simplify and improve an implementation of a DSTM2 "factory" [9] (Section 3.2), and to develop new implementations for linked lists and skip lists for the Java concurrency libraries (Section 3.3). Finally, we attempt to improve various lock-based implementations by using HTM to *elide locks* [20] (Sections 3.4, 3.5 and 3.6).

3.1 Hybrid transactional memory using ATMTP

The appeal of transactional memory lies in its potential to enable scalable implementations of complex algorithms and data structures without the complexity of fine-grained locking or ad hoc application of nonblocking techniques. This potential of transactional memory has been demonstrated, for example, in the implementation of concurrent *red-black trees*, discussed below, using transactional memory [10]. HTM implementations are particularly attractive, because they have significant performance advantages over software implementations. However, there is an additional challenge when using best-effort HTM such as provided by Rock and ATMTP, because programmers cannot always accurately predict which transactions can succeed. *Hybrid transactional memory* [2] is an effective way to exploit best-effort HTM to provide the simplicity and flexibility of software transactional memory (STM) with the efficiency (in most cases) of HTM. Hybrid transactional memory can execute transactions using HTM (if available), but can also use software transactional memory (STM) for transactions that do not commit in hardware.

The chief difficulty with hybrid transactional memory is making sure that hardware and software transactions "play nicely together". In particular, we must be able to detect conflicts between hardware and software transactions, so that the conflicts can be resolved properly. Thus, hardware transactions must access metadata maintained by the STM. In HyTM, our initial hybrid transactional memory prototype [2], transactions executed using HTM augmented every transactional load and store with code to detect conflicts with concurrent software transactions. Simulations showed that this approach imposed significant overhead compared with hardware transactions that were not so augmented (e.g., because the underlying hardware supported "unbounded" HTM) even when all transactions completed in hardware. Furthermore, the requirement for the STM to expose sufficient information for hardware transactions to detect conflict sharply constrained the design space of the STM component of HyTM.

We therefore proposed *phased transactional memory* (PhTM) [13], a form of hybrid transactional memory that supports different *modes of operation* and allows seamless transition between those modes. In particular, we can have a mode that allows only hardware transactions, and another that allows only software transactions. Then hardware transactions need only check that PhTM is in hardware-only mode, avoiding any per-access overhead. And when in software-only mode, PhTM is free to use the best STM available.

In previous work [2, 13], we have shown that our HyTM and PhTM systems can effectively exploit best-effort HTM support to achieve substantially better performance than existing software methods of similar programming complexity. However, this previous work was based on "generic" best-effort HTM support, not tailored for Rock's specific features and limitations. We have therefore repeated these experiments using ATMTP, with the goal of evaluating our ability to produce similar advantages using Rock. In particular, we implemented a concurrent red-black tree using a transactional style of programming, and ran experiments using different transactional memory implementations.

A red-black tree [1] is a kind of balanced search tree that supports insert, delete, and lookup operations. Insert and delete oper-

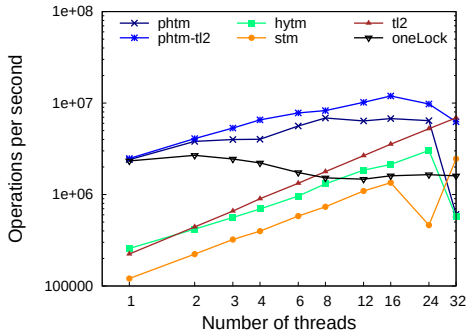

Figure 1. Performance experiments on red-black trees

ations traverse the tree from the root, searching for the appropriate place in the tree to make an update. While many updates result in only local changes, some require "rotations" to preserve the red-black tree invariants, and occasionally these rotations propagate all the way to the root.

Although sequential red-black trees are well understood, directly implementing a red-black tree that exploits concurrency is challenging [4]. For scalability, it is important that operations traversing down the tree not require exclusive ownership of the nodes they read. On the other hand, rotations that propagate up the tree generally do require exclusive ownership, so ownership access of the affected nodes must be "upgraded" from shared to exclusive. This creates the potential for deadlock; avoiding this deadlock presents a significant challenge. Therefore, red-black trees provide an interesting test case for transactional memory: we want to achieve an efficient concurrent red-black tree with programming complexity similar to that required for a sequential one.

The first issue we encountered is that the restriction on function calls within transactions (see Section 2.2) prevents us from successfully using a recursive red-black tree implementation, which cannot avoid nested function calls. It was not too hard to switch to an iterative version we had used in another context, and with a little inlining work we were able to circumvent this restriction.

Experimental observations

Following [13], in our red-black tree experiments, we prepopulate the tree with 2000 keys from 0..4095 (i.e., keys have 12 bits, and the tree is about half full), and then measure the (simulated) time taken for each thread to execute 1000 operations chosen at random according to the following distribution: lookup: 60%, insert 20%, delete 20%; each operation chooses a key value uniformly at random from 0..4095. We ran this experiment using a simple coarse-grained lock-based algorithm (`oneLock`) and using transactions with five different transactional memory implementations: HyTM running in software-only mode (`stm`); TL2 [3] (`tl2`); HyTM [2] (`hytm`); PhTM (`phtm`); and PhTM with TL2 as its STM component (`phtm-tl2`). The results for these experiments are shown in Figure 1.

At the highest level, most of the results we achieve for red-black tree using PhTM are qualitatively similar on ATMTP as they were on our previous simulator that provides generic best-effort HTM (see [13]): Both PhTM implementations are near or at the top and mostly flat throughout their range, except for the precipitous drop in `phtm` going from 24 to 32 threads. The pure STM implementations start off the worst by a considerable margin and then steadily improve as the number of threads increases, with `tl2` beating `stm` by a factor of about 2 throughout the range. The single-lock-based algorithm starts off strong, but declines as the number of threads increases due to the increased contention. Under low contention, all but a handful of operations are successfully executed as hardware transactions. However, as discussed below, `hytm` suffers a significant *decrease* in performance when using ATMTP compared with the previous simulator.

Beyond these observations, it is difficult to compare the results here with those presented in [13] because many factors have changed since [13] was published. The most significant factor was changing our simulated machine architecture from the single-core-per-chip non-uniform memory system used in [13] to a single-chip CMP system used by ATMTP. This change was enabled by our switch from version 1.4 to version 2.0 of the GEMS toolkit from the University of Wisconsin [16]. GEMS 1.4, on which our previous simulator was based, does not support simulating HTM on a CMP. ATMTP, which is based on GEMS 2.0, allows us to model a CMP architecture that is much closer to that of Rock. In particular, interprocessor communication latency is much lower in the CMP system than in the previous model, which explains the much slower descent of the `oneLock` line in this graph compared to the equivalent one in [13]. In addition, the HyTM library that implements the STM (of both HyTM and PhTM) has undergone further development since the publication of [13], and of course, as mentioned earlier, the red-black tree implementation run in these experiments is iterative rather than recursive.

While these results show no particular improvement in performance under heavy contention over our previous results, we have made some initial progress towards more realistic and robust contention management. Specifically, because the simulator used for the previous results provided only generic best-effort HTM, we had no feedback about the reasons for transaction failures. This forced us to use an inflexible policy for switching modes. We did not spend a lot of effort tuning this policy, but we did tailor it to the benchmark at least somewhat, and the same choice was not optimal for other benchmarks.

Because Rock and ATMTP provide feedback (via the `cps` register) about reasons for transaction failure, we can now implement more flexible and adaptive policies that use this feedback to make better decisions. We have implemented a slightly more flexible policy that uses this mechanism to adapt to the workload conditions, rather than specializing for the workload as we did in our previous work. Briefly, we use feedback from the `cps` register to estimate how likely retrying is to be successful and decide how often to retry on that basis. For example, if a transaction executes an instruction that is not permitted in hardware transactions, it will likely do so again if retried, so we do not retry at all in this case. If the transaction fails due to conflict, our experience suggests that it is worth retrying a significant number of times before giving up and executing the transaction in software. Our simple mechanism retries between 0 and 9 times depending on these factors, backing off exponentially with each retry after retrying immediately the first three times.

Based on our experience to date, our simple retry policy appears to be generally as successful as the hand-tuned one we used previously. We have not yet attempted further tuning of this policy, nor have we experimented to see how well it adapts to other workloads. No doubt we will find cases in which it does not do so well, and we hope we will be able to make good use of the feedback provided via the `cps` register to implement an adaptive policy that is effective across a range of benchmarks.

Red-black tree results using HyTM on ATMTP

We now return to the experimental results for HyTM on ATMTP, which were much less encouraging than the results for PhTM described above (see Figure 1). In particular, we found that even in a single-threaded run, more than 30% of the operations failed to complete as hardware transactions. The result was significantly worse

overall performance than we achieved in our previous experiments based on generic best-effort HTM [13].

Simulator statistics indicate that for the single-threaded run, all of these failures were caused by TLB miss exceptions: If the TLB (translation lookaside buffer) does not contain an entry to assist in translating the virtual address for a given memory access to a physical address, an exception is generated to cause the appropriate entry to be loaded into the TLB; these exceptions cause transactions to fail. Because the transaction fails, the exception is not processed so retries will likely fail again for the same reason. TLB misses can occur both in the data TLB (DTLB) for data memory acesses and in the instruction TLB (ITLB) for instruction fetches.

DTLB misses within transactions can be avoided using software prefetching techniques, but because code placement is less transparent to application code, a general solution for avoiding ITLB miss exceptions may require compiler support. This example clearly demonstrates the value of ATMTP: it enables us to identify such issues and provides an environment for testing workaround solutions before Rock becomes generally available.

3.2 Simplifying and optimizing DSTM2's obstruction-free factory

DSTM2 [9] provides a transactional programming interface for the Java programming language, enabling researchers to plug in and compare different software transactional memory (STM) implementations using the same transactional application code. To do so, one provides a *factory* for producing transactional objects whose methods implement the STM. DSTM2 comes with two example factories, the *obstruction-free factory* and the *shadow factory*. The obstruction-free factory implements the STM using the obstruction-free DSTM algorithm [10]. This simple implementation requires a multiword metadata object called a *locator* to be replaced atomically. This is achieved by using a level of indirection, which imposes significant overhead on every access to a transactional object.

Several clever improvements to this algorithm have been devised to eliminate this level of indirection, at least in some cases [4, 14, 15, 26]. We are exploring the possibility of eliminating this level of indirection entirely without significantly complicating the code by using HTM to effect the atomic replacement of locators. Because DSTM2 is implemented in the Java programming language, this requires us to expose an interface to the HTM feature to Java code. To do so, we extended the `java.unsafe` facility in an experimental version of the Java HotSpot™ Virtual Machine. Specifically, we provide two methods, `CHKPT` and `COMMIT`, that enable programmers to execute HTM transactions. The `CHKPT` method executes the `chkpt` instruction, which starts transactional execution, and returns 0. The fail address specified in the `chkpt` instruction is within the `CHKPT` method, and the code there reads the `cps` register and returns its contents. Thus, if the transaction fails, the return value of `CHKPT` is the contents of the `cps` register, which indicates the reason for the failure (this value is always nonzero in this case [18]). The `COMMIT` method simply executes the `commit` instruction. We also provide an `ABORT` method that causes the transaction to fail. These methods can be used in the style shown in Figure 2.

We believe this facility has the potential to support a variety of powerful mechanisms. However, there are significant challenges at several levels to successfully executing a hardware transaction in an experimental STM implemented in Java running in a modified JVM on a simulator. For example, sophisticated managed run-time environments such as the HotSpot JVM are a challenge to explicit `CHKPT` and `COMMIT` usage: the JIT compiler may emit within a transaction body transaction-unfriendly safepoint polling points, deferred symbol resolution, run-time invariant guards, or

```
final int cps = Unsafe.CHKPT() ;
if (cps == 0) {
  ... txn body ...
  Unsafe.COMMIT() or Unsafe.ABORT()
  ...
} else {
  ... failure path ...
  ... decode cps for failure reason ...
  ... decide what to do ...
}
```

Figure 2. Using `CHKPT` and `COMMIT` methods.

so-called "uncommon traps", etc. If executed in a transaction, these will cause failure and roll-back, resulting in a nonzero return from CHKPT(). Unfortunately, that also means the run-time environment was unable to intercept and resolve the condition that needed attention, so retrying the transaction will likely result in persistent failure and lack of progress.

We therefore started with a simple test: we modified the obstruction-free factory of DSTM2 so that it uses small transactions to access the locators atomically, thereby eliminating the need for a level of indirection to achieve this effect.

One challenge that arises in testing this mechanism on the simulator is that our experimental JVM supports the use of hardware transactions only in compiled code, not in the interpreter. By default, all code is interpreted initially, and the JIT compiler dynamically compiles code that is executed frequently. However, even when the code to start a transaction is warmed up sufficiently to be compiled, the body of the transaction may not be. The result is that a trap to the intepreter is executed inside the transaction; this trap causes the transaction to fail. Therefore the interpreter does not execute the code and, more importantly, the JIT statistics are not updated to reflect the attempt to execute the transaction, thus creating a chicken-and-egg situation in which the transaction code never gets compiled and therefore never succeeds.

We work around this problem by specifying `-Xbatch` and `-Xcomp` on the command line, which results in the immediate and complete compilation of all executed methods. However, start-up performance suffers considerably in this mode. In addition, it takes the JVM considerable time to reach compilation steady-state and achieve the same degree of inlining that would occur in the default interpret-then-compile mode, exacerbating the difficulty of running long enough at simulation speed to achieve meaningful results. With all of these issues, it is tricky to orchestrate simulations at reasonable speed with all necessary start-up code executed and functionality enabled when we begin measuring.

Another issue we encountered was that code generated for type casting values returned from `Unsafe.getObject` inside a transaction may access type-specific metadata residing on pages not mapped in the TLB, and the resulting TLB miss causes the transaction to fail. We avoided this issue by factoring the type cast outide the transaction.

This work is in its initial stages. Although we have gotten some hardware transactions to commit in the modified obstruction-free factory when running single-threaded, we have not yet managed to do so reliably with multiple threads. We continue to study this problem.

3.3 NCAS-based concurrency libraries in Java

The example uses of best-effort HTM described in the previous two sections involve experiments with transactional programs. There are still numerous issues to resolve before such programming styles are widely adopted. In the meantime, best-effort HTM can be used to improve performance and scalability of more traditional pro-

grams, including unmodified existing programs in some cases. The remaining examples discussed below illustrate some possibilities.

In this section, we explore how HTM may be used in the Java concurrency libraries, so that users of such libraries can benefit from HTM even if they know nothing about it. Potential benefits include improving the performance and scalability of existing functionality and/or achieving simpler solutions, as well as providing new functionality that has been too expensive or too complicated to provide. To avoid subtle language integration issues in making TM functionality available in Java, we used HTM to implement a simple NCAS (N-location compare-and-swap) interface, which encapsulates HTM's ability to atomically read and modify multiple memory locations. For this work, we have implemented only DCAS (double-compare-and-swap).

Implementing DCAS

We implemented DCAS in a few different ways. We first used the `unsafe.CHKPT` and `unsafe.COMMIT` methods described above to write a short transaction. This approach yielded an interesting problem when the DCAS operands were references: In addition to stores to the operands affected by the DCAS, the JIT compiler may also emit code that accesses hidden JVM structures such as garbage collector metadata (i.e., the *card table*), array bounds limits, safepoint checks, etc. Conflicts on this metadata caused transactions to fail frequently enough to severely degrade performance. To work around this problem, we hand coded in assembly a DCAS implementation that put card marking (i.e., setting bits in the above-mentioned card table) outside of the transaction implementing the DCAS. Most of our experiments have been run using this implementation.

Very recently, we modified the JVM based on the observation that card marking is very often redundant: a bit being set in the card table is already set, so the store does not change its value. Thus, if the JVM checks the bit before setting it, and elides the store if it is already set, then many stores are eliminated. Although this introduces some overhead to check whether the card marking is redundant, it greatly reduces the potential for false conflict. With this modified JVM, we were able to use our original DCAS implementation described above.

For this preliminary work, we have not yet implemented a software alternative for the HTM transactions for these DCAS implementations. A complete solution would require either guarantees from the best-effort HTM that, if repeatedly retried, these transactions will eventually commit, or else a software mechanism to use in the hopefully rare case in which a transaction fails repeatedly and deterministically.

Improving library implementations using DCAS

As a first step in using HTM (in the guise of NCAS) to improve the Java libraries, we are experimenting with implementing nonblocking ordered-list-based sets, providing (as per the `java.util.Set` interface) `add` and `remove` methods for adding and removing elements to the set, a `contains` method for checking whether an element is in the set, and an `iterator` method that returns an iterator object with `next` and `hasNext` methods. Each call to the `next` method of an iterator returns an element of the set, with successive calls returning greater elements than previous ones, until all elements in the set have been returned, at which point we say the iterator is done and subsequent calls to `next` throw an exception. The `hasNext` method returns whether there are any more elements to return (i.e., the iterator is not done). Any element that is in the set when the iterator is created and is not removed before it is done must be returned by some call to `next`; that is, although the iterator itself is not thread-safe, its behavior is not arbitrary in the presence of concurrent modifications to the list.

To our knowledge, no previous nonblocking list-based implementations provides the iterator functionality. Even with coarse-grained locking, it is tricky to provide an iterator that neither copies the entire list ahead of time nor prevents the list from being modified until the iterator is done.

A nonblocking CAS-based algorithm for a set without the iterator was developed by Harris and simplified by Michael [6, 17], but this algorithm requires the ability to "mark" a reference, so that the reference and its mark can be modified atomically. To remove the element in a node of the list, the node's next reference is first marked, which abstractly removes the element, and then the marked node (i.e., the node with a marked reference) is spliced out of the list by redirecting the next reference of the predecessor node. They implemented this algorithm in C/C++, in which a markable reference can be implemented by "stealing" a low-order bit from a pointer (provided that pointers to nodes are word-aligned so that such bits would otherwise always be zero). However, this trick does not work in Java; instead, the Java concurrency libraries provide the `AtomicMarkableReference` class (`AMR`). However, using this class introduces an extra level of indirection and other data manipulation to extract the mark and reference, which renders implementations based on it expensive.

Lea developed an alternative way to encode markable references in the implementation of `ConcurrrentSkipListMap` [12]: A reference is marked by changing it into a reference to a newly allocated "deletion marker" node whose next field contains the original unmarked reference. The marked node and the subsequent deletion marker are then spliced out of the list together. Compared to the `AMR`-based algorithm, this algorithm introduces an extra level of indirection only for marked nodes (soon to be removed from the list), and eliminates the extra computation in extracting the mark and reference. However, it is much trickier, and thus offers more opportunity for subtle bugs in its implementation.

Heller et al. developed the "lazy list" algorithm [7], which simplifies the `AMR`-based algorithm and avoids the cost introduced by `AMR` by separating the mark from the reference and guaranteeing the atomicity of access using locks. This algorithm is highly efficient because locks are taken only for nodes being modified; readers do not take locks, and the `contains` method is wait-free. However, it is blocking.

We designed two nonblocking[1] algorithms that exploit HTM to eliminate the locks used by the lazy list, and that also provide the iterator functionality. The first algorithm atomically accesses only two locations, the mark and the reference to the next node in each node. However, like all previous nonblocking algorithms, threads must sometimes "clean up" nodes that have been deleted by other threads. The second algorithm is even simpler, avoiding the need for "clean up" by atomically accessing four locations (the marks and references in two nodes) when removing a node. Thus, the first algorithm uses DCAS (double-compare-and-swap), the second 4CAS. So far, we have implemented only the DCAS-based algorithm.

Any of the nonblocking sorted lists described above can be used as the basis for a nonblocking skip list [11]: Layers of these sorted lists referring to keys randomly chosen along intervals of 2^k for $k < \lg(n)$ serve as indices, where n is the size of the set of keys. While this adds per-node time and space overhead, it not only reduces the average uncontended path length to find a key from $O(n)$ to $O(\log n)$, but also similarly reduces retry costs upon interference. In practice, skip lists of this form appear to be the most efficient and scalable sorted data structures suitable for everyday

[1] These algorithms are lock-free if the NCAS operation is lock-free. However, our HTM-based implementation of NCAS is only obstruction-free [8].

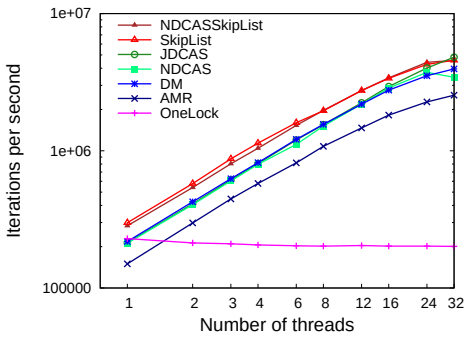

Figure 3. Performance experiments: DCAS lists and skip lists

use. We implemented a skip list of this form based on the DCAS version of sorted lists.

Using the simulator, we compared our DCAS-based implementation, using both the `java.unsafe` facility (JDCAS) and the hand-coded native DCAS (NDCAS), with a coarse-grained lock-based implementation (one lock for the whole list), and versions of the algorithm that either use the `AtomicMarkableReference` class to access the mark and reference atomically (AMR), or use the "deletion marker" nodes instead of markable references (DM). We also compared them with two skip list implementations: the standard one found in `java.util.concurrent` (SkipList), which uses deletion markers, and one that uses our hand-coded native DCAS (we have not yet run these experiments with the `java.unsafe`-based DCAS).

Figure 3 shows the results of running varying numbers of threads, each repeatedly executing three `contains` methods and one each of `add` and `remove` methods, all with randomly generated integers from 1 to 64. As expected, the lock-based algorithm performs well in the single-threaded case, but deteriorates as more threads access the set concurrently. In contrast, the throughput of all the nonblocking algorithms increases almost linearly as the number of threads increases, with most lines leveling out, or even dipping slightly at 32 threads. Also as expected, throughout the entire range, the absolute throughput of the `AMR`-based algorithm is about 30% lower than that of the other three list algorithms, and the two skip list algorithms' throughput are about 30% higher (reflecting $O(n)$ versus $O(\log n)$ path lengths, for the fixed value of $n = 64$ used in this experiment).

One surprising result is that the JDCAS line continues to rise from 24 to 32 threads, beating out even the skip list algorithms at 32 threads. We expected it to perform slightly worse than the algorithm that used the hand-coded native DCAS, and we are investigating this matter. We have some plausible explanations for this result, but we have not yet tested them.

3.4 Lock elision in STL

In this section, we describe an experiment that uses best-effort HTM to implement scalable thread-safe data structures using unmodified non-thread-safe versions from the C++ Standard Template Library (STL) [24]. We can easily achieve a thread-safe implementation using coarse-grained locking (i.e., with a single lock to protect the entire data structure). Although correct, the resulting implementation is not scalable: it does not exploit concurrency between concurrent operations, and requires every operation to modify the lock, even if the operation does not modify any data.

To make lock-based programs more scalable, Rajwar and Goodman proposed *speculative lock elision* (SLE) [20, 21], in which a processor exploits speculation and cache coherence hardware to speculatively execute a critical section without acquiring the lock,

```
// The lock-elision macros, to be used around
// an atomic block.  Parameters:
//
//   ACQUIRE_ST: A *statement* -- acquire lock.
//
//   LOCK_MINE_OR_FREE_EXP: A boolean *expression* --
//   evaluate to true if holding the lock or if lock
//   is free.
//
//   RELEASE_ST: A *statement* -- release lock.

#define TXLOCK_REGION_BEGIN(ACQUIRE_ST,                \
                            LOCK_MINE_OR_FREE_EXP)     \
  {                                                    \
    UINT64 __HTfailures = 0;                           \
    bool __IhaveLock = false;                          \
    while (!beginHT()) {                               \
      __HTfailures++;                                  \
      if (__HTfailures >= MaxHTFailures) {             \
        __IhaveLock = true;                            \
        ACQUIRE_ST;                                    \
        break;                                         \
      }                                                \
      while (!(LOCK_MINE_OR_FREE_EXP)) ;               \
    }                                                  \
    if (!(LOCK_MINE_OR_FREE_EXP)) abortHT();

#define TXLOCK_REGION_END(RELEASE_ST)                  \
    if (!__IhaveLock) {                                \
      commitHT();                                      \
    } else {                                           \
      RELEASE_ST;                                      \
    }                                                  \
  }
```

Figure 4. Macros for using HTM to elide lock acquisitions.

provided that it is not held by some other processor. Conflicting memory accesses (including acquiring the lock) trigger misspeculation, causing the processor to roll back to the beginning of its critical section, at which point it may either retry speculatively or else acquire the lock as normal. However, because SLE is implemented in hardware, which cannot take into account application-level information, it may be applied in contexts where it is not beneficial, possibly even degrading performance in some cases.

Best-effort HTM enables efficient lock elision in *software*, using hardware transactions to execute critical sections speculatively. Performing lock elision explicitly in software is much more flexible [5, 19, 22]: we can ensure the technique is used only in cases where it is likely to be beneficial, and we can use different policies and heuristics for backoff and retry in different situations, etc. In this and the next two sections, we explore the use of this technique in several contexts, with varying degrees of involvement by programmers at various levels.

To facilitate introducing a lock and attempting to elide it using hardware transactions, we developed some simple macros for acquiring and releasing a lock. The macro that acquires the lock takes two parameters, a *statement* that acquires the lock and an *expression* that returns true if and only if the lock is free or the thread evaluating it holds the lock (i.e., if the lock is not held by some other thread). The macro that releases the lock takes a single parameter, which is a statement that releases the lock.

These macros can be defined to produce code to simply acquire and release a lock (i.e., emit the texttttACQUIRE_ST and `RELEASE_ST` statements) or they can be defined to produce code that attempts to elide the lock using HTM. Our versions of the macros that perform the latter are shown in Figure 4. These macros

use the `beginHT` and `commitHT` functions to begin and commit transactions as follows:

- `beginHT`: Begins a hardware transaction, returning `true`. If the transaction later aborts, control returns to the beginHT invocation, which returns a second time, with the value `false`.
- `commitHT`: Commits the hardware transaction.

The macro for acquiring a lock first starts a hardware transaction by calling `beginHT`, and then checks that the lock is not held by some other thread, aborting the transaction if the check fails. When the check succeeds, it executes the critical section transactionally, and the macro for releasing the lock attempts to commit the transaction. Notice that the first macro opens a lexical block that is closed by the second macro, so these macros must be used in block-structured pairs. If the transaction fails, the body of the while loop is executed, which may give up and acquire the lock the traditional way if the transaction fails too many times, setting a flag (`__IhaveLock`) so that the macro ending the critical section knows that it must execute the code to release the lock; otherwise `beginHT` is called again to retry the transaction. To avoid the so-called *lemming effect*, in which one operation acquiring the lock causes all the others to do so too, an operation waits until the lock is not held before retrying its transaction. For now, our simple prototype does not use the `cps` register to decide whether to retry: it just retries a fixed number of times (four in the experiments presented here), and then acquires the lock if it still does not succeed.

We first tried to apply this technique for implementing a thread-safe implementation of an STL data structure on `hash_map`. This experiment failed: we found that all transactions aborted. After a short investigation, we determined that the problem was that every `hash_map` operation computes the index of the appropriate hash bucket using the `sdivx` instruction, which is not allowed within transactions (see Section 2). For this experiment, we were interested in what can be achieved by putting a wrapper around code in a standard, unmodified library, so factoring this calcuation out of the transaction was not an option. We therefore tried applying the same technique to a different STL data structure, namely `vector`.

The `vector` data structure consists of an array of elements of some type and supports constant-time access to any element, constant-time insertion/deletion of elements at the end, and linear-time insertion/deletion at any other position. For workloads in which most operations on the vector are accesses to individual elements distributed over the vector, it seems likely that using HTM to elide a single lock protecting all operations on the vector could significantly improve performance by allowing many operations to execute in parallel. For workloads in which there are frequent internal insertions and deletions, HTM seems less likely to be effective because concurrent insertions and deletions affect a relatively larger amount of data and are more likely to conflict with each other, creating conditions under which the limited best-effort HTM transactions supported by ATMTP are unlikely to succeed.

We designed the following experiment to observe the performance impact of HTM-assisted lock elision: We start with a vector of 100 elements, each representing a counter initialized to 20. Each thread performs 10,000 iterations, each of which accesses a randomly chosen counter, performing an increment (10%), decrement (10%) or read (80%) on the counter. Operations that decrement a counter to 0 also delete the counter from the vector, and operations that increment a counter to 40 also "split" the counter by inserting a new one right after it, and setting both to 20.

As the number of threads increases, the chances of "underflowing" or "overflowing" a counter—resulting in a heavyweight insertion or deletion operation on the vector—increases. This way, we can observe performance with such operations being very rare and also with more of them occuring. (None occurred in experiments with fewer than four threads, at which point a few did occur; at 32 threads, well over 100 insertion and deletion operations occurred.)

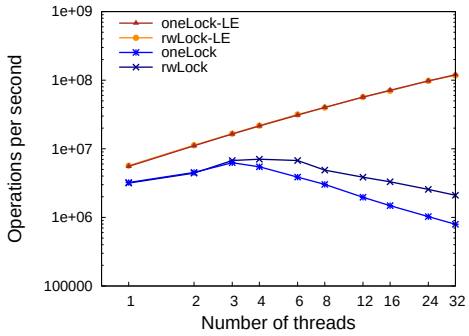

Figure 5. Performance experiments for STL `vector` with lock elision

Results shown in Figure 5 indicate that using HTM to elide the lock resulted in substantially better simulated performance, both when the heavyweight operations are rare and when they are more frequent. While we should be careful not to read too much into absolute performance numbers produced by such simulations, we are very encouraged that we were able to successfully execute a large fraction of operations using hardware transactions, and we believe this will translate to similar performance trends in real hardware. With the same caveats in mind, it is interesting to note that the HTM-assisted lock elision improved performance even in the single-threaded case. While only measurements on (precise performance models of) real machines can really confirm it, we believe it is reasonable to anticipate similar advantages in Rock. Neelakantam et al. [19] make a similar observation, again using a simulated system.

We also implemented a version of `vector` that used read-write locks to allow parallelism between read-only operations; our macros were sufficiently general to allow us to elide these locks too. When we did no lock elision, this optimization produced a noticeable performance improvement at higher levels of parallelism. However, in the versions with lock elision, the lines essentially overlap—primarily because the read count in the read-write lock is written by all readers, which inhibits scalability when the read operations are mostly quite short. A fine-grained locking version might perform somewhat better than the read-write lock version, but would entail significantly more programming complexity due to the need to avoid deadlock, and because of the dynamic nature of the data (insertion and deletions move data and thus complicate the mapping between data and the fine-grained locks).

3.5 Lock elision in libc

It is also possible to use best-effort HTM to elide locks in existing lock-based code by replacing the traditional acquire and release calls with calls to counterpart routines in a new library that supports lock elision in a manner similar to that described in the previous section. In this section, we discuss our experience in initial attempts to do this.

There are, of course, many lock-based libraries to which we might try to apply lock elision. For our experiment, we chose libc, focusing on the standard memory allocator that provides the `malloc` and `free` routines. While several scalable `malloc` implementations have been proposed, the standard libc implementation uses a single lock to protect its data structures. This is a common source of performance bottlenecks for C and C++ applications that rely heavily on dynamic memory allocation. We wanted to see if a

scalable implementation could be achieved by a simple application of lock elision.

Our initial results indicate that it is *not* effective to simply replace the lock acquire and release calls in `malloc` with the new transactional methods, for several reasons. First, because the standard malloc implementation uses a single global lock, there has been no motivation to distribute the data structures protected by this lock, so for example concurrent `malloc` calls for the same size all attempt to allocate the same chunk of memory, so they all conflict. Other problems include nested function calls executed inside transactions (see Section 2.2).

We experimented with small-scale changes to the library code to make it more amenable to lock elision, for example by eliminating function calls in the common case for `free()`. As a result we were successful in eliding the lock in `free()` a nontrivial fraction of the time, but this was still unsuccessful more often than not. Whether this would result in a performance gain or loss depends on precise costs of successful and unsuccessful transactions, of acquiring and releasing the lock, etc. Given that our simulator does not accurately model Rock's performance, we cannot draw a conclusion about this from our data. However, we doubt there is a significant gain here. Furthermore, because more scalable implementations already exist, it does not make sense to expend considerable effort to restructure the standard `malloc`/`free` code so that it is more amenable to lock elision.

Rather than making another guess about where lock elision might be useful, we have begun work on a tool that helps us look for critical sections that may prove amenable to lock elision. Specifically, we modified an internal simulator (not the one used for the rest of the results reported in this paper) to profile critical sections in running code and record the number of violations of Rock's limitations that occur in each one. The idea is that, if there are zero, then the critical section should be amenable to lock elision; if there are a small number, it may not be too difficult to restructure the code slightly to make it amenable; and if there are many this is probably not a promising critical section to elide.

We are still working on this tool as of the publication date, and we are not yet ready to present useful data from it. From our preliminary results, however, we observe that many transactions are counted as failing due to function calls (see Section 2.2). If this turns out to be correct, one option to consider is to have the compiler avoid the `save` and `restore` instructions[2]. Our tool is not yet sufficiently refined to allow us to determine how often this would be sufficient to make a transaction succeed that would otherwise fail. We plan to continue our work on the tool as well as using it to look for more opportunities for lock elision.

3.6 Eliding locks for Java synchronized blocks and methods

Finally, we discuss a particularly interesting opportunity to use lock elision to improve the scalability of existing code. The `synchronized` keyword in the Java programming language indicates that a particular code block or method should be executed while holding a lock (either a specified lock or the lock for the object on which a synchronized method is invoked). We can be a little more ambitious in this context because the decision of whether to attempt to elide a lock can be made by the JIT compiler, which can use run-time information to heuristically choose to elide locks for critical sections that seem likely to benefit from doing so, and in cases in which lock elision turns out to be ineffective, we can dynamically revert to the original locking code.

[2] This could be accomplished by supporting something like gcc's `-mflat` option [23] (which is no longer supported as of gcc 4.0.4), or by using stack manipulation tricks or heap memory to turn a called function into a leaf routine, which then won't necessarily need to allocate a new stack frame.

To explore the feasibility and potential value of using Rock's HTM feature to elide locks for synchronized blocks and methods in the Java programming language, we modified the Java HotSpot™ Virtual Machine. The results and conclusions presented below are similar to those presented by Goetz [5], who describes similar explorations using a different VM and different hardware support. Neelakantam et al. [19] also explore using HTM to elide locks in a different VM with different (simulated) hardware support.

Our prototype JVM attempts to elide locks for *all* synchronized blocks and methods. Although significant additional engineering work is clearly required to make this technique useful in general, running our simple prototype over ATMTP allows us to experiment with different synchronized blocks and methods to evaluate conditions under which they can be executed successfully using best-effort HTM. We report below on our first steps in this exploration.

Our prototype JVM implements lock elision for synchronized blocks by emitting special code for the `monitorenter` and `monitorexit` bytecodes. This code is essentially the same as the code for the macros we used to elide locks in Section 3.4 (see Figure 4): The code for `monitorenter` either acquires the lock in the traditional way and returns, or executes a `chkpt` instruction to begin transactional execution, checks that the lock is not held, and then sets a flag to record that the lock is being elided; if the lock is held, then it executes a `tcc` (trap on condition code) instruction to cause the transaction to fail. The code for `monitorexit` releases the lock in the traditional way if the lock is not being elided, and executes a `commit` instruction to attempt to complete transactional execution of the critical section if it is. If the transaction fails for any reason (including that the `monitorenter` code executes `tcc` because the lock is already held), control returns to a fail location in the code for `monitorenter`, where the transaction is retried. If too many retries occur, the lock is acquired and the critical section is executed nontransactionally. Lock elision for synchronized methods is implemented similarly.

We have not yet experimented extensively with the retry policy and for now it does not make use of the feedback provided in the `cps` register. Note, however, that because the choice to employ lock elision is made by the JIT compiler, many of the reasons for transaction failure can be eliminated. For example, it would not make sense for the compiler to attempt lock elision for a critical section that executes some instruction that is not allowed in hardware transactions, or that clearly performs more stores than the hardware will support. Therefore, dependence on the `cps` register may be lower in this context than in HyTM and PhTM, where transactions execute user code that is (at least for now) not analyzed by the compiler.

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.

Our initial experiment compares these collection classes with lock elision enabled to the original implementations under a simple read-only workload in which, after initialization, all worker threads repeatedly look up objects that are known to be in the mapping. We measure the total number of lookups achieved per second. This experiment is contrived to test our prototype in a simple case where it should clearly perform well. Nonetheless, it is not uncommon for these collections to be used in a read-only or read-mostly fashion, so the results even of this simple test may be relevant to some meaningful workloads.

Our intuition is that, if we are successful in executing lookups using best-effort HTM, our prototype implementation should exhibit significant improvement over the original synchronized versions in this experiment because concurrent lookups can execute

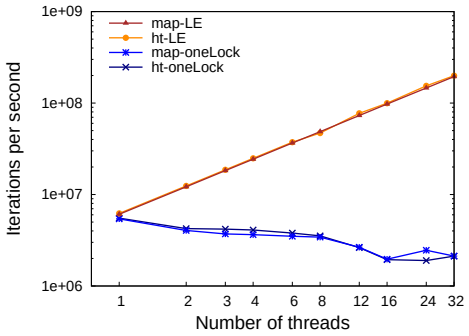

Figure 6. Performance experiments for Java hash table with lock elision

in parallel and there is no need to modify any data or metadata. In contrast, the original versions serialize all operations and require each operation to modify the lock even though lookup operations do not modify the collection.

At first we did not achieve the results we expected: all hardware transactions failed, so each operation just wasted time before falling back on the original locking mechanism. Our investigation into the reasons for this yielded two causes.

The first cause was a simulator bug: the `tcc` (trap on condition code) instruction caused an exception regardless of whether the condition held; we use this instruction to cause the transaction to fail in case the lock is held by another thread, so transactions were failing even when it was not held. Tracing this bug motivated us to extend the simulator with a special breakpoint so that we can dump the registers during transactional execution. This proved to be a valuable debugging tool. The ability to provide greater visibility into transactional execution than will be available with the real Rock hardware illustrates the value of ATMTP even after Rock is available.

The second source of failure turned out to be a repeat of the experience reported in Section 3.4: `HashTable` uses the `sdivx` instruction to compute the bucket index, thereby failing all transactions. Working around this problem required a simple change to the `HashTable` code, in which we compute the bucket index before acquiring the `HashTable` lock. This way, when we use a hardware transaction to elide the lock, the `sdivx` instruction is not executed inside the transaction. Note that it is necessary in this case to check inside the critical section that the table size is the same value as was read before, otherwise the computed bucket index may be incorrect.

The `HashMap` did not require this change because it ensures that the number of buckets is a power of two and uses bit shifting and masking operators rather than `sdivx` to compute the bucket index. Therefore, `HashMap` is another example in which unmodified existing code is amenable to lock elision.

With these issues addressed, we achieved the results shown in Figure 6. For both of the unmodified collections, throughput *decreases* with additional threads. This is because traditional synchronized methods not only serialize the operations, but also generate coherence traffic because each operation must acquire exclusive ownership of the lock metadata. In contrast, with lock elision enabled, both collections exhibit good scalability, as we had hoped, resulting in substantially improved throughput relative to the original synchronized methods as thread count increases. (These two lines are difficult to distinguish in the graph because they almost exactly overlap throughout the range.) This establishes that at least for simple read-only operations, lock elision offers the potential for significantly improved performance and scalability for unmodified code. Our next step is to experiment with operations that modify the collection during the experiment, and other more diverse workloads.

4. Concluding remarks

Sun's forthcoming Rock processor will support best-effort hardware transactional memory (HTM). Our Adaptive Transactional Memory Test Platform (ATMTP) provides a platform on which we and others can develop and test code that exploits this feature.

This paper describes our early work using ATMTP to explore potential applications of Rock's HTM feature, how its limitations interact with target workloads, whether and how they can be overcome in various situations, and what changes to future HTM features might enable more widespread and more effective use of those features. While our evaluation of these techniques is preliminary, we have demonstrated our progress on a number of tools (libraries, compilers, and simulators), which provide a rich environment for experimenting with the use of Rock's HTM feature.

Our early investigation has yielded some encouraging results, giving us confidence that we will be able to exploit Rock's HTM feature to achieve significant performance and scalability improvements in a variety of contexts. We have also encountered a number of challenges and learned some lessons, which fall into the following broad categories:

Exploiting HTM on Rock Many of the lessons learned translate more or less directly to Rock:

- We sometimes need to restructure code slightly to avoid instructions that fail transactions (such as `sdivx` in our `hash_map` example);
- We often need to avoid nested function calls; this is sometimes as simple as inlining, but it sometimes requires more significant code restructuring (such as replacing our recursive red-black tree with an iterative one);
- The `cps` register provides valuable feedback for making intelligent decisions as to when/whether to retry; and
- Nontrivial engineering is required to avoid repeated transaction failures due to TLB misses.

Simulation methodology Other lessons and challenges are more related to our use of ATMTP:

- Deterministic simulation aids debugging, and the simulator gives more visibility into reasons for transaction failure than Rock will; for these reasons, we expect ATMTP to continue to be a valuable tool even after we have Rock-based systems.
- On the other hand, simulation speed limits how much we can learn. For example, we expect that JIT compilation will stabilize after a couple of seconds of real time, but it is impractical for us to simulate for this long, preventing us from easily exploring the best approaches for using HTM in Java programs on Rock for now.
- Differences between Rock and the simulated system also limit development and evaluation of the best techniques for Rock. For example, we believe that dealing with ITLB misses will be easier on Rock than ATMTP, suggesting it is not worthwhile to address these problems yet. Similarly, tuning various mechanisms will require more accurate models of various costs on Rock, or waiting for Rock.

An alert reader may have noticed that our discussion of the nonblocking techniques discussed in Sections 3.2 and 3.3 did not address the possibility of the simple transactions they use failing forever, as is usually required for any technique that uses best-effort

HTM. While we have advocated best-effort HTM designs as a way to make earlier implementation feasible, we have also stressed the value of making some stronger guarantees for specific classes of simple transactions. Such guarantees give substantial power to algorithm designers, especially for nonblocking algorithms. We expect that Rock will provide a guarantee sufficient to support the techniques discussed in Sections 3.2 and 3.3. We have not determined whether ATMTP provides such a guarantee, but in practice such transactions do eventually succeed, which allows us to experiment with mechanisms that do not provide a software alternative. Continuing to demonstrate the power of these guarantees, as well as helping to understand what guarantees suffice and how they can be supported, is an important part of our ongoing work.

Finally, we will soon release ATMTP as open source, and it will be integrated into the forthcoming GEMS 2.1 release from the Multifacet group at the University of Wisconsin. Please check our website [25] and theirs [29] for details and updates. Releasing our simulator—along with some example code showing how to use the Rock HTM instructions—will allow others to take advantage of our work to conduct research of their own, either experimenting with code intended for Rock, or modifying the simulator to experiment with the impact of different design decisions on the effectiveness of best-effort HTM.

Acknowledgments

We are grateful to Shailender Chaudhry, Bob Cypher, and Marc Tremblay for useful conversations about Rock's design and behavior. We also thank the anonymous referees for their excellent reviews.

References

[1] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

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

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

[4] K. Fraser. *Practical Lock-Freedom*. PhD thesis, Cambridge University Technical Report UCAM-CL-TR-579, Cambridge, England, Feb. 2004.

[5] B. Goetz. Optimistic thread concurrency: Breaking the scale barrier. Technical Report AWP-011-010, Azul Systems, Inc., 2006.

[6] T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *International Conference on Distributed Computing (DISC)*, 2001.

[7] S. Heller, M. Herlihy, V. Luchangco, M. Moir, N. Shavit, and W. N. Scherer III. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.

[8] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, 2003.

[9] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *21st Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 253–262, 2006.

[10] 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.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.

[12] D. Lea. `java.util.concurrent.ConcurrentSkipListMap`.

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

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

[15] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[16] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[17] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *14th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2002.

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

[19] 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.

[20] 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.

[21] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[22] C. J. Rossbach, O. S. Hoffman, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing hardware transactional memory in the operating system. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, 2007.

[23] R. M. Stallman and the GCC Developer Community. *Using the Gnu Compiler Collection*. Free Software Foundation. `http://gcc.gnu.org/onlinedocs`.

[24] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), HP Laboratories, Nov. 1995.

[25] Sun Microsystems Laboratories. Scalable Synchronization Research Group. `http://research.sun.com/scalable`.

[26] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking zero-indirection transactional memory. In *Workshop on Transactional Computing (Transact)*, 2007.

[27] M. Tremblay. Transactional memory for a modern microprocessor. Keynote speech at 26th Annual ACM Symposium on Principles of Distributed Computing, Aug. 2007.

[28] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC® processor. In *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[29] Wisconsin Multifacet Project. Multifacet GEMS: General Execution-driven Multiprocessor Simulator. `http://www.cs.wisc.edu/gems`.