

# Simplifying Concurrent Algorithms by Exploiting Hardware Transactional Memory

Dave Dice  
Sun Labs  
dave.dice@oracle.com

Mark Moir  
Sun Labs  
mark.moir@oracle.com

Yossi Lev  
Sun Labs, Brown Univ.  
levyossi@cs.brown.edu

Dan Nussbaum  
Sun Labs  
dan.nussbaum@oracle.com

Virendra J. Marathe  
Sun Labs  
virendra.marathe@oracle.com

Marek Olszewski  
Sun Labs, MIT  
mareko@csail.mit.edu

## ABSTRACT

We explore the potential of hardware transactional memory (HTM) to improve concurrent algorithms. We illustrate a number of use cases in which HTM enables significantly simpler code to achieve similar or better performance than existing algorithms for conventional architectures. We use Sun's prototype multicore chip, code-named Rock, to experiment with these algorithms, and discuss ways in which its limitations prevent better results, or would prevent production use of algorithms even if they are successful. Our use cases include concurrent data structures such as double ended queues, work stealing queues and scalable non-zero indicators, as well as a scalable malloc implementation and a simulated annealing application. We believe that our paper makes a compelling case that HTM has substantial potential to make effective concurrent programming easier, and that we have made valuable contributions in guiding designers of future HTM features to exploit this potential.

## Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming

## General Terms

Algorithms, Design, Performance

## Keywords

Transactional Memory, Synchronization, Hardware

## 1. INTRODUCTION

This paper explores the potential of hardware transactional memory (HTM) to simplify concurrent algorithms, data structures, and applications. To this end, we present a number of relatively simple algorithms that use HTM to solve problems that are substantially more difficult to solve in conventional systems.

At the risk of stating the obvious, simplifying concurrent algorithms has many potential benefits, including improving the readability, maintainability, and flexibility of code, making concurrent

programming tractable for more programmers, and adding to the array of techniques for programmers to use to exploit concurrency. Furthermore, simplifying code by separating program semantics from implementation details enables applications to benefit from platform-specific implementations and future improvements thereto.

Our experiments use the HTM feature of a prototype multicore processor developed at Sun, code named Rock. Our aim is to demonstrate the potential of HTM *in general* to simplify concurrent algorithms, not to evaluate Rock's HTM feature (this is reported elsewhere [11, 12]). In some cases we use Rock's HTM feature in a way that may not be suitable for production use. Throughout the paper, we attempt to illuminate the properties of an HTM feature required for a particular technique to be successful and acceptable to use. We hope that our observations in this regard are helpful to designers of future HTM features.

The examples we present merely scratch the surface of the potential ways HTM can be used to simplify and improve concurrent programs. Nonetheless, we believe that they yield valuable contributions to understanding of the potential of HTM, and important observations about what must be done in order to exploit it.

In Section 2, we review a number of techniques that employ HTM. Section 3 presents our first use of HTM, implementing a concurrent double-ended queue (deque), which is straightforward with transactions, and surprisingly difficult in conventional architectures. Next, in Section 4, we examine an important restricted form of deque called a *work stealing queue* (ws-queue), which is at the heart of a number of parallel programming patterns. In Section 5, we use HTM to simplify the implementation of Scalable Non-Zero Indicators (SNZIs), which have been shown to be useful in improving the scalability of software TM (STM) algorithms and readers-writer locks. Next, in Section 6, we show how HTM can be used to obviate the need for special kernel drivers to support a scalable malloc implementation that significantly outperforms other implementations in widespread use. Finally, in Section 7, we explore the use of HTM to simplify a simulated annealing application from the PARSEC benchmark suite [4], while simultaneously improving its performance. We summarize our observations and guidance for designers of future HTM features in Section 8, and conclude in Section 9.

## 2. TECHNIQUES FOR EXPLOITING HTM

Given hardware support for transactions, simple wrappers can be used to execute a block of code in a transaction. Such wrappers can diagnose reasons for transaction failures and decide whether to back off before retrying, for example. A *best-effort* HTM feature such as Rock's [11, 12] does not guarantee to be able to commit a given transaction, even if it is retried repeatedly. In this case, an al-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA '10, June 13–15, 2010, Thira, Santorini, Greece.

Copyright 2010 ACM 978-1-4503-0079-7/10/06 ...\$10.00.

ternative software technique is needed in case a transaction cannot be committed. The use of such alternatives can be made transparent with compiler and runtime support. This is the approach taken by Hybrid TM (HyTM) [7] and Phased TM (PhTM) [27], which support transactional programs in such a way that transactions can use HTM, but can also transparently revert to software alternatives when the HTM transactions do not succeed.

Transactional Lock Elision (TLE) [10, 36] aims to improve the performance of lock-based critical sections by using hardware transactions to execute nonconflicting critical sections in parallel, without acquiring the lock. When such use of HTM to elide a lock acquisition is not successful, the lock is acquired and the critical section executed normally. TLE is similar to Speculative Lock Elision as proposed by Rajwar and Goodman [34], but is more flexible because software rather than hardware determines when to use a hardware transaction and when to acquire the lock. Compared to HyTM and PhTM, TLE imposes less overhead on single-threaded code, and requires less infrastructure, but puts the burden back on the programmer to determine and enforce locking conventions, avoid deadlocks, etc., and furthermore, does not compose well.

### 3. DOUBLE-ENDED QUEUES

In this section, we consider a concurrent double-ended queue (*deque*). The `LinkedBlockingDeque` implementation included in `java.util.concurrent` [23] synchronizes all accesses to a deque using a single lock, and therefore is blocking and does not exploit parallelism between concurrent operations, even if they are at opposite ends of the deque and do not conflict.

Improving on such algorithms to allow nonconflicting operations to execute in parallel is surprisingly difficult. The first obstruction-free deque algorithms, due to Herlihy *et al.* [20] are complex and subtle, and require careful correctness proofs.

Achieving a stronger nonblocking progress property, such as lock-freedom [17], is more difficult still. Even lock-free deques that do not allow concurrent operations to execute in parallel are publishable results [30], and even using sophisticated multi-location synchronization primitives such as DCAS, the task is difficult enough that incorrect solutions have been published [8], fixes for which entailed additional overhead and substantial verification efforts [14].

Even if we do not require a nonblocking implementation, until recently, constructing a deque algorithm that allows concurrent opposite-end operations without deadlocking has generally been regarded to be difficult [22]. In fact, the authors only became aware of such an algorithm *after* the initial version of this paper was submitted. Paul McKenney presents two such algorithms in [29]. While the simpler of the two (which uses two separate single-lock deques for head and tail operations) is relatively straightforward in hindsight, it was not immediately obvious even to some noted concurrency experts [18, 28]. In fact, McKenney invented the more complex algorithm first. This algorithm, which hashes requests into multiple single-lock deques, is not at all straightforward, and yields significantly lower throughput than the simpler one does.

In contrast, the transactional implementation is no more complex than sequential code that could be written by any competent programmer, regardless of experience with concurrency. We believe such implementations are generally straightforward given adequate support for transactions. Essentially, we just write simple, sequential code and wrap it in a transaction. The details of exploiting parallelism and avoiding deadlock are thus shifted from the programmer to the system. In addition to significantly simplifying the task of the programmer, this also establishes an abstraction layer that allows for portability to different architectures and improvements over time, without modifying the application code.

Our experimental harness creates a deque object initially containing five elements, and spawns a specified number of threads, dividing them evenly between the two ends of the deque. Each thread repeatedly and randomly pushes or pops an element on its end of the deque, performing 100,000 such operations. We measure the interval between when the first thread begins its operations and when the last thread completes its operations. Each data point presented is the geometric mean of three values obtained by omitting the maximum and minimum of five measured throughputs.

We test a variety of implementations, varying synchronization mechanisms and the algorithm that implements the deque itself. A simple unsynchronized version (labeled *none* in Figure 1) gives a sense of the cost of achieving correct concurrent execution. We test several non-HTM versions, including a compiler-supported STM-only implementation using the TL2 STM [13], two direct-coded single-lock implementations (pthread lock, hand-coded spinlock) and the simpler of McKenney’s lock-based algorithms [29]. Using HTM, we test direct-coded and compiler-supported HTM-only implementations, a compiler-supported PhTM [27] implementation (using TL2 when in the software phase) and a compiler-supported HyTM [7] implementation (using the SkySTM STM [25]). Finally, we test a TLE [10, 36] implementation combining HTM and our hand-coded spinlock.

Our deque implementations do not admit much parallelism between same-end operations, and threads perform deque operations as fast as they can, with relatively little “non critical” work between deque operations. We therefore do not expect much more than a 2x improvement over single-threaded throughput.

Figure 1 presents the results of our deque experiments. First, note that the single-thread overhead for the various synchronization mechanisms ranges between factors of 2.5 and 4 (except for STM-only synchronization, for which the slowdown is much worse).

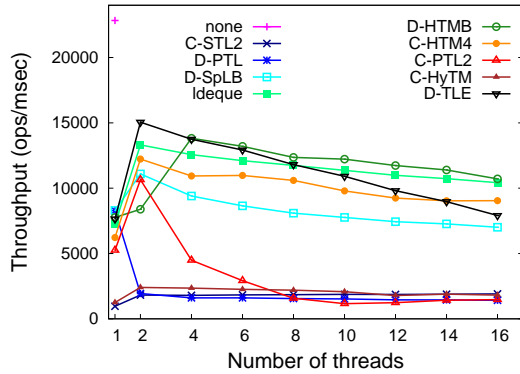
STM-only synchronization (labeled *C-STL2* in Figure 1) uses compiler support built on a TL2-based [13] runtime system. Overheads for STM-only execution are significant, with a single-thread run achieving only 12% of the pthread-lock version’s throughput.

Next we consider lock-based implementations. The pthreads lock implementation that comes with Solaris™ (*D-PTL*) yields a 77% decrease in throughput going from one thread to two, with a continuing decrease (to 83%) as we go out to sixteen threads. To factor out possible effects of using a general-purpose lock that parks and un parks waiting threads, and the Solaris™ implementation thereof in particular, we also test a simple hand-coded spinlock.

The hand-coded spinlock (*D-SplB*) yields essentially the same single-thread throughput as the pthreads lock, and a 34% *speedup* on two threads, dropping off a bit at higher thread counts. It may seem counterintuitive that *any* speedup is achieved with a single-lock implementation, but this is possible because there is some code that executes between the end of one critical section and the beginning of the next, which can be overlapped by multiple threads.

McKenney’s two-queue algorithm [29] (*Ideque*) performs well. Its single-thread performance is only 13% lower than that of the pthreads lock, and it achieves nearly a 2x speedup at two threads, most of which it maintains out to sixteen threads. This algorithm is nearly the best across the board, only being outperformed (slightly) by the direct-coded HTM-only implementation.

The direct-coded HTM-based implementation without backoff (not shown) generally fails to complete within an acceptable period of time at larger thread counts, due to excessive conflicts. This is consistent with our previous experience [11, 12]: due to Rock’s simple “requester-wins” conflict resolution mechanism, transactions can repeatedly abort each other if they are re-executed immediately; this problem can be addressed with a simple backoff mechanism.



**Figure 1: Deque benchmark. Key:** *none*: unsynchronized. *C-STL2*: Compiled STM-only (TL2). *D-PTL*: Direct-coded single-lock (pthreads). *D-SpLB*: Direct-coded single-lock (spin-lock with backoff). *ldeque*: McKenney’s two-lock deque implementation. *D-HTMB*: Direct-coded HTM-only (with backoff). *C-HTM4*: Compiled HTM-only. *C-PTL2*: Compiled PhTM (TL2 STM). *C-HyTM*: Hybrid TM. *D-TLE*: Direct-coded TLE.

The direct-coded HTM-only implementation (*D-HTMB*) employs such a backoff mechanism, implemented in software. This version yields 94% of the single-thread throughput yielded by the pthreads-lock version, achieving nearly a factor of two speedup when run on four or more threads and maintaining most of that speedup out to sixteen threads. (We have not investigated in detail why the expected performance increase is not realized at two threads.)

The compiler-supported HTM-only implementation (*C-HTM4*) performs similarly to the direct-coded implementation but associated compiler and runtime infrastructure reduces throughput by about 20% over most of the range.

PhTM (*C-PTL2*) incurs significant overhead, yielding 63% of the pthreads-lock’s single-thread throughput, but increasing throughput by a factor of 1.9 going from one thread to two. However, this throughput degrades severely at higher threading levels because a large fraction of transactions are executed in software.

Two factors contribute to this poor performance. First, it can be difficult to diagnose the reason for hardware transaction failure on Rock and to construct a general and effective policy about how to react [11, 12]. Given our results with the HTM-only implementations, it seems that PhTM should not need to resort to using software transactions for this workload, but our statistics show that this happens reasonably frequently, especially at higher concurrency levels. Coherence conflicts are the dominant reason for hardware transaction failure, so we believe our PhTM implementation could achieve better results by trying harder to complete the transaction using HTM before failing over to software mode.

Second, in our current PhTM implementation, when one transaction resorts to using STM, all other concurrent (HTM) transaction attempts are aborted; when they retry, they use STM as well. Subsequently, to ensure forward progress, we do not attempt to switch back to using HTM (system-wide) until the thread that initiated the switch to software mode completes. Furthermore, we do not attempt to prioritize the transaction being run by that thread, or to aggressively switch back to hardware mode when it is done—instead, when that transaction finishes, all other concurrently-running software transactions are allowed to finish before the switch back to hardware mode is made. All of these observations point to significant opportunities to improve the performance of PhTM for

this workload, which we have not yet attempted. Nevertheless, we should not underestimate the difficulty of constructing efficient policies that are effective across a wide range of workloads.

HyTM (*C-HyTM*) has even more overhead than PhTM, yielding only 15% of the pthreads-lock’s throughput on a single thread. HyTM’s instrumentation of the hardware path is responsible for most of the slowdown. HyTM does achieve about a factor of two speedup on two threads, maintaining most of that advantage out to sixteen threads, outperforming PhTM on eight or more threads.

While the HTM-only results reported above indicate strong potential for hardware transactions to make some concurrent programming problems significantly easier, we emphasize that there is no guarantee that Rock’s HTM will not repeatedly abort transactions used by deque operations. Without such a guarantee, we could not recommend using the HTM-only implementations in production code, even though it works reliably in these experiments. Furthermore, our PhTM and HyTM results illustrate the overhead and complexity of attempting to transparently execute transactions in software when the HTM is not effective.

Transactional Lock Elision (TLE) [10] (*D-TLE*) yields good performance over the entire range; in fact, on two threads, it is best of any of the variants tested. While this is encouraging for the use of best-effort HTM features, we note that TLE gives up several advantages of the transactional programming approach, such as composability of data structures implemented using it and the ability to use it to implement nonblocking data structures. Whether a TM system is useful for building nonblocking data structures depends on properties of the HTM, as well as properties of the software alternative used in case hardware transactions fail. The original motivation for TM was to make it easier to implement nonblocking data structures, so designers of future HTM features should consider the ability of a proposed implementation to do so, in addition to the following conclusions we draw from our experience:

- If guarantees are made for small transactions, such that there is no need for a software alternative, TM-based implementations of concurrent data structures that use such transactions are easier to use and are more widely applicable.
- Better conflict resolution policies than Rock’s simple request-wins can reduce the need for aggressive backoff, which may be difficult to tune in a way that is generally effective.
- Avoiding transaction failures for relatively obscure reasons, such as sibling interference [11, 12], and providing better support for diagnosing the reasons for transaction failures, significantly improves the usefulness of an HTM feature.

## 4. WORK STEALING QUEUES

In this section, we discuss the use of HTM in implementing *work stealing queues* (ws-queues) [1, 6], which are used to support a number of popular parallel programming frameworks. In these frameworks, a runtime system manages a set of *tasks* using a technique called *work stealing* [1, 6, 16]. Briefly, each thread in such a system repeatedly removes a task from its ws-queue and executes it. Additional tasks produced during this execution are pushed onto the thread’s ws-queue for later execution. For load balancing purposes, if a thread finds its queue empty, it can *steal* one or more tasks from another thread’s ws-queue.

The efficiency of the ws-queues, especially for the common case of accessing the local ws-queue, can be critical for performance. As a result, a number of clever and intricate ws-queue algorithms have been developed [1, 6, 16]. In most cases, a thread pushes and pops tasks to and from one end of its ws-queue, and stealers steal

from the other end. Thus, only the owner accesses one end, and only pop operations are executed on the other.

Existing ws-queue implementations [1, 6, 16] exploit these restrictions to achieve simpler and more efficient implementations than are known for general double-ended queues. Nonetheless, these algorithms are quite complex, and reasoning about their correctness can be a daunting task. As an illustration of the complexity of such algorithms, querying a Sun internal bug database for “work stealing” yields 21 hits, all of which are related to the work stealing algorithm used by the HotSpot Java VM’s garbage collector, and a search for bugs tagged with the names of the files in which the work stealing algorithm is implemented yields 360 hits, many of which are directly related to tricky concurrency-related bugs.

In this section, we present several transactional work stealing algorithms, which demonstrate tradeoffs between simplicity, performance, and requirements of the HTM feature used. We have evaluated these algorithms on Rock using the benchmark used in [6]. Briefly, this benchmark simulates the parallel execution of a program represented by a randomly generated DAG, each node of which represents a single task. A node’s children represent the tasks spawned by that node’s task. The parameters  $D$  and  $B$  control the depth of the tree and the maximum branching factor of each node, respectively; see [6] for details. For this paper, we concentrate on medium sized trees generated using  $D=16$  and  $B=6$ ; our experiments with other values yield similar conclusions. The ws-queue array’s size is initialized to 128 entries. We measure the time to “execute” the whole DAG, and we report the result as throughput in terms of tasks processed per millisecond. For each point, we discard the best and the worst of five runs, and report the geometric mean of the remaining three. We observed occasional variability for all algorithms, which we believe is related to architecture and system factors rather than the algorithms themselves.

The results of our experiments are presented in Figure 3. All of the algorithms scale well, which is not surprising given that concurrent accesses to ws-queues happen only as a result of stealing, which is rare. The Chase-Lev (CL) [6] algorithm provides the highest throughput, and the algorithm due to Arora *et al.* (ABP) [1] provides about 96% of the throughput of CL.

We begin with a trivial algorithm that stores the elements of the ws-queue in an array, and implements all operations by enclosing simple sequential code in a transaction. When a pushTail operation finds the array full, it “grows” the array by replacing it with a larger array and copying the relevant entries from the old array to the new one. Similarly, when a popTail operation finds a drop in the size of the ws-queue, with respect to the size of the array, below a particular threshold (one-third, only if the array size is greater than 128, in our experiments), it “shrinks” the array by replacing it with a smaller array. (Note that although the array did not grow or shrink in our experiments reported here, it did grow and shrink a few times in some other experiments, with no noticeable performance impact.) This algorithm, executed using PhTM (see Section 2), scales as well as ABP and CL (see curve labeled “PhTM (all)”), but provides only about 68% of the throughput of CL.

In our experiments, nearly all transactions succeeded using HTM, so the performance gap between PhTM and CL is mainly due to the overhead of the system infrastructure for supporting transactions (including the latency of hardware transactions). Nonetheless, unless the HTM can commit transactions of *any* size, the trivial algorithm requires a software alternative such as PhTM provides—and associated system software complexity and overhead—due to the occasional need to grow or shrink the size of the ws-queue.

We therefore modified the algorithm to avoid large transactions altogether, in order to explore what could be achieved by directly

```

1 WSQueue {
2   volatile int head;
3   volatile int tail;
4   int size;
5   Value[] array;
6 }
7
8 void WSQueue::pushTail(Value new_value)
9 {
10  while (true) {
11    BEGIN_TXN;      // delete for nontxl
12    if (tail - head != size) {
13      array[tail % size].set(new_value);
14      tail++;
15      return;      // commits, see caption
16    }
17    COMMIT_TXN;    // delete for nontxl
18    grow();
19  }
20 }
21
22 void WSQueue::grow()
23 {
24   int new_size = size * 2;
25   copyArray(new_size);
26 }
27
28 void WSQueue::shrink()
29 {
30   int new_size = size / 2;
31   copyArray(new_size);
32 }
33
34 void WSQueue::copyArray(int new_size)
35 {
36   Value[] old_array = array;
37   Value[] new_array = new Value[new_size];
38   for (int i=head; i<tail; i++) {
39     new_array[i % new_size].set
40       (array[i % size].get());
41   }
42   BEGIN_TXN;
43   array = new_array;
44   size = new_size;
45   COMMIT_TXN;
46   delete old_array;
47 }
48
49 Value WSQueue::stealHead()
50 {
51   BEGIN_TXN;
52   return (head < tail) ?
53     array[head++ % size].get() : Empty;
54   COMMIT_TXN;
55 }
56
57 Value WSQueue::popTail() // Tx1 version
58 {
59   Value tailValue;
60   BEGIN_TXN;
61   tailValue = (tail == head) ?
62     Empty : array[--tail].get();
63   COMMIT_TXN;
64   if (sizeBelowThreshold()) shrink();
65   return tailValue;
66 }
67
68 Value WSQueue::popTail() // Nontxl version
69 {
70   tail--;
71   MEMBAR_STORE_LOAD; // see text
72   int h = head;
73   // head = h; // see text
74   if (tail < h) {
75     tail++; // failed; undo increment
76     return Empty;
77   }
78   if (sizeBelowThreshold()) shrink();
79   return array[tail % size].get();
80 }

```

Figure 2: Work stealing pseudocode. Returning from within a transaction commits the transaction.

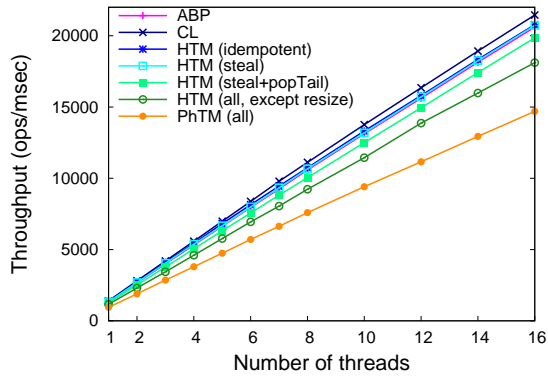


Figure 3: Work stealing benchmark.  $D = 16$ ,  $B = 6$ .

using an HTM that makes guarantees only for small transactions. The first such algorithm is presented in Figure 2. To avoid using large transactions for operations that grow the ws-queue, we modified the pushTail operation so that it commits without attempting to increase the size of the ws-queue if it observes that the ws-queue is full (see line 12). In this case, the thread executing the operation then calls `grow` (line 18) *outside* the transaction.

The `grow` procedure calls `copyArray`, which does array allocation and copying nontransactionally (lines 37–41), and uses a transaction to make the new array current, and record its size (lines 42–45). Like the Chase-Lev algorithm [6], `grow` does not need to modify the head and tail variables.

Similarly, to avoid large transactions for operations that shrink the ws-queue, we modified the popTail operation so that the transaction contains only the code that pops an item from the ws-queue (see lines 61 and 62). Like the `grow` procedure, the `shrink` procedure does its array allocation and copying nontransactionally, and uses a transaction to make the new array current.

The algorithm with this modification is slightly more difficult to reason about because of the possibility of stealHead operations being executed concurrently with growing or shrinking the array. However, much of the simplicity of the trivial algorithm is retained: All of the operations that modify the state of the ws-queue are still executed entirely within transactions, and we only need to consider complete executions of the stealHead operation between steps of the `grow` and `shrink` procedures, rather than considering arbitrary interleavings. Concurrent stealHead operations can only reduce the relevant portion of the array for copying, but no harm is done if stolen elements are unnecessarily copied, as they are outside the range specified by head and tail.

Previous ws-queue algorithms [1, 6] have been carefully optimized to avoid expensive synchronization primitives (such as CAS) in common-case pushTail and popTail operations. As a result, these more complex algorithms are likely to outperform our simple HTM-based algorithm. Our results confirm that this is the case on Rock. Specifically, the modified algorithm—labeled “HTM (all, except resize)” in Figure 3—improves on the trivial PhTM algorithm by roughly 23%, but still provides only about 84% of the throughput of CL. Therefore, we explored whether we could eliminate transactions from the common-case operations, while still using them in less common operations to keep the algorithm simple.

First, we modified pushTail to not use a hardware transaction in the common case (it still uses one in `copyArray`). The resulting algorithm—labeled “HTM (steal+popTail)” in Figure 3—performs

about 10% better than the modified algorithm described above, and comes very close to matching the hand-crafted ABP algorithm.

However, reasoning that the algorithm remains correct with this change becomes somewhat more difficult for several reasons. The order of the store to tail and to the array element is now important, whereas in the transactional version, they could have been written in either order. This affects not only the difficulty of reasoning about the algorithm, but also how it is expressed: We are prevented from using the compact post-increment notation to increment tail, as in the transactional stealHead and popTail operations.

Furthermore, head may change during the pushTail operation. As a result, the pushTail operation may grow the array unnecessarily if a concurrent stealHead operation has made a slot available in the array. This behavior is benign in terms of correctness, but the algorithm is at least somewhat more complex because of the need to reason about it. Even so, this algorithm is still considerably simpler and easier to reason about than the previous work stealing algorithms, and its performance is very close to theirs.

We next modified the algorithm so that popTail also does not use transactions. We must now reason about concurrent interleavings of popTail and stealHead operations, whereas this was not necessary when both were executed as transactions. As before, the fact that the only operations that can execute during the popTail operation are stealHead operations executed using transactions makes this reasoning fairly manageable.

However, now a more subtle issue arises. Since the transactional version of popTail is executed atomically, the order of its accesses to head and tail is unimportant. In the nontransactional version, it is critical that popTail updates tail and *then* reads head to determine whether the ws-queue was empty when tail was modified (in which case we need to undo this modification, and return `Empty`). In many memory consistency models, including TSO [37] (which is supported by Rock), the load of head may be reordered before the store to tail. To avoid this, a `membar #storeload` instruction is required (line 71). Identifying the problem and reasoning that the memory barrier solves it is considerably more complex than thinking about the less aggressively optimized versions. Furthermore, the memory barrier makes the popTail code less compact and less readable than the transactional version.

Nonetheless, using a transactional stealHead operation still makes it considerably easier to reason about this algorithm than existing nontransactional algorithms [1, 6], in which popTail uses CAS to remove the last element, to avoid a race with a concurrent stealHead operation that cannot occur if stealHead is transactional. This algorithm—labeled “HTM (steal)” in Figure 3—performs comparably with ABP, and delivers about 96% of the throughput of CL.

Researchers have recently observed [24, 33] that in some contexts, *idempotent work stealing*, in which an element may be returned from a ws-queue multiple times, suffices for some applications. They show that, given this weaker semantics, the memory barrier discussed above can be elided. However, their algorithms are not much less complex than the existing algorithms, and are considerably more complex than our algorithms that use HTM.

Interestingly, given the weaker semantics required by idempotent work stealing, we too can eliminate the memory barrier from the popTail operation. However, this change alone results in an algorithm in which a popTail operation and a concurrent stealHead operation can each think the other took the last element from the ws-queue, which results in an element being lost. We overcome this problem by performing an additional store in popTail in order to “undo” the effect of potential concurrent stealHead operations (see line 73). The resulting algorithm is significantly simpler than others [24, 33], again due to the use of hardware transactions for

the stealHead operation. Whether this algorithm yields any performance benefit over the version with the memory barrier depends on details of the architecture; for Rock, where memory barriers are inexpensive, we observed little significant improvement.

Although our first modification eliminated the large transactions associated with growing and shrinking the ws-queue, it may seem that implementations that use HTM directly for other operations require guarantees for at least small transactions. Interestingly, however, with a little care, successful stealing is not necessary to ensure forward progress of the overall application. In particular, even if all stealHead operations are unsuccessful, eventually every thread will complete the work in its own ws-queue and the application will complete. Thus, we can use a purely best-effort HTM feature with the algorithm that uses hardware transactions only for stealHead operations, but not for the simpler algorithms that use transactions for pushTail and/or popTail.

Finally, we note that our HTM-based algorithms can deallocate the old array immediately after replacing it with a new one (see line 46 in Figure 2) because the owner is guaranteed not to access the old array again, and any concurrent stealHead operations accessing the old array fail when the new array is installed. This behavior of the stealer depends on the assumption that the hardware transaction is effectively “sandboxed” – a transaction either aborts immediately when something it has read changes, or its inconsistent state is not observable to the external world. This avoids the need for complex and expensive memory management techniques such as hazard pointers [31] or Repeat Offender mechanisms [19], or simpler but wasteful techniques such as not deallocating the old array, which are often used in practice.

## 5. SCALABLE NON-ZERO INDICATORS

Next we explore using HTM to simplify and improve the performance of Scalable Non-Zero Indicators (or SNZI, which is pronounced “snazzy”) [15]. SNZI objects have been used in scalable STM implementations [25] and readers-writer locks [26]. SNZI supports Arrive, Depart and Query operations, with Query indicating whether there is a *surplus* of arrivals (i.e., there have been more Arrive than Depart operations).

A SNZI implementation based on a single counter is trivial but not scalable. Previous scalable SNZI algorithms use a tree of SNZI nodes: Arrive and Depart operations on a child may invoke Arrive and/or Depart on its parent, but only when the surplus at the child might change from zero to nonzero, or vice versa. These algorithms maintain the following SNZI invariant: *every SNZI node has a surplus iff it has a descendant that has a surplus or the number of Arrive operations that started at the node is greater than the number of Depart operations that started at the node*. This way, threads may arrive at any node (the corresponding depart should start at the same node), and propagate upward only as necessary to maintain the invariant. Queries are made directly at the root. Note that an arrival at a node must propagate up the tree only if the node’s surplus is zero, and a departure must propagate only if it will make the surplus zero. This way, SNZI nodes act as “filters” for nodes higher in the tree, thus reducing contention and improving scalability.

A key difficulty in previous CAS-based SNZI algorithms arises when multiple threads concurrently arrive at a node with zero surplus. Ensuring that exactly one of them propagates a surplus up the tree in a nonblocking manner involves first attempting to do the propagation, then detecting whether multiple threads have performed such a propagation, and if so, performing compensating depart operations to ensure the invariant is maintained. These algorithms are subtle and require careful correctness proofs.

As in Section 4, by using hardware transactions we can avoid

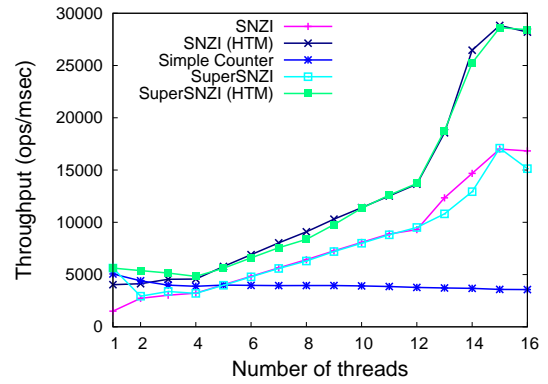


Figure 4: Throughput results for SNZI algorithm.

the “mistakes” due to concurrency, and thus avoid the need for complicated code to detect and compensate for them. In particular, if two threads concurrently attempt to propagate a surplus to the same parent node using hardware transactions, these transactions will conflict and will not be able to both commit successfully. If one succeeds, the other will retry and find that it no longer needs to propagate its contribution to the surplus up the tree, allowing it to complete its operation fairly quickly.

Interestingly, because the writes performed by a hardware transaction are not visible to others until it commits, our HTM-based SNZI algorithm can update each node’s counter *before* propagating its surplus to the node’s parent (if needed). This allows a simple iterative description of the algorithm, whereas previous algorithms are expressed recursively, requiring careful reasoning about the order of recursive execution.

We compare the performance of our HTM-based SNZI algorithms to the previous CAS-based implementations, as well as to a simple counter-based implementation. For both the HTM and CAS-based SNZI algorithms, we include *SuperSNZI* [15] versions that adapt to the level of contention by switching between direct arrivals at the root node (when there is little contention) and tree arrivals that start at the tree leaves (when contention on the root node grows). To test scalability, each of a number of threads repeatedly arrives at and departs from a single SNZI object without doing any work in between the operations. An additional thread repeatedly queries the SNZI once per microsecond. Each data point is the geometric mean of the throughputs of 10 runs.

Figure 4 presents the performance results for each of the algorithms. At low thread counts, the HTM-based *SuperSNZI* version performs slightly better than the non-*SuperSNZI* HTM-based variant, both of which perform comparably to the simple counter-based and non-HTM *SuperSNZI* algorithms at one thread. At higher thread counts, the HTM-based algorithms perform comparably to one another, outperforming the non-HTM-based algorithms by a wide margin—at 16 threads, they nearly double the throughput of the non-HTM SNZI algorithms and achieve nearly eight times the throughput of the simple counter-based algorithm.

Interestingly, the non-*SuperSNZI* HTM variant remains competitive at low thread counts—performing better than the simple counter-based solution at two threads and higher—despite the fact that the HTM transactions often have to update multiple SNZI nodes at low contention levels. This is in contrast to the non-HTM SNZI versions, where the more complex *SuperSNZI* algorithm is required in order to remain competitive with the simple counter at low thread counts. Without it, the basic CAS-based SNZI algorithm performs

Implementation	T=1		T=2		T=4		T=8		T=16		T=256	
libc	1019	1920K	723	1928K	624	1944K	619	2040K	624	2104K	622	5560K
libumem	1153	2336K	2289	2408K	4705	2616K	9026	2712K	15381	3416K	12433	13080K
Hoard	1154	3048K	2285	3504K	4419	4416K	8914	6240K	16319	9952K	10520	63648K
CLFMalloc	1980	3080K	4178	3016K	8490	3032K	16736	11256K	31948	19528K	31134	69576K
LFMalloc/RCS	9767	2344K	19221	2352K	36767	2432K	72667	2656K	128660	3040K	126191	7904K
LFMalloc/HTM	3988	2360K	8020	2368K	16167	2448K	32147	2736K	63055	3056K	59881	7280K
LFMalloc/TLE	4177	2360K	8438	2368K	16775	2512K	33445	2736K	66178	3056K	65253	7920K
LFMalloc/mutex	1502	2360K	2961	2368K	5927	2448K	11641	2736K	20847	3056K	20709	7856K
LFMalloc/spin	4582	2360K	9112	2368K	18152	2448K	36154	2672K	69527	3056K	29749	7920K

**Table 1: Malloc results, showing throughput in malloc-free pairs per millisecond and memory footprint in KB.**

more than three times worse than the simple counter-based algorithm when run with one thread, largely due to the multiple CAS operations required to make changes to multiple nodes in the tree.

Note that, with 16 timed threads, one shares a core with the query thread, impeding scalability. Also, the improvement for all algorithms at 12 threads is because, after 12 threads, the nodes in middle layer of the tree have non-zero surpluses most of the time. Thus, more Arrive operations can complete without modifying the root node, resulting in shorter Arrive operations *and* less contention.

We have also implemented the *resettable* SNZI-R variant [15, 25] using HTM, with similar conclusions.

The HTM implementations used in these experiments include a software backup that can be used if transactions fail repeatedly, and thus these algorithms can be used with purely best-effort HTM that makes no guarantees about committing small transactions. For the Arrive operation, the software backup arrives directly at the root with a CAS operation (this is correct without changes to the HTM transactions thanks to Rock’s strong atomicity guarantee). In the case of a Depart operation that fails to complete using a hardware transaction, the software backup uses the same Depart code used by the CAS-based SNZI algorithm, starting from the node at which the previous Arrive operation started.

This algorithm is still considerably simpler than the CAS-based SNZI algorithm as compensating undo operations are not needed, as discussed above. Moreover, the transactions used are small enough that the software backup was never used during our experiments (our SNZI trees are three levels deep, which has been sufficient to achieve scalable results on some of the largest multi-core systems available [25, 26]). Thus, given an HTM with guarantees for such transactions, similar results could be achieved with an even simpler algorithm that does not include the software backup.

## 6. MEMORY ALLOCATION

In this section, we use HTM to simplify LFMalloc, a fast and scalable memory allocator due to Dice and Garthwaite [9]. The key idea behind LFMalloc is to maintain per-processor data structures to alleviate the poor scalability of central data structures such as those used by libc’s allocator, while avoiding the excessive memory use of per-thread data structures. Per-processor data structures ensure that synchronization conflicts occur only due to scheduling events such as preemption and migration. LFMalloc exploits this observation using lightweight Restartable Critical Section (RCS) synchronization implemented using special Solaris™ scheduling hooks. The result is excellent performance and a reasonable memory footprint. It has the significant disadvantage, however, of requiring a special kernel driver to implement RCS.

Using HTM to synchronize the per-processor data structures eliminates the need for a special kernel driver. (Interestingly, for convenience, Dice and Garthwaite built a transactional interface imple-

mented using RCS.) LFMalloc is described in detail in [9]; here we describe changes we made, and compare the performance of various implementations.

We use the `mmicro` benchmark [9] to compare malloc implementations. Each thread repeatedly allocates 64 200-byte blocks and then frees them in the same order. In Table 1, we report throughput in malloc/free pairs per millisecond and total memory footprint in KB. As expected, libc’s malloc consistently has the smallest memory footprint, but provides low single-thread throughput, becoming even worse as the number of threads increases. libumem and Hoard [2] improve on its single-thread performance and have much better scalability, but also have significantly larger memory footprints. Consistent with previous results [9], unmodified LFMalloc consistently provides dramatically better throughput than any of the previous implementations, and its memory footprint is much lower than that of libumem or Hoard, though somewhat higher than that of the libc allocator. For comparison we also include Michael’s Lock-free CLFMalloc [32] allocator, which consistently provides higher throughput than other previous allocators.

The high single-thread performance of LFMalloc is due to the lack of any synchronization on local heaps, except when a thread is preempted or migrated during an allocation. LFMalloc also scales better than any of the previous implementations due to its use of per-processor data structures. We have also rerun other benchmarks described in [9], and overall the performance and scalability of LFMalloc is substantially better than previous allocators. However, its use of a special kernel driver is a barrier to adoption.

By replacing the RCS blocks in LFMalloc with hardware transactions, we obviate the need to implement and install a special kernel driver. As shown in Table 1, the resulting implementation has significantly higher overhead than the RCS-based LFMalloc (due to the latency of hardware transactions), but it still performs significantly better than the previous implementations, and maintains LFMalloc’s competitive memory footprint. Even so, without a guarantee that the HTM feature can always (eventually) commit the small and simple transactions used in this implementation, it would not be usable in practice (see [12] for a detailed discussion).

We achieved a more robust implementation that is usable even without guarantees for such transactions using a per-processor lock to protect the per-processor data structures, and then using TLE to elide the lock. As shown in Table 1, TLE performs slightly worse than HTM for one thread due to the additional overhead to examine the lock. However, TLE outperforms HTM slightly at higher threading levels because when a transaction fails, there is a quick way to make progress (namely acquiring the lock), whereas the HTM-only version may retry repeatedly. Furthermore, there is virtually no lock contention because conflicts occur only due to relatively rare events such as preemption and migration.

We also tried implementations that use *only* a lock, without TLE. As in Section 3, we tried both pthreads mutex locks and hand-coded

spinlocks. As before, the hand-coded spinlock provides significantly better performance than the more general pthreads lock.

It is tempting to conclude that the spinlock implementation should be used. However, all of the lock-based implementations can suffer when a thread is preempted while holding the lock, as evidenced by their poor performance with 256 threads. We note that, even though this *can* happen with the TLE variant, it is much less likely because the lock is held much less frequently. In contrast, if a hardware transaction is aborted when the thread executing it is preempted, no thread ever has to wait for another that is not running.

While preemption while holding a lock can be avoided by using the Solaris `schedctl` library call, our results show that HTM has the potential to provide an effective alternative approach. A robust HTM with low latency and guarantees for small simple transactions could deliver significantly better performance than the previous allocators tested, perhaps approaching the performance of the RCS-based `LFMalloc` implementations, while avoiding long waiting periods due to locking, and not requiring a special kernel driver.

## 7. THE CANNEAL BENCHMARK

Next we discuss how HTM can be used to simplify the `canneal` benchmark of the PARSEC suite [4], which uses a simulated annealing algorithm to optimize the routing cost of a chip design. In each iteration, each thread examines two randomly chosen elements on the chip, and evaluates how swapping their locations would affect the routing cost. The locations are swapped if the cost would decrease, and with some probability even if it would increase, to allow escaping from local optima; this probability is inversely proportional to the increase in the routing cost, but is also decreased during runtime to allow convergence.

The original implementation [4] uses an aggressive synchronization scheme that performs swaps atomically, but accesses location information without holding locks when deciding whether to swap. As a result, the locations of the elements examined when the gain or loss from a potential swap is evaluated may change during the evaluation. Such “races” can cause an inaccurate evaluation of the swap gain/loss value. However, the simulated annealing algorithm should naturally recover over time from any resulting mistakes [4].

While the benchmark is mostly straightforward, the code to swap two elements atomically is more complex. To avoid deadlock, locations are first ordered. Then:

1. The location of the first element is “locked” by atomically replacing the pointer to its location with a special value using a CAS instruction. While the location is locked, it cannot be changed by any other swap operation; moreover, attempts to read the location spin until it is unlocked.
2. A second CAS atomically fetches the location of the second element, and replaces it with the original location of the first.
3. The location of the first element is unlocked and replaced with the original location of the second using a regular store instruction, as the location does not change once locked.

This approach also complicates read accesses, which must first check whether a location is locked.

### 7.1 Simplifying the Atomic Swap Operation

We replaced the above-described code for swapping two elements with a hardware transaction that simply executes the swap. This change allowed us to replace all read accesses to location information with simple loads. This is possible because Rock’s HTM feature provides *strong atomicity* [5], so that ordinary loads and stores can be used together with transactions.

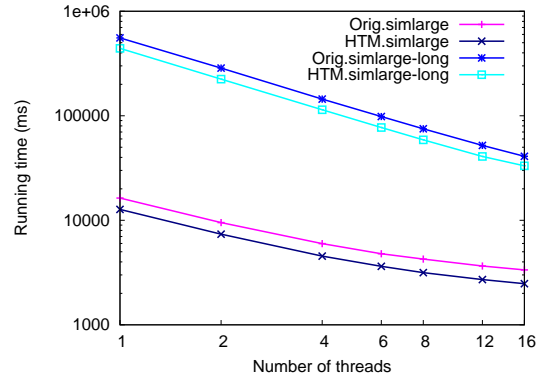


Figure 5: Execution time for the `canneal` benchmark.

Our modified implementation simply retries hardware transactions without backoff until they succeed; we expect very little contention between swap operations for any reasonable input. If the HTM feature does not make guarantees for the small transactions used by this algorithm, a software alternative would be necessary. Given that conflicts are rare, it seems that a simple TLE scheme should be effective. However, using TLE would again require additional overhead and at least some additional complexity for reading locations. This again illustrates the benefit of being able to rely on hardware transactions without providing a software alternative.

To evaluate our modified implementation on Rock, we initially used the PARSEC suite’s [4] `simlarge` configuration. We first observed that the original implementation did not scale very well. Bhaduria *et al.* [3] made similar observations, blaming workload imbalance and a large serial portion that caused one thread to execute more operations than all others. Investigating further, however, we found that by increasing the number of iterations executed, a routing with significantly lower cost is achieved, and the scalability of the algorithm improves substantially.

Figure 5 shows results for the original implementation vs. the simplified variant of `canneal`, denoted as Orig and HTM respectively, for two configurations: the first uses the original `simlarge` configuration, which evaluates 1,920,000 potential swaps, and the second (labeled with “`simlarge-long`” in Figure 5) uses the same input file, but evaluates 75,000,000 potential swaps. We measure the execution time for varying thread counts, using the configurations described above. The results are plotted on a log-log scale.

In a single-thread run, the simplified HTM-based implementation completes about 21% faster than the original for the short run, and about 25% faster for the long run. This gain is mostly due to reading locations using regular load instructions, which is enabled by the use of a hardware transactions to swap elements.

Both implementations achieved a speedup of about 5x using 16 threads on the short run, and over 12x on the long one. All runs of the same configuration achieved about the same routing cost, with the long configuration achieving a 28% cheaper routing.

These results show that the ability to execute a simple hardware transaction that swaps the value of two memory locations enables significant performance improvements *and* simpler code. We reiterate, however, that the simpler code is acceptable in practice only given sufficient guarantees that a software alternative for the transactions is not needed.

### 7.2 Further simplifications and improvements

Depending on the HTM feature used, additional simplifications and performance improvements are possible.

First, if *all* accesses to location information were performed inside hardware transactions, we could eliminate the level of indirection for location information by storing and accessing it “in place”. (This level of indirection could possibly be eliminated anyway by encoding location information in a single word, but such constraints do not apply if the data is accessed in hardware transactions.) Whether this would result in a performance benefit depends crucially on the latency of a small transaction that is read-only, or at least does not modify any shared data or encounter any conflicts. This points out the potential benefit of optimizing certain classes of small and simple transactions.

Second, if we could use a single hardware transaction to decide whether to perform a swap and perform it if so, this would eliminate the races discussed above, obviating the need to reason about whether they affect correctness and convergence.

However, this more ambitious simplification places significantly stronger requirements on the HTM feature. In particular, the computation to decide whether to swap two elements examines their neighbors; thus the number of locations accessed is not bounded by a constant in the algorithm. Furthermore, the transaction would make nested function calls (e.g., for the `exp()` library function). On Rock, it is not reasonable to rely on hardware transactions alone for such transactions (see [11, 12]). Making this reasonable would require that the HTM guarantee to (eventually) commit transactions that perform a number of data-dependent loads that is not bounded by any constant in the algorithm (the number of neighbors depends on the input data), and that make nested function calls.

## 8. DISCUSSION

The examples we have presented—and others we have omitted due to time and space constraints—make a compelling case that HTM has strong potential to simplify the development of concurrent algorithms that are scalable, efficient, and correct. However, in many cases this potential depends critically on certain properties of the HTM feature used.

An important property required by many of our examples is the ability to rely on a small and simple hardware transaction to eventually commit. Without such a guarantee, most of our examples require a software alternative to hardware transactions. Apart from adding significant complexity in most cases, the need to work correctly with such software alternatives can add significant overhead to common-case code, regardless of how infrequently the alternative is actually needed. For example, using TLE to provide a software alternative for swap transactions in `canneal` requires reads to check the lock, adding significant overhead.

An important question is how such guarantees can be stated, and how programmers can determine whether their code meets the criteria for the guarantee. This can be difficult, given the range of possible scenarios, however unlikely they may be. The requirement to provide such guarantees can stifle important innovation and optimizations. This tension may be alleviated by a last-resort fallback mechanism, such as parking all other threads, aborting their transactions, and executing a transaction alone if it fails to complete. While this can make designs more flexible and criteria for guarantees easier to state, dependence on such mechanisms in any but rare circumstances may have disastrous performance consequences.

We also note that some of our algorithms depend on strong atomicity [5]. Examples include the simple load for read accesses in `canneal`, and the optimized local operations in our work-stealing algorithms. If strong atomicity were not provided, correct algorithms could be achieved in many cases by replacing simple load and store instructions with very small transactions. However, the performance benefit of using the nontransactional instructions

would be lost unless transaction latency were very low, at least for such simple transactions. We have observed several examples in which HTM could be used profitably if common-case latency for certain classes of transactions were very low. Classes to consider include: read-only transactions, transactions that do not write shared data, or do not conflict on shared data, transactions that access only a single (shared) memory location, etc.

Rock’s simple “requester-wins” conflict resolution policy requires the use of backoff in situations of heavy contention. While backoff is simple and can be hidden in library code, it has the disadvantage that it may waste time by backing off too much or not enough if not properly tuned to the workload. Designers should therefore consider more sophisticated conflict resolution policies to reduce the number of aborts and the reliance on backoff mechanisms.

The original motivation for HTM [21] was to make it easier to build nonblocking data structures. The question arises, therefore, of whether a given HTM feature is actually useful for this purpose. Nonblocking progress guarantees typically forbid waiting only *in software*. For example, waiting for a response to a request for a cache line is not usually considered to violate nonblocking progress conditions. However, if an HTM implementation allows a running transaction to wait for a preempted one, it cannot claim to support nonblocking data structures. For example, it would not enable sharing between an interrupt handler and the interrupted thread [35].

It seems easy to avoid such problems by aborting any transaction being executed by a thread when it is preempted or suffers another long-latency event such as a page fault. However, there is a tension between such approaches and the desire to provide guarantees that certain classes of transactions can eventually be completed. Such guarantees may need to be stated in terms of the length of transactions relative to the frequency of disruptive events such as interrupts. Balancing these concerns is a challenge for designers who hope to achieve all of the potential benefits of HTM.

We have also noted the value of Rock’s “sandboxing” property, namely that it is not possible to cause bad events such as program crashes inside a transaction. This property allows simpler and faster code to be used in many cases, for example because consistency checks that are normally required for correctness can be elided in the common case. However, we must emphasize the importance of good feedback about the reasons for transaction failure, both to allow appropriate responses and to facilitate debugging and analysis. As reported previously [11, 12], Rock’s feedback about aborted transactions can be difficult to interpret in some cases, and its support for debugging of code inside hardware transactions is very limited. Future HTM features should improve on both aspects.

## 9. CONCLUDING REMARKS

We have presented several examples demonstrating the potential power of hardware transactional memory (HTM) to enable the development of concurrent algorithms that are simpler than nontransactional counterparts, perform better than them, or both. We have also highlighted the properties required of an HTM feature to enable these uses, and summarized these observations with the hope of assisting designers of future HTM features to enable maximal benefit from HTM.

## References

- [1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proc. 10th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.
- [2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In

- Proc. ninth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, New York, NY, USA, 2000. ACM.
- [3] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proc. International Symposium on Workload Characterization*, October 2009.
- [4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [5] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing Transactions: The Subtleties of Atomicity. In *the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.
- [6] D. Chase and Y. Lev. Dynamic Circular Work-Stealing Deque. In *Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, 2005.
- [7] 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.
- [8] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. N. Shavit, and G. L. Steele Jr. Even better DCAS-based concurrent deques. In *Proc. 14th International Conference on Distributed Computing*, pages 59–73. IEEE, 2000.  
<http://citeseer.nj.nec.com/detlefs00even.html>.
- [9] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proc. 3rd International Symposium on Memory Management*, pages 163–174, New York, NY, USA, 2002. ACM.
- [10] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. Transact 2008 workshop.  
<http://research.sun.com/scalable/pubs/TRANSACT2008-ATMTP-Apps.pdf>.
- [11] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. 14th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, New York, NY, USA, 2009. ACM.
- [12] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009.
- [13] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*, 2006.
- [14] S. Doherty and M. Moir. Nonblocking algorithms and backwards simulation. In *Proc. 21st International Conference on Distributed Computing*, 2009.
- [15] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proc. 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 13–22, 2007.
- [16] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.
- [17] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
- [18] M. Herlihy. Personal communication, 2010. See “Sadistic homework problem” in various presentations.
- [19] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.
- [20] 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.
- [21] 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, 1993.
- [22] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [23] JSR166: Concurrency Utilities.  
<http://gee.cs.oswego.edu/dl/concurrency-interest/>.
- [24] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proc. 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 227–242, New York, NY, USA, 2009. ACM.
- [25] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009. <http://research.sun.com/scalable/pubs/TRANSACT2009-ScalableSTMANatomy.pdf>.
- [26] Y. Lev, V. Luchangco, and M. Olszewski. Scalable Reader-Writer Locks. In *Proc. 21st Annual Symposium on Parallelism in Algorithms and Architectures*, pages 101–110, 2009.
- [27] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. The Workshop on Transactional Computing, Aug. 2007.  
<http://research.sun.com/scalable/pubs/TRANSACT2007-PhTM.pdf>.
- [28] P. McKenney. Personal communication, 2010.
- [29] P. E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2010.  
<http://www.rdrop.com/users/paulmck/perfbook/perfbook.2010.01.23a.pdf> [Viewed January 24, 2010].
- [30] M. Michael. Cas-based lock-free algorithm for shared deques. In *Proc. Ninth Euro-Par Conference on Parallel Processing*, pages 651–660, 2003.
- [31] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.
- [32] M. M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, 2004.
- [33] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–54, New York, NY, USA, 2009. ACM.
- [34] 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.
- [35] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/txLinux: Transactional memory for an operating system. In *Proc. 34th Annual International Symposium on Computer Architecture*, 2007.
- [36] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing hardware transactional memory in the operating system. In *Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, 2007.
- [37] The SPARC Architecture Manual Version 8, 1991.  
<http://www.sparc.org/standards/v8.pdf>.