Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory *

Luke Dalessandro[1] François Carouge[2] Sean White[2]
Yossi Lev[3] Mark Moir[3] Michael L. Scott[1] Michael F. Spear[2]

[1]University of Rochester
{luked, scott}@cs.rochester.edu

[2]Lehigh University
{frc309, saw207, spear}@cse.lehigh.edu

[3]Oracle Labs
{yossi.lev, mark.moir}@oracle.com

Abstract

Transactional memory (TM) is a promising synchronization mechanism for the next generation of multicore processors. Best-effort Hardware Transactional Memory (HTM) designs, such as Sun's prototype Rock processor and AMD's proposed Advanced Synchronization Facility (ASF), can efficiently execute many transactions, but abort in some cases due to various limitations. *Hybrid TM* systems can use a compatible software TM (STM) in such cases.

We introduce a family of hybrid TMs built using the recent NOrec STM algorithm that, unlike existing hybrid approaches, provide *both* low overhead on hardware transactions *and* concurrent execution of hardware and software transactions. We evaluate implementations for Rock and ASF, exploring how the differing HTM designs affect optimization choices. Our investigation yields valuable input for designers of future best-effort HTMs.

Categories and Subject Descriptors C.1.4 [*Computer System Organization*]: Parallel Architectures; D.1.3 [*Software*]: Concurrent Programming

General Terms Algorithms, Design

Keywords Transactional Memory

1. Introduction

As the industry shifts to pervasive multicore computing, considerable effort is being invested in exploring hardware support for concurrency and parallelism. Transactional Memory (TM) [25] is a leading example of such support. It allows the programmer to mark regions of code that should be executed atomically, without requiring them to specify *how* that atomicity is achieved. Typical implementations are based on speculation. While proposals exist for unbounded and fully virtualized hardware TM (HTM) [24], concrete industry proposals are currently focused on *best-effort* HTM that imposes limits on hardware transactions. Sun's prototype multicore SPARC™ processor code named Rock [8], and AMD's proposed Advanced Synchronization Facility (ASF) extension to the x86 architecture [3] are two prominent examples. The limitations of such systems can be hidden from programmers using a *hybrid* approach, in which logical transactions can be executed using HTM, but resort to a software TM (STM) alternative when necessary.

Early hybrid TM algorithms [11, 30] support *concurrent* execution of hardware and software transactions, achieved by instrumenting a hardware transaction's memory operations with accesses to STM metadata, allowing it to detect conflicts with software transactions. This approach enables logical transactions to switch to software on a per-transaction basis but imposes significant overhead on hardware transactions and constrains the STM implementation. This results in systems with both high overheads and limited scalability.

Lev et al.'s PhTM hybrid significantly reduces the impact on hardware transactions by establishing global *phases* of execution [29]. As a result hardware transactions need instrumentation only at the boundaries of transactions, to detect global phase changes. PhTM performs significantly better than prior options when hardware transactions are mostly successful, and allows more flexibility for the STM implementation, which is no longer constrained by the potential for concurrent hardware transactions. However, PhTM's phase changes amount to application-wide synchronization barriers that result in poor performance for workloads in which switching becomes frequent. In addition, tuning of PhTM's phase change protocol is complicated and prone to overfitting.

Our goal is to develop a family of novel hybrid TM algorithms that combine the best features of both HyTM and PhTM: support for concurrent hardware and software transactions without the overhead of per-access instrumentation for hardware transactions. We achieve this by using Dalessandro et al.'s NOrec STM [10] as the software component. Section 2 presents more detailed background about best-effort HTM and previous hybrid TM algorithms. Section 3 expands a two-paragraph hybrid TM sketch appearing in Dalessandro et al. [10] into a simple but full-fledged hybrid design and then discusses a variety of novel, hardware-independent refinements that address its obvious weaknesses. Sections 4 and 5 develop platform-specific implementations, optimizations, and evaluations for Rock and ASF respectively.

While direct performance comparisons between Rock hardware and the ASF simulator are not meaningful, differences between their HTM implementations result in different optimization choices. Section 6 elaborates on these differences, and highlights some of the advantages and disadvantages of each platform in supporting a NOrec-based hybrid system—a contribution that we believe provides useful guidance to designers of future HTM. Section 7 gives an overview of related work and Section 8 concludes.

2. Background

Comprehensive coverage of TM proposals and implementations is available in a recent book by Harris et al. [24]. Here we focus on issues of particular relevance to our work.

* At the University of Rochester, this work was supported by NSF grants CNS-0615139, CCF-0702505, CSR-0720796, and CCR-0963759. At Lehigh University, this work was supported by NSF grant CNS-1016828.

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.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

Copyright © 2011 ACM 978-1-4503-0266-1/11/03. . . $10.00

2.1 Hardware TM

Herlihy and Moss [25] proposed TM as a hardware mechanism to support the development of non-blocking data structures. In this proposal, and in many others that followed it, the atomicity of transactions is enforced by using the cache coherence protocol.

To avoid revealing speculative state to other processors, transactional stores are typically buffered locally until the transaction completes. Herlihy and Moss use a special transactional cache, which bounds the size of transactions, for this purpose.

To ensure that successful transactions appear to occur atomically, a processor attempts to acquire exclusive ownership of all cache lines modified by its transaction. If successful, it can then decide locally to commit the transaction, at which point requests for any cache lines modified by the transaction will receive the data written by the transaction; partial effects are never visible.

HTM implementations must also ensure that reads are consistent and current at commit time. Again, many proposals use cache coherence to achieve this: if the processor retains (at least shared) ownership of every cache line read by the transaction up to the point of its commit, then each of the locations read is unchanged and the transaction is consistent at the commit. Typically the transaction is aborted if such a cache line is invalidated or evicted.

A key issue is how a processor responds to a request for a cache line it holds transactionally. The *requestor-wins* policy provides the requested line and aborts the local transaction. While simple, this policy can lead to poor performance and livelock: competing transactions may abort each other repeatedly. Backoff is effective in avoiding this effect, but can be difficult to tune. More sophisticated conflict resolution mechanisms are appealing but entail more implementation complexity.

Some HTM implementations, such as the β-TVP proposal of Tabba et al. [40], can reacquire ownership of a lost cache line, check that any values read from it previously are current, and thus avoid aborting the transaction. Similar *value based validation* is found in some STM systems, including the NOrec algorithm.

As described, HTM implementations cannot commit transactions that exceed the size or geometry of the hardware structures used to track the cache lines accessed by a transaction and to buffer speculative stores. Proposals for *unbounded* HTM [4, 5, 21, 33, 35] overcome such limitations, but introduce significant complexity; commercial systems built or proposed to date do not attempt to implement unbounded transactions.

Without unbounded transactions, an STM alternative is needed to support transactions that exceed hardware limits. Given this, it is reasonable for hardware transactions to fail occasionally for other reasons, such as interrupts, traps, etc. The option of an occasional spurious failure simplifies hardware design, because difficult corner cases can be resolved simply by aborting the transaction.

This design strategy is known as *best-effort* HTM: the hardware attempts to support as many transactions as possible given the constraints of the implementation, but can abort any transaction for any reason. It is important for software to receive good feedback about the reason for a particular abort, to enable sensible decisions about whether and when to retry a hardware transaction.

Sun's prototype Rock processor and AMD's ASF proposal are concrete examples of best-effort HTMs. While their designs are similar in some ways, they have different interfaces, capabilities, and reasons for aborting transactions. For example, Rock uses a core's store buffer to hold transactional stores, imposing a limit of 32 stores per transaction. The ASF specification simply states that conforming implementations will support (potentially repeated or redundant) stores to at least 4 elements. We examine the design decisions of Rock and ASF in more detail as we develop hybrid TM implementations for them in Sections 4 and 5 respectively.

```
1   padded orec orecs []

3   HW_READ(address)
4     value = *address
5     if (orecs[hash(address)].mode != WRITE)
6       return value
7     abort

9   HW_WRITE(address, value)
10    if (orecs[hash(address)].mode != UNOWNED)
11      abort
12    *address = value
```

Figure 1. Pseudocode illustrating HyTM instrumentation for hardware, from Damron et al. [11]. This code assumes the STM counterpart uses visible (or semivisible) readers. We use **padded** to indicate variables that are padded to avoid false sharing.

2.2 Hybrid TM

We use the term *hybrid TM* for systems that can execute a logical transaction in hardware, or use a compatible STM implementation when this is not successful. Hybrid systems [11, 30] do not require that the hardware be able to commit any particular transaction. Some other hybrid-like systems [10, 26] depend on the HTM to be able to commit at least some small transactions. Next we briefly describe two hybrid TM systems most closely related to this paper.

HyTM[1] [11] generates two code paths for each source-level transaction—a software path targeting an ownership record (*orec*)-based STM, and a hardware path in which memory accesses are instrumented to perform appropriate checks of the STM metadata.

A software transaction modifies the mode of each orec covering a location it accesses, and hardware transactions access this information on each memory access in order to detect conflicts with software transactions. The instrumentation for hardware-path accesses is shown in Figure 1. The instrumentation ensures that a hardware transaction detects a conflict with a software one, either by observing that the mode of an orec indicates a conflict, or by aborting when a software transaction modifies the mode after the hardware transaction has read it. The overhead imposed on hardware transactions by this per-access instrumentation is considerable, including both additional instructions and the use of scarce, best-effort HTM resources to track per-location metadata (i.e., orecs).

PhTM [29] introduces global phases of execution, typically hardware-only and software-only. It employs simple instrumentation at transaction start and finish to enforce the phase discipline, and eliminates the costly per-access instrumentation of other hybrid systems. Because hardware and software transactions do not run concurrently, PhTM also allows more flexibility in implementing the STM used in software phases. However, if transactions must frequently be executed in software, phase changes incur system-wide synchronization barriers that severely impact performance and scalability. In addition, the algorithms used to trigger phase changes are difficult to tune and susceptible to overfitting. For workloads in which most transactions can succeed in hardware, the software-only phase may be implemented as a single, serial transaction running with minimal software instrumentation [29]. This design has been shown to lower the cost of phase synchronization barriers, at the cost of significantly less concurrency [9].

The emergence of STMs that do not employ per-location metadata (e.g., the NOrec STM [10] described in Section 3) provides an opportunity to reevaluate HyTM's defining characteristic—the con-

[1] We use *HyTM* to both refer to Damron et al. [11], and to describe algorithms in which hardware and software transactions execute concurrently.

```
1  padded unsigned seqlock

3  thread local unsigned snapshot
4  thread local ReadSet reads
5  thread local WriteSet writes

6  SW_VALIDATE
7    snapshot = seqlock
8    if (snapshot & 1)
9      goto 7
10   foreach (addr, val) in reads
11     if (*addr != val)
12       SW_ABORT
13   if (snapshot != seqlock)
14     goto 7
```

```
15 SW_BEGIN
16   snapshot = seqlock
17   if (snapshot & 1)
18     goto 16

19 SW_COMMIT
20   if (writes.empty())
21     return
22   while (!CAS(&seqlock, snapshot,
         snapshot + 1))
23     SW_VALIDATE
24   foreach (addr, val) in writes
25     *addr = val
26   seqlock = seqlock + 1
27   reads.reset(), writes.reset()
```

```
28 SW_READ(addr)
29   if (addr in writes)
30     return writes.find(addr)
31   val = *addr
32   if (snapshot != seqlock)
33     SW_VALIDATE
34     goto 31
35   reads.append(addr, val)
36   return val

38 SW_WRITE(addr, val)
39   writes.append(addr, val)

41 SW_ABORT
42   reads.reset(), writes.reset()
43   /* restart transaction */
```

Figure 2. The basic NOrec algorithm. The CAS on line 22 represents atomic compare-and-swap. Read and write sets are implemented at byte granularity and thus compatible with the draft C++ TM specification [1].

current execution of hardware and software transactions—without per-access instrumentation overhead on hardware transactions. That is the goal of the work described herein.

3. NOrec Hybrid Design

Our family of hybrid TM algorithms begins with Dalessandro et al. NOrec STM (Figure 2) [10]. NOrec does not require fine-grained, shared metadata, merely a single global sequence lock [27] (seqlock) for concurrency control. Transactions buffer transactional writes and log read address/value pairs in thread local structures. Committing writers increment the seqlock before writeback. Active transactions poll the global seqlock for changes indicating concurrent writer commits. A new value is evidence of possible inconsistency and triggers validation, done in a value-based style by comparing the logged address/value pairs to the actual values in main memory [17, 22, 34, 40].

An odd seqlock value indicates that a transaction is performing writeback; it effectively locks the entire shared heap. A consequence of this protocol is that only a single writer can commit and perform writeback at a time. This sequential bottleneck is minimized by validation *prior* to acquiring the seqlock for commit (Figure 2, Line 22). Successfully incrementing seqlock transitions a writer to a committed state in which it immediately performs its writeback and then releases the lock. A read-only transaction need only confirm that its reads were consistent and can commit without modifying seqlock.

NOrec scales well when its single-writer commit serialization does not represent the overall application bottleneck, i.e., writeback does not dominate the run time, and has been shown to have low latency. NOrec supports the *privatization* idiom [24] as required for compatibility with the C++ draft TM standard [1] and provides strong *publication* guarantees [32].

For the purposes of initial development, we assume that the best-effort HTM is strongly atomic [6], and that we can issue *nontransactional loads* within the context of a transaction—i.e., that we can load a location without adding it to our transactional read set. This second capability is not fundamental, but is useful for some optimizations and is supported by both Rock and ASF. In Sections 4 and 5 we discuss the implementation of variants of these algorithms on Rock and ASF, respectively.

In a NOrec-based hybrid TM, hardware transactions must respect the integrity of NOrec's single-writer commit protocol. Accordingly, hardware transactions that modify program data cannot commit during software writeback, and must signal their commit so as to trigger validation by concurrent software transactions. These two constraints neither impose requirements on hardware transactions in the absence of active software transactions, nor restrict the behavior of read-only transactions, however one further detail must be addressed in the hybrid TM design; hardware transactions may become *logically inconsistent* due to a concurrent software transaction's non-atomic writeback.

```
transaction {              transaction {
  x = 1                      if (x != y)
  y = 1                        STATEMENT
}                          }
```

(a) Static transactions.

Initially: x == y == 0

```
1  SW_WRITEBACK            HW_POST_BEGIN
2    x = 1
3                            r1 = x
4                            r2 = y
5                            if (r1 != r2)
6                              STATEMENT
7    y = 1
```

(b) Dynamic execution trace.

Figure 3. Execution of two transactions resulting in a hardware transaction observing inconsistent state.

Consider the execution in Figure 3. The logical transaction in the left column executes and commits as a software transaction, while the logical transaction on the right executes in hardware. The dynamic trace shows one possible interleaving of operations from the software transaction's writeback with the hardware transaction's execution.

The hardware transaction performs an inconsistent read at Line 4—its read of x on Line 3 returns the value written by the software transaction, while the read of y returns the value written *before* the software transaction occurred. The strongly atomic hardware transaction will ultimately detect this inconsistency when the software transaction performs its update of y on Line 7, but can perform arbitrary actions in the meantime.

The danger here is that the inconsistent transaction may perform an action that has visible effects (e.g., a nontransactional store) that is not consistent with the semantics of the program. This situation is nearly identical to that of *zombie* transactions in STM [14, 38, 41], where inconsistent software transactions may perform most of the same dangerous operations that an inconsistent hardware transaction could. The difference is that a zombie transaction cannot com-

```
1  padded unsigned seqlock



5  HW_POST_BEGIN
6    if (seqlock & 1)
7      while (true) // await abort

9  HW_PRE_COMMIT
10   seqlock = seqlock + 2
```

(a) Our naive algorithm

```
1  padded unsigned seqlock
2  padded unsigned counter


5  HW_POST_BEGIN
6    if (seqlock & 1)
7      while (true) // await abort

9  HW_PRE_COMMIT
10   counter = counter + 1
```

(b) 2-Location

```
1  padded unsigned seqlock
2  padded unsigned counter[]
3  thread local unsigned id

5  HW_POST_BEGIN
6    if (seqlock & 1)
7      while (true) // await abort

9  HW_PRE_COMMIT
10   counter[id] = counter[id] + 1
```

(c) P-Counter

Figure 4. Instrumentation of hardware transactions in our basic hybrid NOrec algorithms. Line 7 spins until the committing software transaction completes writeback and releases the seqlock *at which point the hardware transaction aborts and restarts*. 2-Location adds a counter location that only hardware transactions modify, P-Counter distributes this counter to reduce cache contention. The HW_* instrumentation occurs *within* the scope of a hardware transaction, and all accesses are transactional.

mit from an inconsistent state, while a hardware transaction could if not prevented. We use the term zombie to refer to inconsistent logical transactions with the understanding that they are slightly different in hardware and in software.

In STM one of two solutions is used to avoid incorrect behavior due to inconsistent execution. An implementation may eliminate zombies by ensuring that each operation performed by a transaction is done from a consistent state, a property known as *opacity* and detailed by Guerraoui and Kapalka [20]. Alternatively, an implementation may admit the possibility of zombie execution and include appropriate preventative and recovery code, a process known as *sandboxing* and employed by Harris et al. [23] and Saha et al. [37]. Each of these approaches is possible in hybrid TM designs. Our initial algorithms are opaque (satisfy opacity); we discuss sandboxing hardware transactions in Section 3.2.

3.1 Basic Algorithms

The code in Figure 4a represents a naive integration of NOrec and a strongly atomic HTM. We instrument hardware transactions with a HW_POST_BEGIN barrier in which they read NOrec's shared seqlock. This includes seqlock in their read sets, effectively *subscribing* hardware transactions to software commit notifications: an abort is triggered when a software transaction acquires the lock to commit. If a hardware transaction starts and detects that a software transaction already owns the seqlock it spins until the software transaction completes writeback and releases the lock. Hardware transactions update seqlock in a HW_PRE_COMMIT barrier, *signaling* their commits to software transactions.

This implementation trivially satisfies the two constraints mentioned above. The HW_POST_BEGIN seqlock read and corresponding branch ensure both that no hardware transaction can commit during software writeback, and that the overall implementation is opaque: no transactional load can occur while the seqlock is held by software for writeback, thus logical zombies are impossible. The increment and write of the seqlock in HW_PRE_COMMIT notifies active software transactions of the need to validate their consistency.

Unfortunately, each hardware transaction conflicts with every software transaction (due to the early read of seqlock in HW_POST_BEGIN), and will abort when any software transaction commits, regardless of actual data conflicts. Similarly, each hardware transaction conflicts with every other hardware transaction (due to the early read of seqlock and the write of seqlock in HW_PRE_COMMIT), and will abort when any hardware transaction commits. These conflicts exist for each transaction's full duration, eliminating concurrency.

2-Location Figure 4a uses the NOrec global seqlock for bidirectional communication: hardware transactions read it to subscribe to software commit notification and increment it to signal their commits to software transactions. Dalessandro et al. observe that a simple modification can address this shortcoming: a second shared location, counter, decouples subscribing from signaling.

As shown in Figure 4b,[2] hardware transactions still subscribe to software commits by reading seqlock in HW_POST_BEGIN, but they use counter to signal their own commits. This reduces the window of vulnerability to metadata conflicts between hardware transactions from the entire hardware transaction to the time that it takes to execute the increment in HW_PRE_COMMIT and commit.

Where software transactions previously polled only seqlock (Figure 2, Line 32) they must now maintain snapshots for, and poll, both seqlock and counter. NOrec's SW_COMMIT must also now perform additional validation after acquiring seqlock, in order to detect any hardware commit(s) that occurred before the CAS (Figure 2, Line 22). This validation consists of reading the hardware counter, and performing value-based validation if it has changed. This extra validation adds serial overhead to software transactions.

P-Counter In 2-Location, hardware transactions use the counter to signal their commits to software transactions, but *not* to coordinate with other hardware transactions. Thus, hardware-hardware conflicts during counter increments are artificial and can be reduced or eliminated by distributing counter. Figure 4c assigns each processor core its own hardware counter. Hardware transactions modify different counter variables in HW_PRE_COMMIT and thus no longer conflict with each other. Any number of counters and any mapping of threads to counters can be used; we explore some variants in Section 4.

Where the 2-Location modification required software transactions to poll two locations, P-Counter requires them to poll $n + 1$, where n is the number of counters. Thus, there is a tradeoff: using more counters reduces conflicts between hardware transactions, but increases overhead on software transactions—including the serial overhead in their SW_COMMIT barriers due to validation as described above. In practice, it makes sense to dynamically determine the number of counters and the thread-to-counter mapping based on current system conditions; this is future work. 2-Location can be seen as the degenerate case in which all threads share a single counter.

[2] This algorithm is slightly different than that presented in Dalessandro et al.. Their version requires progress guarantees for 2-location transactions, which we do not assume here.

```
1   HW_POST_BEGIN
2     return

4   HW_PRE_COMMIT
5     if (seqlock & 1) while (true);
6     ...

8   HW_VALIDATE
9     while (non_tx_load(seqlock) & 1)
10      ; // spin

12  HW_READ(address)
13    value = *address;
14    HW_VALIDATE;
15    return value;
```

Figure 5. Code to ensure opacity in a system with lazy subscription. Uses a nontransactional read in HW_VALIDATE. All other accesses are transactional.

```
1   padded unsigned sw_exists

3   SW_BEGIN
4     atomic_add(sw_exists,1)
5     ...

7   SW_COMMIT, SW_ABORT
8     atomic_add(sw_exists,−1)
9     ...

11  HW_PRE_COMMIT
12    if ( sw_exists == 0) return
13    ...
```

(a) SW-Exists

```
1   thread local bool readonly

3   HW_POST_BEGIN
4     non_tx_store(readonly, true)
5     ...

7   HW_POST_WRITE
8     non_tx_store(readonly, false
          )

10  HW_PRE_COMMIT
11    if (non_tx_load(readonly))
12      return
13    ...
```

(b) Read-Only

Figure 6. Filters that allow hardware transactions to avoid accesses to seqlock and counter in HW_PRE_COMMIT. The pseudo-code augments the underlying algorithm's metadata and instrumentation. HW_∗ instrumentation occurs within the scope of a hardware transaction.

3.2 Lazy Subscription and Sandboxing

In the HW_POST_BEGIN barriers of the hybrid algorithms described so far, a transactional read of the NOrec seqlock subscribes hardware transactions to software commit notification, and the corresponding branch stalls hardware transactions while software transactions write back. This both establishes opacity and prevents hardware writers from committing during software writeback.

The early read of the seqlock is conservative as it forces all hardware transactions to abort whenever any software transaction commits, regardless of true data conflicts. We can be more optimistic—and admit more concurrency—by delaying the transactional read of seqlock until commit time, as shown in Figure 5. In this manner, hardware transactions subscribe to software commit notification immediately prior to committing, increasing hardware-software concurrency by reducing the number of false aborts received by hardware transactions.

Delaying the read of seqlock sacrifices opacity, as nothing prevents a hardware transaction from reading inconsistent data during software writeback. The listing in Figure 5 re-establishes opacity by adding a HW_READ barrier that polls the seqlock nontransactionally and pauses if it is locked. This requires instrumentation on the hardware path. While per-access instrumentation in HyTM was shown to be a significant obstacle, this instrumentation neither performs writes nor uses scarce best-effort HTM resources, and is thus worth exploring.

As observed previously, sandboxing exists as an alternative to opacity. Rather than instrumenting all loads for consistency, we instrument only instructions whose effects may expose a transaction's inconsistency. Fortunately, sandboxing best-effort hardware transactions appears simpler than sandboxing STM [2, 23, 34, 37], as many of the behaviors that must be avoided (divide by zero, bus error, an infinitely looping thread reaching a context switch) cause hardware transactions in most best-effort HTM systems to abort without software intervention. Others can be avoided with software assistance by inserting a HW_VALIDATE call in front of any instruction that may be dangerous to perform from an inconsistent state (nontransactional store, indirect branch, etc.). The number of such instructions is likely to be significantly smaller than the number of reads.

3.3 Communication Filters

Recall that our constraints for the serializability of hardware and software transactions apply to writer transactions only in the presence of active software transactions. Our algorithms so far are thus overly conservative. We can "filter out" a significant number of cache misses and false aborts by avoiding competition for shared counters in the absence of software transactions.

The SW-Exists filter introduces a shared location, sw_exists, for use as an early exit test in the HW_PRE_COMMIT barrier. Software transactions make their presence visible by setting this location; hardware transactions may elide counter updates if sw_exists is not set. Figure 6a shows a simple atomic counter implementation of SW-Exists; a more scalable mechanism such as Ellen et al.'s [18] scalable non-zero indicator (SNZI) may be desirable depending on best-effort HTM characteristics. The atomic counter implementation causes a hardware transaction that has read it to abort on every change, and induces cache contention among the software transactions seeking to modify it; SNZI limits aborts to the important cases—the transitions between the zero and non-zero states—while reducing software contention.

Some transactions can be statically shown by the compiler to be read-only and in this case HW_PRE_COMMIT can be omitted in its entirety. Alternatively, a transaction may dynamically track its read-only status using a HW_WRITE barrier. Figure 6b presents instrumentation to update a thread-local flag after writing, and then test it at commit time. Note that the combined use of lazy subscription with sandboxing and the Read-Only filter requires at least a HW_VALIDATE barrier before committing because without it, the readonly flag may be based on inconsistent execution.

4. Rock

Sun's prototype multicore processor code named Rock uses aggressive checkpointing and speculative execution to improve single-threaded performance [8], and uses the same mechanisms to provide HTM via new *checkpoint* and *commit* instructions. Rock's checkpointing mechanism restores all registers when a transaction aborts. Memory accesses within a transaction are transactional unless explicitly marked nontransactional.

Rock augments L1 cache lines to track speculative reads, and holds speculative writes in the store buffer until commit or abort. Thus transactions can read as many lines as fit in the L1 data cache

(limited by associativity), and can write as many locations as fit in the store buffer (32 in the configuration we used). Nontransactional stores are held in the store buffer as well; they cannot be used to circumvent its limited capacity. Some instructions cause aborts deterministically, as do some TLB misses and all page faults. Furthermore, any misspeculation (e.g., due to branch misprediction) during a transaction's execution will trigger an abort. The programmer should assume that every transaction may need to fall back to software mode even in the absence of contention or capacity overflow [12, 13].

4.1 Algorithms

We initially built both a Rock-compatible implementation of the NOrec-hybrid sketched in Dalessandro et al. [10] along with a straightforward implementation of P-Counter. We had to modify the former algorithm slightly to eliminate its dependence on small hardware transactions in the software commit protocol, because Rock does not guarantee such transactions will commit. We found that results using this algorithm were indistinguishable from P-Counter using a single counter, so we do not consider it further. The distributed counter in P-Counter uses a fixed number of counters, and threads are mapped to them by thread id.

Lazy Subscription Rock's HTM lends itself well to the sandboxing implementation of lazy subscription (Section 3.2). Memory accesses are transactional by default and therefore have no effect if the transaction aborts. Most potential side effects of executing with inconsistent data, such as exceptions, simply cause the transaction to abort and therefore do not have a visible effect. We believe manual sandboxing instrumentation is needed only for commit instructions, nontransactional stores, and indirect branches that potentially lead to one of these (including unintentionally branching into executable data that may contain them). Referring to Figure 5, HW_PRE_COMMIT serves to sandbox commit instructions, so to fully sandbox Rock transactions, we must simply insert calls to HW_VALIDATE before nontransactional stores and indirect branches.

We find that lazy subscription and sandboxing consistently outperforms both eager HW_POST_BEGIN subscription and lazy subscription with read instrumentation. Admittedly, our benchmarks use neither nontransactional stores nor indirect branches. Rock's limitation on the total number of stores in a transaction implies that the use of nontransactional stores will be limited. Thus we do not anticipate significant overhead related to instrumenting them. Indirect branches are more common: they are used to implement function call returns, virtual function calls, and computed gotos. We *do* expect that sandboxing will show overheads for transactions that use these operations. Hardware validation requires a load (commonly a cache hit) and an easily predicted branch. As an indirect branch is already a relatively high overhead operation, indirect branch instrumentation should be proportionally less expensive than that of a load or store.

SW-Exists We have found that it is always beneficial to use the SW-Exists filter, implemented using the SNZI algorithm variant from Lev et al. [28]. The simple implementation in Figure 6a, by contrast, suffers frequent cache misses and aborts as the counter changes among non-zero values.

Read-Only We considered the Read-Only optimization but dismissed it for two reasons. First, it consumes precious locations in the store buffer, as multiple writes to the same location, even if thread local and nontransactional, are not necessarily coalesced. We can overcome this cost with a write barrier that checks the value of the flag and only updates it if it is not yet set, but we have seen in HyTM that branching in a barrier has severe negative consequences. Second, as discussed above, using this optimization in a sandboxed algorithm requires validating in HW_PRE_COMMIT, which we had initially hoped to avoid. We do believe that statically identifying read-only transactions and using HW_VALIDATE as the HW_PRE_COMMIT barrier for such cases is likely to be profitable.

4.2 Test Results

We used the same Rock processor used by Dice et al. [12, 13], and the Deque, Hashtable, and Red-Black Tree microbenchmarks [13]. The STAMP benchmarks, used in Section 5, were not an option on Rock due to hardware limitations—specifically, the fact that function calls within a transaction usually cause an abort. All microbenchmarks use the draft transactional C++ API [1], compiled with an internal development version of Oracle's transactional C++ compiler. STM compilation targets the open-source SkySTM library [28] interface; our hybrid and STM implementations implement this interface.

We report results for four TM systems. CGL is a pessimistic software implementation that simply acquires a global lock in SW_BEGIN and releases it in SW_COMMIT. This admits no concurrency, but is useful when comparing single-thread overheads. NOrec is the software implementation given in Figure 2. PhTM-NOrec is Lev et al.'s [29] PhTM algorithm as implemented by Dice et al. [12] and configured to use NOrec rather than TL2 [14] in the STM phase. (Separate experiments confirm that PhTM-NOrec consistently outperforms the equivalent PhTM-TL2 on our system.)

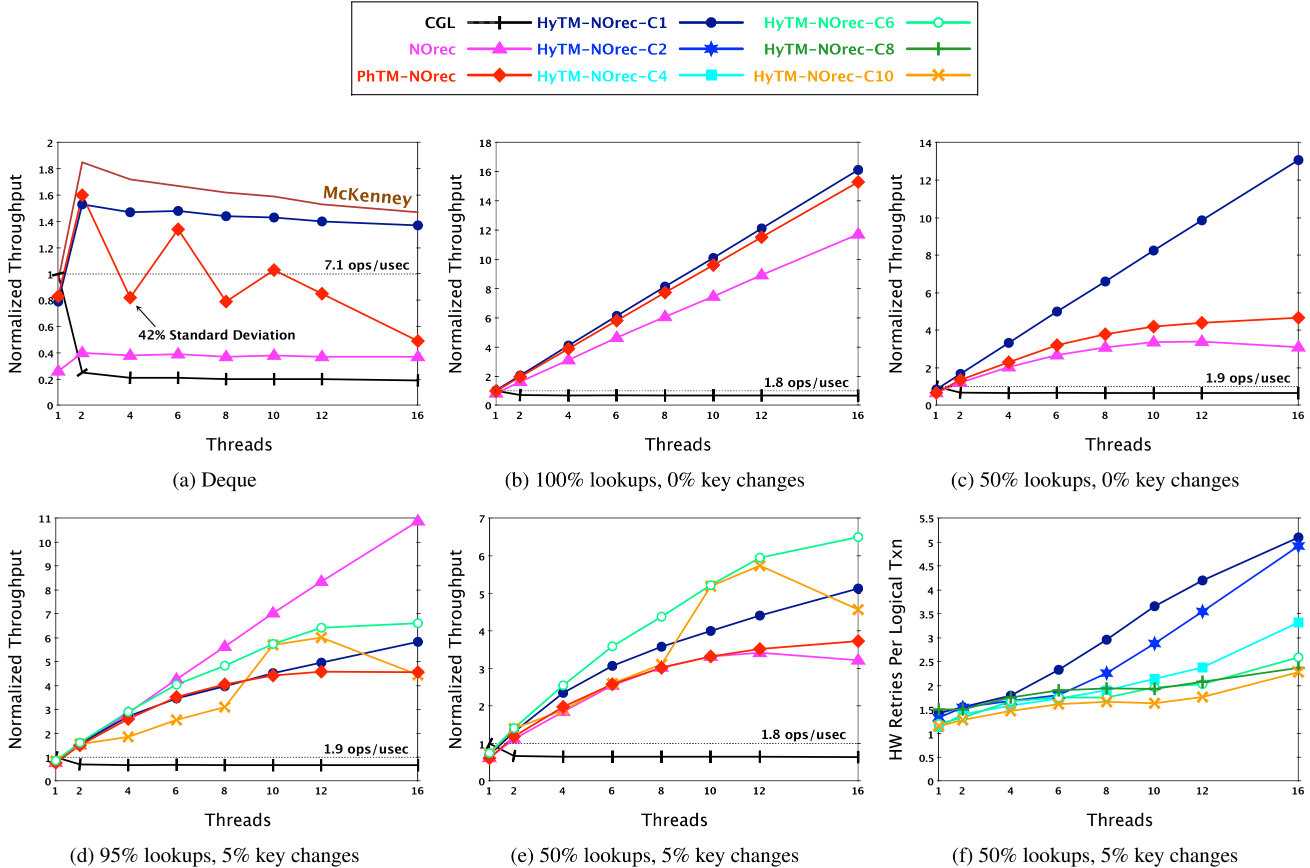
HyTM-NOrec is the P-Counter algorithm with lazy subscription, sandboxing, and the SNZI SW-Exists optimization. This combination consistently performed best. Algorithm suffixes indicate the number of counters used.

For all experiments in this paper, we take five data points, drop the high and low ones, and average the remaining ones. Variance was generally low, except as discussed or indicated.

Deque benchmark In this benchmark, threads are divided evenly between the two ends of a deque, where they each perform $100,000$ operations chosen at random. Accesses to opposite ends of the deque usually do not conflict, so some parallelism is available, but contention among operations on each end means we cannot expect more than a $2\times$ speedup regardless of the number of threads. Results are shown in Figure 7a. For completeness, we also compare to McKenney's state-of-the-art lock-based, two-queue deque implementation [31]—a simple but clever concurrent algorithm that eluded noted concurrency experts for several years [12]. It is interesting that HyTM-NOrec largely closes the performance gap between the simple CGL and McKenney's algorithm without impacting programming complexity.

While CGL performs best with one thread, every transactional method outperforms it in all multithreaded cases, and every hybrid method outperforms NOrec. PhTM-NOrec either matches or slightly improves on the performance of the HyTM-NOrec variants with one and two threads, but then degrades to the point that it is worse than single-threaded throughput at sixteen threads. This is consistent with prior results [12], and suggests that added contention forces PhTM-NOrec into system-wide software mode frequently.

HyTM-NOrec allows a transaction to switch to software mode without causing all other transactions to do so, and thus avoids the cascade effect that PhTM-NOrec suffers. With 16 threads (8 for each end of the deque), only about 3% of operations caused PhTM-NOrec to switch to a software phase, but almost 24% of them *committed* using STM. In contrast, with HyTM-NOrec-1C, 0.6% of transactions switched to software without causing others to switch, allowing for a much higher success rate for hardware transactions, and better throughput and scalability as a result.



(a) Deque (b) 100% lookups, 0% key changes (c) 50% lookups, 0% key changes

(d) 95% lookups, 5% key changes (e) 50% lookups, 5% key changes (f) 50% lookups, 5% key changes

Figure 7. Selected Hashtable and Deque Results. Hashtable results are all for 128K key sets. -NC suffix indicates N hardware counters. Throughput is normalized to the single-threaded CGL results, which are explicitly given. In graphs 7a, 7b, and 7c we show only HyTM-NOrec-1C results as they are indistinguishable from other options. In 7d and 7e we add 6C and 10C results to illustrate their differences.

Hashtable benchmark This benchmark consists of lookup, insert, and delete operations on a large hashtable (2^{17} buckets), parameterized by the number of keys to access. Large key ranges result in access to the entire table, a working set that exceeds Rock's L1 cache capacity, while small key ranges fit comfortably in cache. In either case, conflicts between transactions are rare, and transactions are small and should be able to succeed in hardware. Prior hardware-assisted methods have been able to achieve fast, scalable performance with this workload. This predictable behavior provides us the opportunity to test hybrid TM performance as a function of the number of transactions failing to software in a controlled manner. To do so, we introduce a fourth type of operation, *key change*, which contains an instruction that is not supported in Rock transactions, and thus *must* complete in software.

We have conducted experiments with this benchmark for both small and large key ranges, and with a variety of operation mixes. Although all algorithms are faster when using a small key range such that the hashtable fits in the L1 cache, the results achieved for small and large key ranges are qualitatively similar. A representative sample, all using the large key range (128K), is shown in Figures 7b through 7e.

The HyTM-NOrec family outperforms PhTM-NOrec in almost all cases, often by significant margins. In the 100% lookup case (Figure 7b), almost all transactions succeed in hardware with all HTM-assisted algorithms, and their performance is similar. However, if we introduce a small number of key change operations (Figure 7d), all of the hardware-assisted algorithms perform worse, but PhTM-NOrec is affected worse than most of the HyTM-NOrec variants. This again illustrates the advantage that HyTM-NOrec has over PhTM-NOrec: one transaction switching to software does not cause a cascade of other transactions to do so too.

Although HyTM-NOrec outperforms PhTM-NOrec, it turns out that NOrec significantly outperforms all hardware-assisted methods in some cases; Figure 7d shows one example. There are several reasons for this. First, the workload is heavily dominated by read-only transactions, and NOrec overhead is low for such transactions: its serialized writeback phase is executed infrequently and therefore does not become a bottleneck.

Second, all of our hybrid TM implementations first try to use a hardware transaction, repeatedly in some cases, before switching to software if they do not succeed. The time spent on such transactions is largely wasted when they ultimately resort to software, as at least the key change operations must in this case. There are several possible ways to improve this situation. If the HTM gave clearer feedback about which transactions are unlikely to succeed, we might be able to switch to software more quickly. Relatedly, an HTM that does not fail transactions due to transient conditions such as cache misses and branch prediction may require less retrying in order to achieve good performance for some cases, thus allowing transactions that are going to switch to software to do so sooner. Finally, if compiler analysis or profiling could identify transactions that are unlikely to succeed using HTM, the system could run them in software from the outset.

Third, a transaction switching to software not only makes that operation slower, but can also degrade the performance of ongoing hardware transactions. For example, software transactions cause changes to the SW-Exists mode which causes hardware aborts due to branch misprediction. Moreover, software transactions require hardware transactions to increment counters that are often not present in their caches, adding substantial overhead to otherwise fast transactions. Therefore, hardware transactions are much more likely to fail when executed while software transactions are running. To avoid a cascade effect, the HyTM-NOrec logic sets a much higher threshold for failing to software if it observes, after a hardware transaction fails, that a SW transaction is already running.

As the number of non-read-only operations increases, NOrec's serialized writeback phase becomes a bottleneck, resulting in negative scalability at high thread counts. The HyTM-NOrec algorithms continue to increase throughput with larger numbers of threads, while the PhTM-NOrec performance is limited by that of its STM component. This can be seen clearly in Figure 7c. Even as we force some transactions to software (Figure 7e), the HyTM-NOrec variants outperform NOrec.

Within the HyTM-NOrec family, we observe that larger numbers of counters reduce hardware-hardware contention. This is due to reduced contention on the counters, and thus fewer conflicts between hardware transactions. For example, Figure 7f shows how adding counters reduces the abort rate. However, it is not always the case that such reduced contention increases overall throughput. Observe, for example, that performance becomes worse in Figure 7e going from 12 to 16 threads when using 10 counters. This is not surprising. More counters means validation of software transactions takes longer. Especially at higher threading levels, this in turn means there is more chance of a concurrent hardware transaction committing, forcing the software transaction to revalidate. Additionally, the `SW_COMMIT` barrier requires that software transactions read all of the hardware counters—likely missing in cache for some of them—after acquiring `seqlock` at commit, thus increasing NOrec's serial bottleneck. Similarly, in workloads in which there is application-level contention, slower validation makes it more likely that an actual conflict occurs, forcing the software transaction to retry. Thus, different numbers of counters are preferable for different workloads and threading levels.

We have not experimented with adaptive policies, such as varying the number of counters used, nor with static analysis to identify which transactions should attempt to use hardware transactions, and which should resort to software more quickly, or even immediately. We did experiment with various more sophisticated synchronization mechanisms between hardware and software transactions, but as in previous work [13], code used to implement more complex decisions about whether and when to retry a failed hardware transaction can affect cache, branch predictor, and TLB state, often reducing the chances for a subsequent attempt to succeed. Therefore, a more robust best-effort HTM may be needed to effectively implement such policies.

Red-black tree benchmark We have also performed experiments with red-black trees (not shown), similar to those reported in Dice et al. [13]. In experiments with 100% lookups on small trees (key ranges of 128 and 512), HyTM-NOrec-1C performs best or close to best in all multithreaded tests. However, as we increase the key range and/or the fraction of operations that modify the tree, transactions acquire larger read sets and unpredictable branches, and have more trouble succeeding, for reasons similar to those reported by Dice et al. [13]. Again, a best-effort HTM that is less prone to failure due to misspeculation may be needed to improve our results for these kinds of workloads.

Discussion We also tested a variant of PhTM whose software phase executes a single, uninstrumented transaction at a time (`SEQUENTIAL-NOABORT` mode in Lev et al. [29]), as tested by Christie et al. [9]. While it outperformed other algorithms in some of our tests, we do not consider this a viable solution. It cannot support explicit user-level abort due to its lack of instrumentation. It cannot execute multiple transactions concurrently, even if they are read-only. In contrast, NOrec allows read-only transactions to run concurrently without interference, and serializes only the writeback phase of non-read-only transactions, not the entire transaction. A simple variant of the `SEQUENTIAL-NOABORT` PhTM mode could allow read-only transactions to execute concurrently by using a reader-writer lock. However, this mode can be used only by transactions that are known in advance to be read-only. NOrec overcomes all of these problems.

It is tempting focus on performance comparisons between our new hybrid algorithms and previous systems. However, in our experience, performance of these systems can be highly dependent on idiosyncrasies of Rock, and minor changes in tuning parameters, optimizations, retry policies, etc. can have significant impact on performance. Thus, conclusions we would draw from such comparisons may be invalid for more robust HTM implementations. It is more important to identify specific implementation-related challenges to performance and scalability, and how these challenges differ between our system and previous work.

We have observed that tuning of PhTM—for example when to switch to and from STM phase or which STM to use in that phase—is sensitive to a number of factors, including specific characteristics of workloads. An unnecessary phase switch in PhTM can be expensive, and if this occurs frequently, it can have disastrous consequences for performance. In contrast, HyTM systems make these decisions on a per-transaction basis, and thus overall performance is much less sensitive to occasional wrong decisions.

On the other hand, we have seen that NOrec-based hybrid solutions on Rock do not isolate hardware and software performance as much as expected, as the software transactions influence the performance of concurrent hardware transactions more strongly than the basic algorithm implies on its surface.

5. ASF

AMD's ASF proposal [3, 15] extends the x86_64 architecture with explicit transactional constructs. ASF guarantees that transactions accessing four or fewer locations will never abort due to limited capacity. This bound admits an implementation that tracks the cache lines accessed by a transaction in a 4-way set-associative cache, while leaving open the possibility of using a separate hardware "Locked Line Buffer" (LLB) instead [9]. ASF hardware is not currently available; we used PTLsim, a cycle-accurate out-of-order x86 simulator [43], running Linux 2.6.20.

Within transactions, loads and stores must be prefixed with the x86 `LOCK` prefix to be considered transactional; unlabeled accesses are nontransactional. The exact memory model for the resulting mix of transactional and nontransactional access is not clearly defined [3], but the implication (and the behavior of the current simulator) is that ordinary accesses propagate immediately, and become globally visible, without preserving program order with respect to transactional stores. ASF checkpoints only the program counter and stack pointer at the beginning of a transaction; any rollback of general registers on abort must be achieved by software (e.g., using compiler support or `setjmp`). A transaction that suffers a page fault is restarted automatically after completion of the usual kernel page-in operation. Diestelhorst et al. [16] discuss the challenges that out-of-order execution presents to progress guarantees.

ASF transactions that encounter traps, exceptions, infinite loops, etc., abort without visible side effects, such as segmentation violations (Section 4.1). Because nontransactional stores do not consume transactional resources on ASF, programmers and compilers may choose to use them more than they would on Rock. Such use may increase the cost of sandboxing, because each nontransactional store may need to be instrumented with HW_VALIDATE.

5.1 Algorithms

As in Section 4, we implement 2-Location as a baseline hybrid TM system. Given ASF's guarantees for small transactions, we use HTM to emulate a double-location compare-and-swap (DCAS), as proposed by Dalessandro et al. [10]. We keep both counters on the same virtual page, so SW_COMMIT will never fail due to capacity or pathological page allocation. However, by using a hardware transaction to perform the DCAS, software transactions lose the livelock-free guarantee of the original NOrec algorithm.

We find that the Read-Only filter (Figure 6b) is always beneficial on ASF as the nontransactional write to readonly does not consume HTM resources. Every hybrid algorithm described in this section includes the Read-Only filter.

We refer to our ASF variant of 2-Location with the Read-Only filter as 2-Location-ASF, or 2LA.

Lazy subscription In addition to 2LA, we tested lazy subscription algorithms based on the naive algorithm in Figure 4a, with the read and branch on seqlock delayed from HW_POST_BEGIN to HW_PRE_COMMIT for hardware transactions. We tested both the Opaque and Sandbox methods described in Section 3 for avoiding incorrect behavior due to inconsistent execution.

Opaque uses post-read validation barriers as detailed in Figure 5. The use of nontransactional loads in HW_VALIDATE avoids conflicts and allows a transaction to survive concurrent software writer commits, but it does not let the transaction make progress during those commits. This algorithm requires one line of transactional capacity for metadata in writing transactions, as we read seqlock transactionally in HW_PRE_COMMIT. Sandbox relies on sandboxing for correctness, rather than calling HW_VALIDATE after every read.

SW-Exists Opaque and Sandbox update seqlock in non-read-only transactions, regardless of the presence of software transactions. In Opaque-SWExists and Sandbox-SWExists, we add the SW-Exists filter (Figure 6a) to avoid this. As spurious updates of seqlock do not affect correctness, we leverage nontransactional loads to avoid some artificial conflicts, while using a transactional load to prevent races due to a software transaction arriving, executing, and committing concurrently with a hardware transaction committing. The pseudocode below demonstrates how we handle this circumstance by subscribing to the sw_exists field when attempting to commit without modifying seqlock. (Variants that do not use the SW-Exists filter do not include lines 5, 6, 10 and 11.)

```
1  HW_PRE_COMMIT:
2    if (non_tx_load(readonly))
3      HW_VALIDATE // not needed in Opaque
4      return
5    if (non_tx_load( sw_exists ) != 0)
6      HW_VALIDATE
7      reg = tx_load(seqlock)
8      if (reg & 1) asf_abort
9      tx_store (seqlock, reg+2)
10   else
11     if (tx_load( sw_exists ) != 0) asf_abort
```

Calling HW_VALIDATE before loading seqlock transactionally helps to avoid subscribing to seqlock during a software transaction writeback, which forces the hardware transaction to abort.

We used a simple counter for sw_exists, not a SNZI. Similarly, we did not experiment with separate counter(s) for hardware transactions to signal their commits. The Read-Only and SW-Exists optimizations are effective in reducing contention from hardware transactions on seqlock, and our use of ample hardware resources (see Section 5.2) further avoids the need to resort to software frequently. Furthermore, even when transactions do resort to software, if transaction commits are infrequent enough that contention on sw_exists or seqlock is unlikely, using SNZI and separate counters is unnecessary. Nonetheless, for different workloads and/or system configurations, it may be necessary to revisit the use of SNZI and/or separate counters to aid scalability.

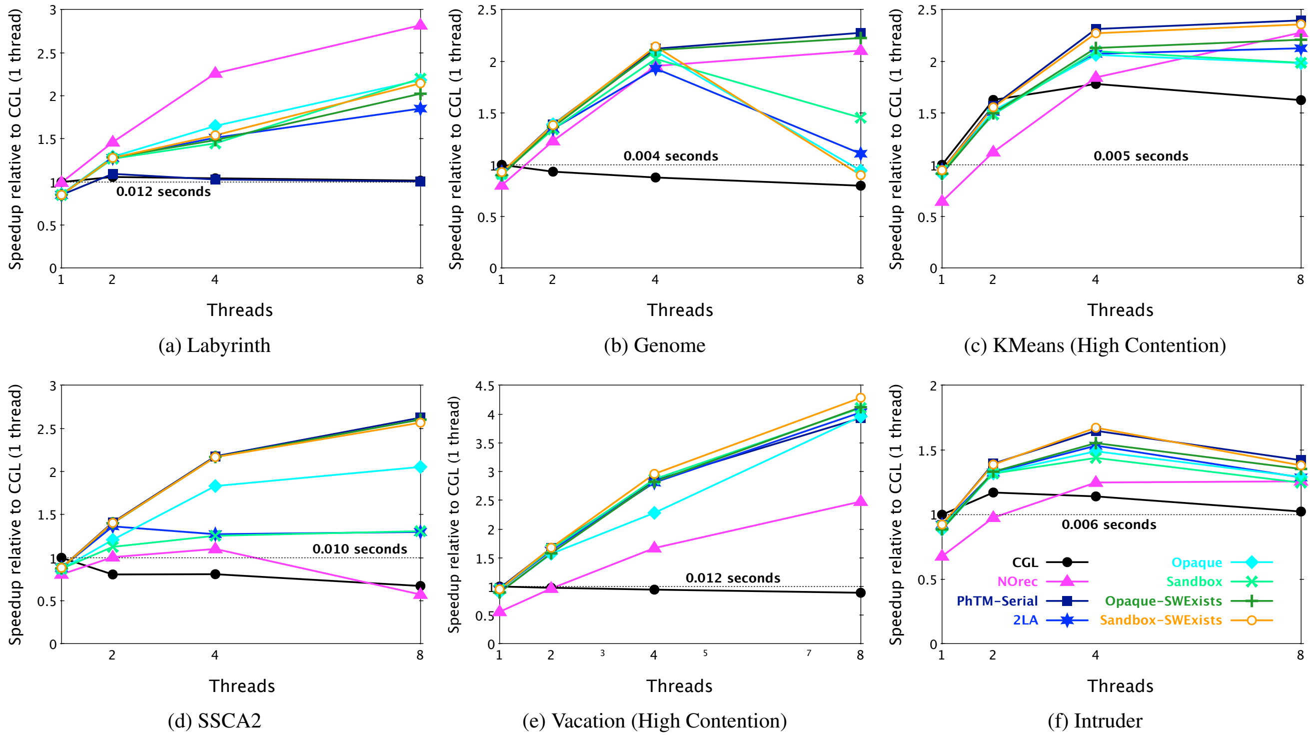
5.2 Test Results

We evaluated our hybrid NOrec algorithms using the STAMP benchmark suite [7] on the PTLsim ASF simulator [43]. We did not employ a transactifying compiler for ASF, instead putting all hybrid instrumentation in libraries. This enabled us to easily add per-access instrumentation to hardware transactions. Furthermore, as STAMP is manually instrumented, this decision avoided unnecessary instrumentation due to imprecise compiler analysis, which could produce unnecessary best-effort HTM capacity overflows. We did not add any sandboxing instructions to STAMP for our experiments, and thus our results provide a best-case estimate of performance. This decision appears safe based on a review of STAMP's code, and because STAMP provides limited post-operation sandboxing, based on programmer knowledge, with an explicit abort-and-restart macro.

We used PTLsim's ASF out-of-order core module (asfooo) with a locked-line buffer (LLB) of 256 entries. We modified the simulator to support up to 16 cores, but report results here for only 8: performance at 16 was uniformly poor, and noisy as well, suggesting an inherent lack of scalability in the simulated machine. Code was compiled with gcc 4.3.2, using –O3 optimizations. Results were generated using the default "simulator" input parameters for STAMP. Due to space constraints, we present only a subset of these results; the trends and observations we report extend to the entire suite. We tested basic CGL and NOrec implementations as well as the five hybrid NOrec variants described above. In addition, we tested PhTM-Serial, a version of PhTM in which software-mode consists of a single serial transaction that executes with lightweight instrumentation so as to support user-level aborts. Blundell et al.'s [5] use of such serial execution in unbounded HTM suggests that this may be a viable alternative if hardware aborts are rare. Our hybrid infrastructure is configured to retry hardware transactions repeatedly with exponential backoff unless a transaction exceeds the capacity of the 256-entry LLB, in which case it falls back to software.

With a 256-entry LLB, few STAMP transactions exceed the capacity of the LLB and therefore need to run in software; see Table 1. Furthermore, comparing the behavior of algorithms that do not allow conflicts on metadata between hardware transactions (such as PhTM-Serial) and those that do, we find that the abort rates are very similar for most workloads, suggesting that there are few such conflicts and therefore that the use of SNZI and multiple counters would probably not provide significant benefit for these workloads at the threading levels we tested.

The Benefit of Lazy Subscription Our results support the intuition that PhTM-Serial is inappropriate for general purpose use. In Bayes (not shown) and Labyrinth (Figure 8a) it does not scale at all. For most of the STAMP benchmarks, 2LA scales well to 8 threads, and offers better performance than the NOrec algorithm on which it is based. Lazy subscription reduces the time a hardware transaction is vulnerable to aborts by software commits, and as a result the lazy subscription algorithms often outperform 2LA. However,



(a) Labyrinth (b) Genome (c) KMeans (High Contention)

(d) SSCA2 (e) Vacation (High Contention) (f) Intruder

Figure 8. STAMP benchmarks, run on a simulated 8-core ASF x86_64 system. Results are shown as speedups over the CGL single-threaded execution time, which is also explicitly given.

Benchmark	Total Transactions	Percent Hardware
Labyrinth	194	94.3%
Genome	5912	99.4%
KMeans	8193	99.9%
SSCA2	47257	99.9%
Vacation	4096	97.7%
Intruder	11209	99.5%

Table 1. Dynamic transaction count and percentage that fit in the ASF LLB, allowing them to complete in hardware (single-threaded execution).

they suffer the same metadata conflicts between hardware transactions, which can eliminate the benefit of lazy subscription if those conflicts occur frequently. For example, in the SSCA2 benchmark, Sandbox performs just as poorly as 2LA. The SW-Exists filter can avoid these conflicts, as discussed below, allowing the lazy subscription algorithms to outperform all others. Thus, while lazy subscription is not always sufficient in its own right, it is a key enabler for the best algorithms.

The SW-Exists Filter By comparing Sandbox-SWExists to Sandbox and Opaque to Opaque-SWExists, we can assess the benefit of avoiding metadata modifications when committing writer transactions. For this evaluation, we used SSCA2 (Figure 8d) which executes roughly 47K transactions, all of which modify shared state, and perform at most six memory operations (and therefore never need to fall back to STM).

NOrec cannot scale on this workload, as its single counter becomes a bottleneck. Among hybrid algorithms, PhTM-Serial is best, as there are no software transactions, and PhTM-Serial does not require metadata updates for hardware transactions. Furthermore, the low abort rate for PhTM-Serial indicates that conflicts on program data are rare.

The algorithms that use the SW-Exists filter perform almost as well as PhTM-Serial. The -SWExists implementations avoid any modification of metadata when all transactions run in hardware mode, and thus the slight slowdown relative to PhTM-Serial, which is not coupled with a noticeable increase in abort rate, indicates that Opaque-SWExists and Sandbox-SWExists are paying a small amount in unnecessary instrumentation, but are not wasting work by aborting due to metadata conflicts.

In contrast, the algorithms that do not use the SW-Exists filter (2LA, Sandbox, and Opaque) experience an increase in aborts as the thread count increases, resulting in significantly worse performance. For example, Sandbox reaches 1.1 aborts per commit at 8 threads, whereas PhTM-Serial has only 0.002 aborts per commit at the same thread count. As Sandbox and Sandbox-SWExists differ only in their use of the SW-Exists filter (and likewise Opaque and Opaque-SWExists), we conclude that the aborts in Sandbox and Opaque are a consequence of conflicting updates to metadata, which the SW-Exists filter avoids.

Opacity vs. Sandboxing Although neither of these approaches is a clear winner, we observe Sandbox-SWExists outperforming Opaque-SWExists in most cases, due to its lower instrumentation costs. In some cases, however, the opposite is true. Recall that both opaque and sandboxing algorithms access `seqlock` transactionally whenever a writing transaction commits. When this variable is not cached at commit time, the commit operation may take longer, increasing vulnerability to unnecessary aborts due to conflicts with in-flight transactions. For the opaque algorithms, HW_VALIDATE is called on every read, and thus `seqlock` is likely to remain in cache. For the sandboxing algorithms, compiler-inserted calls to

HW_VALIDATE can have the same prefetching effect. However, as our instrumentation omitted these calls, the sandboxing algorithms were more likely to experience a cache miss on seqlock at commit time.

When Software TM is Better While only 5% of transactions exceed the hardware's transactional capacity in Labyrinth, the cost of running these transactions almost to completion in hardware before aborting and retrying in software resulted in worse performance than NOrec even at 1 thread. In Labyrinth, transactions first perform thousands of nontransactional loads and stores, then up to several hundred transactional loads and stores. Thus, a hardware transaction that overflows the LLB capacity does so after performing a lengthy prefix, magnifying the cost of the overflow.

Across all workloads, the relative benefit of hybrid TM seems to diminish at higher thread counts. This is not surprising because NOrec is livelock-free, its transactions do not abort due to false sharing, and it is effective at preventing aborts, whereas hybrid TM with requester-wins conflict detection is prone to livelock and can experience high abort rates, even in workloads with few conflicts. Labyrinth at 8 threads provides an extreme example: across all threads, NOrec aborted 8 times when committing 208 transactions; every hybrid algorithm aborted at least 600 times.

More generally, when the fraction of execution time that all software transactions spend in the commit operation is small, the instrumentation and shared metadata inherent in STM algorithms can become less of a bottleneck, and the wasted work due to excessive hybrid TM aborts can outweigh its lower instrumentation overhead. This can be particularly true as read-write conflicts are not deferred until commit time by hybrid TM: seemingly benign conditions such as a software transaction validating, or sharing between a committing reader and in-progress writer, can cause hardware transactions to abort.

6. Discussion

A concern with NOrec has been the lack of scalability of its serialized writeback mechanism: a workload that performs many write-dominated transactions will perform poorly. STAMP's SSCA2 benchmark is an example; Figure 8d clearly illustrates NOrec's weakness. Our results show that NOrec-based hybrids can avoid this problem. If an application completes a large fraction of its logical transactions in hardware, then its scalability is determined more by the hardware than by the STM algorithm. Our tests show NOrec-based hybrid scalability to be good for such workloads, including for SSCA2. Admittedly, some applications may include frequent transactions that are ill suited for execution using either HTM or NOrec, in which case it may be desirable to include modes that use more scalable STM algorithms.

In addition to achieving some encouraging results, we have made observations about the impact that aspects of best-effort HTM design have on its effectiveness in NOrec-based, hybrid systems. Neither PhTM-NOrec nor the HyTM-NOrec variants are competitive with NOrec when transactions commonly require software execution, because logical transactions that ultimately succeed in software do so only after failed attempts as hardware transactions. HyTM implementations have a potential advantage over PhTM in dealing with this problem: static analysis, profiling, and programmer guidance may allow transactions that are unlikely to succeed using HTM to aggressively switch to software, without forcing the entire system to undergo a mode change. We would like to construct more adaptive retry policies. However, limitations of best-effort HTM make it difficult without sacrificing performance in other cases. Below we discuss the impact of Rock's and ASF's HTM designs on our hybrid systems, and make some suggestions for designers of future HTM features to consider.

Branch prediction Rock's sensitivity to branch misprediction in transactions negatively impacts adaptive policies, which often branch between alternatives based on the contents of a shared variable. By nature, shared variables are not necessarily cached, and the branches associated with such adaptation can be unpredictable. Rock provides *speculation barriers* [13] that can avoid the need to predict a branch accurately, but their high overhead prevents their ubiquitous use. We strongly urge HTM implementors to avoid dependence on branch prediction for transactional success.

Contention management ASF's guarantees for small transactions are attractive but apply only in the absence of conflicts. In practice, backoff is necessary to avoid livelock. This is inherent to the requestor-wins conflict resolution policy used by both ASF and Rock. Backing off just long enough to avoid conflicts is difficult. Thus, STM sometimes outperforms HTM because it can actively resolve conflicts, rather than merely avoid them using backoff.

Requestor-wins has a subtle effect in our HyTM-NOrec implementations. A software transaction reads data, including program data as well as metadata such as counters, in order to detect potential conflicts. If a concurrent hardware transaction has speculatively written such data, requestor-wins will cause the hardware transaction to abort, without guaranteeing that the software transaction will succeed. This increases the potential for poor performance and livelock, even in workloads where true conflicts are infrequent.

Unlabeled Accesses Rock and ASF both provide the capability to execute transactional and nontransactional accesses within a transaction. Both Rock and ASF require one type to be explicitly labeled, and treat unlabeled accesses as the other type. However, they make opposite choices: Unlabeled ASF accesses are nontransactional whereas Rock considers them to be transactional. Rock's approach allows the same code to be used both transactionally and nontransactionally and supports the use of libraries inside transactions, Transactional Lock Elision (TLE) [13] with a single code path, and straightforward compilation of transactional code. ASF's choice carries the implicit assumption that transactional accesses have large incremental costs and require careful use. This is true in a minimally conforming ASF implementation and a good assumption even in PTLsim's default 256 LLB implementation. However it complicates code generation and library reuse.

Rock's prohibition on CAS inside transactions limits some of the benefits of treating unlabeled accesses as transactional. Consider memory allocation, where a hardware transaction calls a thread-safe malloc implemented with a CAS-based lock. On Rock, the transaction will fail due to the restriction on CAS. On ASF, it *can* succeed, but a worse problem arises if a transaction aborts while executing malloc, because it will fail to release the lock, preventing all subsequent memory allocation. An ideal implementation would allow ordinary code to be executed within hardware transactions, and have its effects negated upon abort, but provide for unfettered use of nontransactional operations too.

Sandboxing for lazy subscription requires validation barriers on nontransactional accesses. If nontransactional accesses are more common on ASF than on Rock, then sandboxing may be more expensive on ASF.

Nontransactional accesses ASF and Rock both support nontransactional accesses within transactions, but with different memory-model semantics. Rock's nontransactional accesses are required to preserve TSO regardless of the surrounding transaction's success or failure. The specification for ASF's nontransactional accesses is currently incomplete, but the implication (and PTLsim's implementation) is that they can become globally visible without respecting the program order of interleaved transactional accesses.

Several of our key optimizations depend on support for nontransactional loads from shared locations, for which ASF's and

Rock's differing semantics are both effective. Riegel et al. [36] find that it is beneficial to be able to access shared data with a nontransactional CAS instruction inside a transaction (Section 7), provided the CAS becomes visible before the transaction is committed.

We do not currently have a use for nontransactional stores to shared locations, but we do find nontransactional stores to thread private locations useful. For these, strong ordering with respect to stores of other threads is not needed. PTLsim effectively supports such accesses and the corresponding optimizations. However, Rock's requirement that nontransactional stores respect TSO leads them to occupy a store buffer slot, inhibiting their use significantly. For example, on Rock, dynamically tracking whether a transaction performs a store (for the Read-Only optimization) requires us to avoid writing the flag on every store, complicating the write barrier code and introducing a branch we would prefer to avoid.

We recommend that designers carefully consider how they can support flexible use of nontransactional accesses, as well as the interaction of those accesses with the memory consistency model. White and Spear [42] suggest that it may be useful to distinguish between nontransactional stores that become visible before the transaction commits and those that respect (say) TSO order.

Visibility into transactional execution Information about the behavior of a transaction that is executing can be useful for a variety of reasons. For example, a cheap way to determine how many loads and stores a transaction has performed would provide a convenient solution to the problem of determining whether a Read-Only optimization can be applied in a given transaction, or could inform a decision about whether and when to retry. It is also useful for code to be able to determine whether it is executing in a transaction (for example to allow a TLE implementation to decide between committing a transaction and releasing a lock). Numerous other kinds of information are potentially useful, such as addresses accessed, addresses that cause conflicts, etc. In addition, transaction-aware performance counters and sampling mechanisms may be useful.

Scalability with contention With requestor-wins conflict resolution, it is not surprising that the naive HyTM-NOrec algorithm that requires every hardware transaction to increment a shared counter at the end performs poorly. It may be tempting for designers to consider this "a software problem," especially in light of our success in distributing the counters in the SW-Exists algorithm. However, the techniques we used not only complicate our algorithms but entail performance tradeoffs, such as requiring software transactions to read more counters. By aiming to avoid pathological performance under contention in such cases, designers can allow us to avoid these mechanisms, or at least to use fewer counters. Possible approaches include more sophisticated conflict resolution policies, (perhaps limited) support for nesting, and support for nontransactional CAS within transactions.

7. Related Work

In related concurrent work, Felber et al. [19] and Riegel et al. [36] evaluate hybrid TM systems on the ASF simulator. Like ours, one of their hybrids starts with the sketch from Dalessandro et al. [10], leading to some overlap with the basic algorithm that we have presented. This includes lazy subscription via read instrumentation, though not with sandboxing, and the Read-Only filter. Riegel et al. observed that ASF's nontransactional CAS capability allows hardware transactions to modify the `seqlock/counter` nontransactionally within a transaction in order to avoid unnecessary conflicts between hardware transactions. While this capability is not clearly supported by the ASF documentation, it works using the current simulator implementation, and is shown to be effective at preventing counter contention from causing unnecessary aborts.

Interestingly, the nontransactional CAS optimization is both orthogonal and complementary to the most effective of our ASF optimizations, Read-Only and SW-Exists. Our limited testing suggests that integrating the three techniques often slightly outperforms Read-Only with SW-Exists for the STAMP applications. This avenue deserves more attention as the ASF specification matures.

8. Conclusion And Future Work

We have explored the use of best-effort HTM to implement several hybrid versions of the low-overhead NOrec. These NOrec-based HyTM algorithms support the concurrent execution of hardware and software transactions, as previous hybrid TMs do, but with the low overhead on hardware transactions previously achieved only in phased TMs.

Our results show the significant potential for this approach, but also reveal a number of challenges in realizing that potential. We expect to continue to improve our results using the Rock and (simulated) ASF best-effort HTM as currently designed. However, it appears that overcoming some of the challenges will require better best-effort HTM support than the existing implementations and proposals provide. An important contribution of our work is to identify and discuss such issues, in order to guide designers of future best-effort HTMs.

Perhaps the greatest opportunities lie in adaptation. The P-Counter algorithm, for example, would benefit from a dynamic choice of an appropriate number of counters. We also need better ways to identify transactions that should start in software mode. The decision might be made using compile-time static analysis or profiling feedback, or using runtime lightweight statistics gathering and dynamic adaptation [39]. Even with good initial choices, we need faster ways to determine when a given dynamic transaction should resort to software, rather than retrying in hardware.

Acknowledgments

Our thanks to Dave Dice for his implementation of NOrec STM, to Dan Nussbaum for his initial integration of NOrec into Oracle's SkySTM hybrid infrastructure, to Stephan Diestelhorst for updated PTLsim source code and advice on extending it to larger numbers of threads, and to our shepherd Milo Martin for his patience and helpful advice.

References

[1] A.-R. Adl-Tabatabai and T. Shpeisman (Eds.). Draft Specification of Transactional Language Constructs for C++. `research.sun.com/scalable/pubs/C++-transactional-constructs-1.0.pdf`, Aug. 2009. Version 1.0.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun. 2006.

[3] Advanced Micro Devices. Advanced Synchronization Facility: Proposed Architectural Specification. Publication #45432, rev. 2.1, `developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, Mar. 2009.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *11th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2005.

[5] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35:24–34, June 2007.

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), Nov. 2006.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *IEEE Intl. Symp. on Workload Characterization*, Sep. 2008.

[8] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, and S. Yip. Rock: A High-Performance SPARC™ CMT Processor. *IEEE Micro*, 29(2):6–16, Mar.-Apr. 2009.

[9] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *EuroSys Conf.*, Apr. 2010.

[10] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *15th ACM Symp. on Principles and Practice of Parallel Programming*, Jan. 2010.

[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[12] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying Concurrent Algorithms by Exploiting Hardware Transactional Memory. In *22nd ACM Symp. on Parallelism in Algorithms and Architectures*, 2010.

[13] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early Experience with a Commercial Hardware Transactional Memory Implementation. SMLI TR-2009-180, Sun Microsystems Laboratories, Oct. 2009.

[14] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *20th Intl. Symp. on Distributed Computing*, Sep. 2006.

[15] S. Diestelhorst and M. Hohmuth. Hardware Acceleration for Lock-Free Data Structures and Software-Transactional Memory. In *Wkshp. on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, Apr. 2008.

[16] S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, J.-W. Chung, and L. Yen. Implementing AMD's Advanced Synchronization Facility in an Out-of-Order x86 Core. In *5th ACM SIGPLAN Wkshp. on Transactional Computing*, Apr. 2010.

[17] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun. 2007.

[18] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *26th ACM Symp. on Principles of Distributed Computing*, Aug. 2007.

[19] P. Felber, C. Fetzer, P. Marlier, M. Nowack, and T. Riegel. Brief announcement: Hybrid time-based transactional memory. In N. Lynch and A. Shvartsman, editors, *Distributed Computing*, Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2010.

[20] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *13th ACM Symp. on Principles and Practice of Parallel Programming*, 2008.

[21] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabju, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *31st Intl. Symp. on Computer Architecture*, Jun. 2004.

[22] T. Harris and K. Fraser. Revocable Locks for Non-Blocking Programming. In *10th ACM Symp. on Principles and Practice of Parallel Programming*, Jun. 2005.

[23] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Jun. 2006.

[24] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan Claypool, 2nd edition, 2010.

[25] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *20th Intl. Symp. on Computer Architecture*, May. 1993.

[26] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *11th ACM Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[27] C. Lameter. Effective Synchronization on Linux/NUMA Systems. In *May 2005 Gelato Federation Meeting*, May. 2005.

[28] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *4th ACM SIGPLAN Wkshp. on Transactional Computing*, 2009. `http://research.sun.com/scalable/pubs/TRANSACT2009-ScalableSTMAnatomy.pdf`.

[29] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *2nd ACM SIGPLAN Wkshp. on Transactional Computing*, Aug. 2007.

[30] S. Lie. Hardware Support for Unbounded Transactional Memory. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

[31] P. E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? `http://www.rdrop.com/users/paulmck/perfbook/perfbook.2010.01.23a.pdf`, 2010. [Viewed Jan. 24, 2010].

[32] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *20th ACM Symp. on Parallelism in Algorithms and Architectures*, June 2008.

[33] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *12th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2006.

[34] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *16th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2007.

[35] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *32nd Intl. Symp. on Computer Architecture*, Jun. 2005.

[36] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. TUD-FI10-06-Nov.2010, Technische Universitaet Dresden, Nov. 2010.

[37] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[38] M. Spear, V. Marathe, W. Scherer, and M. Scott. Conflict detection and validation strategies for software transactional memory. In S. Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2006.

[39] M. F. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *22nd ACM Symp. on Parallelism in Algorithms and Architectures*, June 2010.

[40] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional Value Prediction. In *4th ACM SIGPLAN Wkshp. on Transactional Computing*, Feb. 2009.

[41] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Intl. Symp. on Code Generation and Optimization*, Mar. 2007.

[42] S. White and M. Spear. On Reconciling Hardware Atomicity, Memory Models, and __tm_waiver. In *2nd Workshop on the Theory of Transactional Memory (WTTM)*, Sep. 2010.

[43] M. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *2007 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, Apr. 2007.