NOrec: Streamlining STM by Abolishing Ownership Records *

Luke Dalessandro[1] Michael F. Spear[2] Michael L. Scott[1]

[1]Computer Science Dept., Univ. of Rochester [2]Computer Science and Engineering, Lehigh University
{luked, scott}@cs.rochester.edu spear@cse.lehigh.edu

Abstract

Drawing inspiration from several previous projects, we present an ownership-record-free software transactional memory (STM) system that combines extremely low overhead with unusually clean semantics. While unlikely to scale to hundreds of active threads, this "NOrec" system offers many appealing features: very low fast-path latency—as low as any system we know of that admits concurrent updates; publication and privatization safety; livelock freedom; a small, constant amount of global metadata, and full compatibility with existing data structure layouts; no false conflicts due to hash collisions; compatibility with both managed and unmanaged languages, and both static and dynamic compilation; and easy accommodation of closed nesting, inevitable (irrevocable) transactions, and starvation avoidance mechanisms. To the best of our knowledge, no extant STM system combines this set of features.

While transactional memory for processors with hundreds of cores is likely to require hardware support, software implementations will be required for backward compatibility with current and near-future processors with 2–64 cores, as well as for fall-back in future machines when hardware resources are exhausted. Our experience suggests that NOrec may be an ideal candidate for such a software system. We also observe that it has considerable appeal for use within the operating system, and in systems that require both closed nesting and publication safety.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Algorithms, Performance

Keywords Ownership Records, Software Transactional Memory, Transactional Memory, Transactional Memory Models

1. Introduction

The last few years have witnessed a flurry of interest in transactional memory (TM), a synchronization methodology in which the programmer marks regions of code to be executed atomically, and the underlying implementation endeavors to execute such regions concurrently whenever possible, generally by means of speculation. Many researchers—ourselves among them—have come to believe that hardware TM will eventually be seen as desirable—even necessary—in many-core machines. In the meantime, there are hundreds of millions of multicore machines already in the field. For the sake of backward compatibility, emerging TM-based programming models will need to be implemented in software on these machines. It also appears likely that software TM (STM) will be needed as a fall-back on future machines when hardware resources are exhausted [9].

Even for the staunchest advocates of hardware TM, then, STM remains of paramount importance. Since 2003, our group has developed or ported more than 20 different STM systems, with a remarkable degree of variety in implementation strategy. Thirteen of these systems are included in version 5 of the Rochester Software Transactional Memory (RSTM) suite [31].

STM systems differ in buffering mechanism (cloning, redo-log, undo-log), conflict detection time (eager/late/mixed, visible/invisible readers), metadata organization (object-based, ownership records, Bloom filters), contention management strategy, validation strategy (lock-based, timestamp-based, sandboxing), progress guarantees (nonblocking, livelock-free, starvation-free), ordering semantics, and support for special features (nesting, irreversible operations, condition synchronization). This extraordinary variety raises the obvious question: if one is looking for a good, all-around implementation of TM for use in some new programming system, which version should one choose? This paper is our attempt to answer that question.

To be clear: we don't claim the final word on STM. We see significant opportunities for further work in language integration, TM-aware compiler optimizations, and dynamic adaptation to workload characteristics. And much work remains to be done to optimize performance and "harden" existing implementations for production use. At the same time, many pieces of the puzzle have come clear in recent years; it's time to put them together.

For the sake of predictable performance on the widest range of workloads—and in particular for livelock freedom—we have argued [35] that a general purpose STM system must be "lazy"—it must delay the detection of conflicts until transaction commit time. This implies that writes must be buffered in a "redo log." At the same time, it permits very simple contention management—just favor the transaction that is ready to commit.

Perhaps the biggest remaining STM design questions center around metadata organization and mechanisms to *validate* (ensure the consistency of) and commit active transactions. In these there tends to be a natural tradeoff between scalability and overhead: more scalable techniques are more distributed, and thus touch a larger number of separate memory locations. For machines of modest size (up to 32 cores, say), results from systems like RingSTM suggest [34] that centralized metadata, carefully used, can minimize overhead while preserving acceptable scalability.

In this paper we advocate a minimalist approach to transaction metadata: no ownership records (as used in most STM systems), and no Bloom filters (as in RingSTM or many hardware TM systems). We discuss two concrete systems. The first, Transactional Mutex Locks (TML) [36], is particularly simple: the runtime pro-

* This work was supported in part by NSF grants CNS-0615139, CCF-0702505, and CSR-0720796; by equipment support from Sun and IBM; and by financial support from Intel and Microsoft.

PPoPP'10, January 9–14, 2010, Bangalore, India.

tects atomic regions with a single *sequence lock* [17]. Multiple read-only transactions can run and commit concurrently. A reader can upgrade to writer status at any time, but there can be only one writer, system-wide. All other readers abort and retry. While it precludes write concurrency, this system avoids the need for buffered writes, and requires only the most minimal instrumentation: readers check the sequence lock immediately after reading, to see if they must abort; writers acquire the sequence lock immediately before writing, if they do not hold it already. (Even this minimal instrumentation can often be elided based on simple dataflow analysis.) We have yet to see another STM system match the performance of TML at thread counts of roughly four or fewer.

Our second system, which we describe in detail in Section 2, is herein named "NOrec" (no ownership records).[1] It also uses a sequence lock, but only to protect the transaction commit protocol. Readers check, after each read, to see if any writer has recently committed; if so, they perform value-based validation [10, 15, 29] to make sure their previous reads, if performed right now, would return the values previously seen. As in many STM systems (TML, JudoSTM [29], RingSTM, and even TL2 [7]), NOrec's global metadata ultimately imposes a limit on scalability, but the serial bottleneck is relatively small. As in JudoSTM and RingSTM, threads can figure out what they want to write concurrently; as in RingSTM, they can determine, concurrently, that these writes do not conflict with previous writers.

Serialization of commit and (for NOrec) writeback has the very desirable property of ensuring *privatization safety* [23, 32]. When combined with every-reader abort (in TML) or value-based validation (in NOrec), it also ensures *publication safety* [25].[2] Post-read validation avoids the need for sandboxing, as performed in JudoSTM, to prevent erroneous behavior in transactions that are doomed to abort. Privatization safety makes it acceptable to access shared data structures outside transactions, at no additional cost to either transactional or nontransactional code, so long as the overall program is transactional data race free [5] (or violation free in the semantics of Abadi et al. [1]). Publication safety also constrains the behavior of racy programs, as required by the Java Memory Model [21]. Unlike TML and TL2 (the only previous STM system, to our knowledge, that is intrinsically publication safe), [3] NOrec is compatible with closed nesting [28]: one can abort an inner transaction without dooming the outer transaction to abort as well.

Among the various runtimes in the RSTM suite (against which we conducted experiments), only TML and (in some cases) RingSTM have lower instrumentation overhead than NOrec. TML, of course, precludes concurrency among writers, and RingSTM can introduce false conflicts due to Bloom filter collisions. NOrec's single global counter also avoids the performance impact of metadata-induced cache pressure. In short, for systems of modest size, NOrec combines very low time and space overheads with very clean semantics in a very simple package—a winning combination.

This paper makes the following contributions: the design of the NOrec system (Section 2); a description of its semantic properties and their importance for applications with publication and privatization (Section 3); a characterization of its performance (Section 4); and the recognition of its particular appeal in certain special contexts: in conjunction with closed nesting, within the operating system, or as a fall-back for "best effort" hardware TM (Section 5).

[1] In RSTM v.5 [31], it is named "Precise" in recognition of its lack of false conflicts.

[2] In the terminology of Menon et al. [25], the version of NOrec presented in Section 2.2 is ALA publication safe. We present a variant in Section 3.3 that is SLA publication safe.

[3] As before, this refers to ALA publication safety.

2. A Fast TM

NOrec combines three key ideas: (1) a single global sequence lock—shared with our simpler Transactional Mutex Lock (TML) system [36]; (2) an indexed write set, as in our work on contention management for word-based, blocking TMs [35]; and (3) value-based conflict detection, as in the JudoSTM of Olszewski et al. [29]. The resulting TM is both fast at low thread counts and surprisingly scalable—even on multi-chip machines.

2.1 Design

In a future where hardware TM is common on many-core chips, STM will be important primarily in smaller and legacy systems, and as a software fall-back when hardware resources are exhausted. In these contexts an algorithm's instrumentation overhead is at least as important as its scalability. We describe the design of NOrec by starting with the lowest overhead STM algorithm we know of, and adding the minimum overhead needed for satisfactory scalability.

Single Global Sequence Lock Our minimum overhead algorithm is the Transactional Mutex Lock (TML) [36],[4] which uses a global sequence lock [17] to serialize writer transactions. A sequence lock resembles a reader-writer lock in which a reader can upgrade to writer status at any time, but must be prepared to restart its critical section if one of its peers upgrades.

The primary advantage of a sequence lock over a traditional reader-writer lock is that readers are invisible, and need not induce the coherence overhead associated with updating the lock data structure. The primary disadvantage is that doomed readers may be active at the same time as a writer, and may read inconsistent values from memory, leading to potentially erroneous behavior, especially in pointer-based algorithms. Some problems can be avoided by (typically manual) sandboxing, but using the lock in this way is error prone, and still does not solve problems related to memory deallocation. Sequence locks are used in the Linux kernel to protect read-mostly structures where dynamic memory allocation is not required; traditional reader-writer locks and read-copy-update (RCU) [24] are used in other cases.

TML integrates the concept of a single global sequence lock with eager conflict detection and in-place updates. The STM write barrier acquires the lock for writing, the read barrier checks the lock version to ensure consistency, and the STM's built-in memory management system handles any dangerous deallocation situations. The result is a very low overhead STM that is highly scalable in read-mostly workloads, as seen in our performance results (Section 4). The simplicity of the resulting algorithm facilitates compiler optimization [36], but the results presented here do not consider these.

A side effect of having a single lock is that the resulting algorithm, while blocking, is clearly livelock free. This property is preserved through the following modifications, which we make to improve scalability.

There are two impediments to scaling in TML. First, the eager, in-place nature of the algorithm, when combined with its single lock, means that only one writer can ever be active at a time. Second, invisible readers must be extremely conservative and assume that they may have been invalidated by any writer.

Lazy/Redo Our first extension is to use lazy conflict detection and a redo log, for concurrent speculative writers. Updates are buffered in a write log, which we must search on each read to satisfy possible read-after-write hazards. We use a linear write log indexed by a linear-probed hash table with versioned buckets to support $O(1)$ clearing—a structure that was shown to scale well in our work on word-based contention management [35]. Writing transactions do

[4] This algorithm is equivalent to the unpublished *M4* algorithm described in a 2007 patent by Dice and Shavit [8].

not attempt to acquire the sequence lock until their commit points, allowing speculative readers and writers to proceed concurrently.

The primary benefit of this extension is to shrink the period of time that a writer holds the lock, increasing the likelihood that concurrent read-only transactions will commit.

Value-Based Validation We would also like to allow transactions, both readers and writers, to detect if they have *actually* been invalidated by a committing writer, rather than making a conservative assumption. The typical detection mechanism associates transactional metadata with each data location: word-based systems usually use a table of *ownership records* (orecs). Most of the complexity in traditional STMs is in correctly and efficiently maintaining these orecs. The read barrier in a typical "invisible" reader system inspects both the location read and its associated orec, storing the location and possibly information from its orec in a read set data structure. The transaction then re-checks the orecs during validation, possibly comparing to previously saved values, to see whether all its reads remain mutually consistent (i.e., could have occurred at the same instant in time).

The alternative to metadata-based validation is *value-based validation* (VBV), used by Harris & Fraser as a "second chance" validation scheme in their work on revocable locks [15], by Ding et al. in their work on speculative parallelization using process-level virtual memory [10], and by Olszewski et al. in JudoSTM [29]. Rather than logging the address of an ownership recodr, a VBV read barrier logs the address of the location and the value read. Validation consists of re-reading the addresses and verifying that there exists a time (namely now) at which all of the transaction's reads could have occurred atomically.

VBV employs no shared metadata, issues no atomic read-modify-write instructions, and introduces no false conflicts above the level of a word. Fortuitously, our global sequence lock provides a natural "consistent snapshot" capability for validation.

Lazy conflict detection, buffered updates, and VBV allow active transactions to "survive" through a nonconflicting writer's commit. This adds significant scalability to NOrec in workloads where writers are common or transactions are long. The resulting algorithm has one main scalability bottleneck remaining: the sequence lock provides for only a single active committing writer at a time. This is the limitation that would likely make it unsuitable as the primary TM mechanism on a machine with hundreds of cores.

To minimize this commit bottleneck, we arrange for validation to occur *before* lock acquisition, a technique pioneered in RingSTM [34]. Details appear in the following section.

```
volatile unsigned global_lock

local unsigned lock_snapshot
local List<Address, Value> reads
local Hash<Address, Value> writes
```

Listing 1. NOrec Metadata

2.2 Implementation

Metadata NOrec requires little shared metadata, and very little metadata overall (Listing 1). The sequence lock is simply a shared unsigned integer. Each transaction maintains a thread local snapshot of the lock, as well as a list of address/value pairs for a read log, and a hashtable representation of a write set. As is standard, the implementation stores these thread locals, along with a few others (e.g., jump buffers used during aborts), as part of a transaction Descriptor, which is then an explicit parameter to the STM API calls.

Our current implementation logs values as unsigned words. Clients of the TM interface are responsible for appropriate alignment and type modifications, and for splitting operations on larger types into multiple calls to TXRead or TXWrite.

```
unsigned Validate()
1   while (true)
2     time = global_lock
3     if ((time & 1) != 0)
4       continue
5
6     for each (addr, val) in reads
7       if (*addr != val)
8         TXAbort() // abort will longjmp
9
10      if (time == global_lock)
11        return time
```

Listing 2. NOrec Validation

Validation Validation is a simple consistent snapshot algorithm (Listing 2). We start by reading the global lock's version number in line 2, spinning if there is a writer currently holding the lock. Lines 6–8 loop through the read log, verifying that locations still contain the values seen by earlier reads. Lines 10 and 11 verify that validation occurred without interference by a committing writer, restarting the validation if this is not true.

The Validate routine returns the time at which the validation succeeded. This time is used by the calling transaction to update its snapshot value. This mechanism resembles the *extendable timestamps* of Riegel et al. [30], with important differences that we cover in Section 3.2.

```
void TXBegin()
1   do
2     snapshot = global_lock
3   while ((snapshot & 1) != 0)
```

Listing 3. Transaction Begin

TXBegin Beginning a transaction in NOrec simply entails reading the sequence lock, spinning if it is currently held by a committing writer. This snapshot value indicates the most recent time at which the transaction was known to be consistent.

```
Value TXRead(Address addr)
1   if (writes.contains(addr))
2     return writes[addr]
3
4   val = *addr
5   while (snapshot != global_lock)
6     snapshot = Validate()
7     val = *addr
8
9   reads.append(address, value)
10  return val
```

Listing 4. Read Barrier

TXRead Given lazy conflict detection and buffered updates, the read barrier first checks if we have already written this location (lines 1 and 2). If not, we read a value from memory (line 4).

Lines 5–7 provide *opacity* [14] via post-validation. Opacity is crucial for unmanaged (non-sandboxed) languages: it guarantees that even a speculative transaction will never see inconsistent state. Line 5 compares the sequence lock to the local snapshot. If the snapshot is out of date, line 6 validates to confirm that the transaction is still consistent, capturing the returned time as the new local snapshot. Line 7 rereads the memory location and returns to line 5 to try again.

Lines 9 and 10 log the address/value pair for future validation, and return the value read. The reads log is currently an append-only list, allowing us to detect inconsistent reads during validation. An address may appear multiple times in the list. In a data-race-free program, post-validation for opacity guarantees that all entries for a

given location contain the same value. Programs with data races can result in entries with different values—in which case subsequent validation will fail, as it should.

An alternative would be to store the read set in a hashed structure as we do the write set, guaranteeing a unique value for each address. The read barrier would then have two options: (1) always look up the address in the read set first, returning the found value if one exists, or (2) always read the actual location first, and abort if an inconsistent value exists in the read set. The first option can be considered optimistic, in the sense that at the time of the read a conflict may exist that will be "fixed" by some future committing transaction that restores the earlier value. The optimistic option can tolerate this temporary conflict and commit successfully, where the second, pessimistic option must abort.

We expect that the lower constant overhead of the list combined with its simpler read barrier logic will result in better performance in most applications. It also greatly simplifies closed nesting implementations (Section 5.1). The most compelling reason to use a hashed set is to accommodate programs in which locations are reread frequently and validation is also frequent; here the list has validation cost proportional to the number of reads performed, while the hash is proportional to the number of locations read.

```
void TXWrite(Address addr, Value val)
1    writes[addr] = val
```

Listing 5. Write Barrier

TXWrite The write barrier (Listing 5) simply logs the value written using a simple hash-based set. Details of the set implementation are given elsewhere [31, 35].

```
void TXCommit()
1    if (read−only transaction)
2      return
3
4    while (!CAS(&global_lock, snapshot, snapshot + 1))
5      snapshot = Validate()
6
7    for each (addr, val) in writes
8      *addr = val
9
10   global_lock = snapshot + 2 // one more than CAS above
```

Listing 6. Transaction Commit

TXCommit All transactions enter their commit protocol (Listing 6) with a snapshot of the sequence lock, and are guaranteed, due to post-validation in the read barrier (Listing 4, Lines 5-7), to have been consistent as of that snapshot. We exploit this property in both the read-only and writer commit protocol.

A read-only transaction linearizes at the last time that it was proven consistent, i.e., snapshot time. No additional work is required at commit for such transactions (lines 1 and 2 of Listing 6).

A writer transaction will attempt to atomically increment the sequence lock using a compare-and-swap (CAS) instruction, using its snapshot time as the expected prior value. If this CAS succeeds, then the writer cannot have been invalidated by a second writer: no further validation is required. A failed CAS indicates a need for validation because a concurrent writer committed. Line 5 in Listing 6 performs this validation and moves the snapshot forward, preparing the writer for another commit attempt. As in RingSTM, transactions never hold the commit lock while validating, minimizing the underlying serial bottleneck of single-writer commit.

3. Semantics

If transactions were used to protect all accesses to shared data, most researchers would expect TM to display what the database

```
initially  p == null, published == false
     T1:                    T2:
1    p = new foo()          atomic {
2    atomic {                 if (published)
3      published = true         val = p−>x
4    }                      }
```

Listing 7. Publication

```
initially  p != null, published == true
     T1:                    T2:
1    atomic {               atomic {
2      published = false      if (published)
3    }                          val = p−>x
4    p = null               }
```

Listing 8. Privatization

community calls *strict serializability*: in every program execution, transactions would appear to occur, each atomically, in some global total order that is consistent with program order in every thread. Complications arise when we introduce the notions of *publication* [25] and *privatization* [18, pp. 22–23] [32] [37, pp. 6–7]. Because transactions—particularly software transactions—are likely to incur nontrivial cost, programmers are motivated to avoid their use when accessing data that are known to be private to a given thread at a particular point in time. It seems reasonable to assume, for example, that newly allocated data can safely be initialized outside a transaction before being made visible to other threads (*published*—see Listing 7). Similarly, if a thread excises data from a shared structure, or otherwise makes them inaccessible to other threads (*privatized*—see Listing 8), it seems reasonable that the thread could safely access the data nontransactionally thereafter.

Unfortunately, since the purpose of private accesses is to avoid examining or modifying transactional metadata, a naive STM implementation may allow these accesses to appear "out of order" in the presence of publication and privatization, violating both intuitive and formal notions of transactional semantics. In this section we review the notions of *publication* and *privatization safety*, and discuss how they are supported in NOrec.

3.1 Publication and Privatization Safety

In standard parlance, two memory accesses are said to *conflict* if they occur in different threads, they touch the same location, and at least one of them is a write. A *memory consistency model* (or just "memory model") defines a "happens before" order $<_{hb}$ on accesses in the execution of a parallel program. This order determines which writes are permitted to be seen by a given read. Typically synchronization accesses (e.g., lock acquire and release operations) are ordered across threads, all accesses are ordered within a thread, and $<_{hb}$ is the (partial) transitive closure of these inter-thread and intra-thread orders.

If all conflicting accesses in an execution are ordered by $<_{hb}$, then the value seen by every read will be determined by a unique most recent write. If, however, there exists a sequentially consistent execution in which some conflicting accesses are not ordered, then the program is said to have a *data race*. There may be more than one most recent write for a given read, or there may be incomparable writes. In some memory models (e.g., that of Java [21]), a read is required to return the value from one of these writes. In other memory models (e.g., that of C++ [3]), program behavior in the presence of a data race is undefined. Grossman et al. [13] were the first to explore memory models specifically for TM. The discussion here follows their framework closely.

Strict serializability provides a natural basis for TM memory models: transactions are globally totally ordered, accesses within each thread are ordered, and these two orders are mutually con-

```
initially  n == 0, published == false
     T1:                    T2:
1    n = 1                  atomic {
2    atomic { }               v = n
3     published = true        f = published
4                            }
```

Listing 9. Publication via empty transaction

sistent, so their transitive closure induces a global partial order on all memory accesses. This partial order turns out to be just what one would expect in a simplistic (nonspeculative) TM implementation based on a single global lock [5, 18, 25, 33]. Stated a bit more precisely, in the style of memory models, a TM system is said to provide *single lock atomicity (SLA)* if, for every program execution, there exists some global total order on transactions that is consistent with program order, and that when closed with program order produces a happens-before order that explains the program's reads.

Unfortunately, many STM systems fail to provide SLA, because private accesses do not follow the metadata protocols of the STM system. It is common, for example, for a transaction to perform "cleanup" operations (e.g., write-back from a log) after it has committed. If some other transaction privatizes data that have not yet been cleaned up, subsequent private reads may see the wrong values, and private writes may be lost. Similarly, it is possible for private writes to change the values of data that are still being read by a transaction that is doomed to abort but has not yet realized this fact. If care is not taken, such a transaction may suffer logically impossible errors: division by zero, segmentation fault, infinite loop, jump to an invalid or non-transaction-safe code address. (For a discussion of techniques to support SLA in the presence of privatization, see our paper at ICPP'08 [23].)

In the case of publication, most STM systems permit executions that are forbidden by SLA semantics, at least with Java-like conventions on races. Many examples can be found in a 2008 paper by Menon et al. [25], which first identified the "publication problem." In Listing 9, for example (adapted from Figure 9 of that paper), there is no conflict between the transactions in T1 and T2, and many STM systems will allow them to execute concurrently (or even elide T1's, since it is empty). In this case, however, all of T1's code may execute between lines 2 and 3 of T2, leading to a result (v == 0, f == true) that is forbidden by SLA.

Interestingly, as Menon et al. point out [25], ensuring SLA semantics in the presence of publication is a challenge only for programs with data races. In Listing 9, SLA permits all of T2 to fall between lines 1 and 2 of T1, in which case T1's write of n is unordered with respect to T2's read. This asymmetry with respect to privatization (where problems occur even in the absence of races) occurs because only one thread (the one that currently enjoys private access) has the right to perform publication, while any thread has the right to privatize. In this regard, privatization resembles the acquisition of a lock; publication resembles its release.

Rather than force STM implementations to provide SLA semantics for programs with data races, Menon et al. propose a series of semantics that progressively relax the transactional orderings required by SLA. Their DLA (disjoint lock atomicity) orders only those transactions that conflict. This allows a Java-like system to display the ⟨v == 0, f == true⟩ outcome in Listing 9. Their ALA (asymmetric lock atomicity) permits similarly counter-intuitive results in the event of an anti-dependence between the publisher's transaction and some subsequent transaction in another thread. Finally, their ELA (encounter-time lock atomicity) permits counter-intuitive results even in the case of forward (flow) dependences, if the compiler hoists reads outside of conditionals (effectively performing a speculative prefetch that creates a data race not found in the original program).

Of these various relaxations, we find ALA the most compelling: it permits all the classic compiler optimizations within transactions, while still providing behavior indistinguishable from SLA in programs that publish only by forward dependences. Stated a bit more precisely, a TM system system is said to provide *asymmetric lock atomicity (ALA)* if, as in SLA, there exists for every program execution a global total order on transactions $<_t$ that induces a happens-before order $<_{hb}$ that explains the program's reads. In this case, however, we need (as in Java [21]) an intermediate "synchronizes with" order $<_{sw}$ to complete the explanation. Specifically, for transactions T1 and T2, we say T1 $<_{sw}$ T2 if T1 $<_t$ T2 and there is a forward data dependence from T1 to T2. Then $<_{hb}$ is the transitive closure of $<_{sw}$ and program order.

Note that the code in Listing 7 will behave the same under SLA and ALA semantics, even if the compiler hoists the read of p above line 2 in T2. (Standard sequential optimizations will not hoist the read of x, since dereferencing p will not be known to be safe.) Note also that Menon et al. define SLA, ALA, etc. in terms of equivalence to locks. We believe the ordering-based definitions here to be equivalent to theirs.

Summary An STM system is said to be *publication* (*privatization*) *safe* with respect to a given memory model if it adheres to that model even in the presence of publication (privatization). Privatization safety tends to be a challenge for STM under any memory model. An implementation that supports privatization must ensure that a thread does not access a datum privately until previously committed transactions have finished writing it, and (in the absence of sandboxing) until any still active but doomed transactions have finished reading it.

Publication safety is a challenge for STM under a memory model that requires well-defined behavior for programs with source-level data races, or that allows the compiler to apply common sequential optimizations to code within transactions, even under a memory model in which source-level data races have undefined behavior. In the former, Java-like case, the implementation needs to be SLA publication safe. In the latter, C++-like case, the implementation needs to be ALA publication safe.

3.2 Existing STM Systems

SLA semantics can be implemented, trivially, with a single global lock, at the expense of all concurrency. Menon et al. propose *start linearization* as a more scalable mechanism to provide the same semantics. Simply stated, start linearization ensures that successful transactions commit and clean up (finalizing any buffered state) in the order in which they started.

In Listing 8, cleaning up in commit order ensures that post-privatization reads never see stale values, and post-privatization writes are never overwritten by stale values. Sandboxing or, alternatively, validation on each transactional read can be used to avoid erroneous behavior in doomed transactions.

In Listing 9, if T2's transaction begins before T1's, committing in start order avoids the dangerous ordering alluded to in Section 3.1, by forcing T1 to wait on line 2 until T2 commits. This prevents the read of published from returning true. Though transactions must complete in the order they start, they achieve scalability through pipelined overlap.

ALA semantics are easier than SLA to implement, because ALA transactions are ordered only by forward dependences. In particular, a read-only (or empty) transaction cannot publish data in an ALA system because it cannot be the source of a flow dependence.

An ALA-publication-safe TM system allows the compiler or (with Java-like conventions) the programmer to create a race for published data, so long as there is a flow dependence from a transaction that follows the write in the publishing thread to a transaction that performs the read in another thread. In Listing 7, for example,

if T2 reads true on line 2, ALA publication safety requires that T2's read of p return the value written by T1 in line 1 (or else that T2 abort), *even if the read of* p *was hoisted above line 2*.

Menon et al. describe a variant of Dice et al.'s TL2 [7] as the canonical example of an ALA-publication-safe implementation. Update times associated with ownership records allow a transactional read to detect when the write it is "seeing" may have been performed after the transaction's own start time. In this case, the transaction pessimistically assumes that some previous read returned a now-invalid write, and aborts. As in the SLA implementation described above, privatization safety is handled by forcing transactions to clean up in the order they committed.

Interestingly, the *extendable timestamps* of TinySTM [30] break ALA publication safety for Java-like languages, or for C++-like languages in which the compiler can introduce data races. In a race-free program, extendable timestamps avoid unnecessary aborts by allowing a transaction that sees a recently-written value to update its "start" time, verify that no previously read location has been updated since the old start time, and continue. Unfortunately, in a program with racy publication, such a transaction may fail to notice that a previously read location has been updated nontransactionally.

3.3 Ordering Semantics for NOrec

Like TinySTM, NOrec attempts to validate its read set and continue when it detects a possible conflict. Validations may happen more often in NOrec, because detection is based on a global sequence lock rather than on per-orec timestamps. Crucially, however, value-based validation means that NOrec is able to detect the nontransactional writes involved in racy publication.

Consider Listing 7 again, where the read of p in T2 has been hoisted above the read of published. If T1 starts its transaction after T2, but commits before T2 reads published, then TL2, TinySTM, and NOrec will all detect a possible conflict when the read of published occurs. TL2 will abort and restart T2, ensuring that even a hoisted read of p will see the value written by T1 at line 1. TinySTM and NOrec will validate their read sets (which consist solely of the hoisted p). Since T1's nontransactional write of p will not have modified p's orec, TinySTM will conclude that T2 is consistent, even if its value of p is actually stale. NOrec, on the other hand, will have logged both the address and the value of p. It will fail validation if the logged value (NULL) does not match the current value (written by T1 at line 1). In this way, NOrec realizes exactly the transactional schedules of TinySTM that are ALA publication safe.

The base NOrec algorithm is clearly not SLA safe: In Listing 9, neither transaction writes a shared location, thus neither transaction ever validates and T2 will not detect possible inconsistent reads of n and published. T2 is capable of detecting the inconsistency, however: validation would fail if it occurred. If we force every transaction to validate at least once after its final read, NOrec becomes SLA publication safe. Perhaps the simplest strategy would be to force every transaction to increment the sequence lock, but this would induce unnecessary serialization and cache contention. We could instead have every transaction validate at commit time if it has not done so since its last read, but this requires a local flag that increases (slightly) the instrumentation overhead on reads. We have obtained the best performance simply by having every transaction validate unconditionally (and possibly redundantly) at commit time.

NOrec is inherently SLA privatization safe due to its single-writer commit implementation. Both TL2 and TinySTM require an additional mechanism to be privatization safe. The results reported in Section 4 use either a table of activity flags, as in Menon et al.'s canonical ALA system, or the two-counter quiescence mechanism described in our previous work on ordering-based semantics [33].

4. Performance Results

Our performance goals are twofold: low overhead and reasonable scalability. We use the RSTM RBTree microbenchmark [31] to isolate performance in specific circumstances, and the STAMP benchmark suite [26] for overall performance in larger programs.

All results were collected on a Sun Niagara2, a machine with 8 cores on a single chip, and 8 hardware threads per core, for a total of 64 concurrent hardware contexts. Cores are relatively simple—all branches are predicted as not taken,[5] and there is no out-of-order execution. Each core shares an 8KB, 4-way associative L1 data cache among its 8 thread contexts. This structure is optimized for throughput computing, and accentuates the cost of transactional instrumentation. Benchmarks were compiled with gcc 4.4.2 using -O3 settings and built-in thread-local storage.

4.1 Overhead

We measure overhead using single-threaded RBTree executions in four different configurations: with small (64 node) and large (65K node) trees, and with read-mostly and balanced workloads (Figure 1). Small trees result in small transactions, with relatively small write sets due to rotation. Large trees result in longer transactions with significantly larger read and write sets. Results for the RBTree benchmark are the average of five runs, where each run consists of a single thread executing three million pseudo-random transactions obtained from the same initial seed, thus all tests do the same amount of useful work.

We compare 7 different STM systems, all of which are distributed as part of the RSTM package [31]. These are full-fledged STM implementations that support subsumption nesting as well as inevitability, both of which introduce branches onto the common instrumentation path. The systems have been optimized for the Niagara2 system in two important ways: (1) we have carefully controlled the inlining of transactional instrumentation to prevent damaging instruction-cache behavior, and (2) we have structured the source code in order to minimize the number of costly "taken" branches on common code paths.

CGL is a coarse-grained locking implementation in which all transactions acquire and release a single global lock. *SGLA* is a good-faith reproduction of the SGLA algorithm presented by Menon et al. [25], except that instead of relying on sandboxing to protect against erroneous behavior in doomed transactions, we poll a global commit counter in the read barrier, and validate when the counter has changed. *ALA*, known as "Flow" in the RSTM suite, is an optimized implementation of TL2 [7] with counter-based quiescence added for privatization safety. *TML*, *NOrec*, and *NOrec SLA* are as described in Section 2.2.

Unsafe is a highly scalable orec-based system descended from TL2, but using TinySTM's extendable timestamps [30], a hashed write set, and a "patient" contention management policy that waits for conflicting transactions that are actively committing. As described in our 2009 PPoPP paper, Unsafe scales extremely well, and is essentially livelock-free [35]. The name we give it here reflects the fact that this system is *neither publication nor privatization safe*; we include it as a rough upper bound on potential scalability. In the RSTM suite, Unsafe is known as "Fair".

We present transaction throughput results in Figure 1. These are normalized to the results of running the benchmark without transactional instrumentation. Using hardware performance counters, we also recorded metrics such as instruction and data cache misses at both the L1 and L2; TLB usage; and overall instruction, atomic-op,

[5] The lack of branch prediction results in performance that is highly dependent on code layout. We have done our best to minimize the number of taken branches in common case scenarios in the code, preferring read-only transactions where possible.

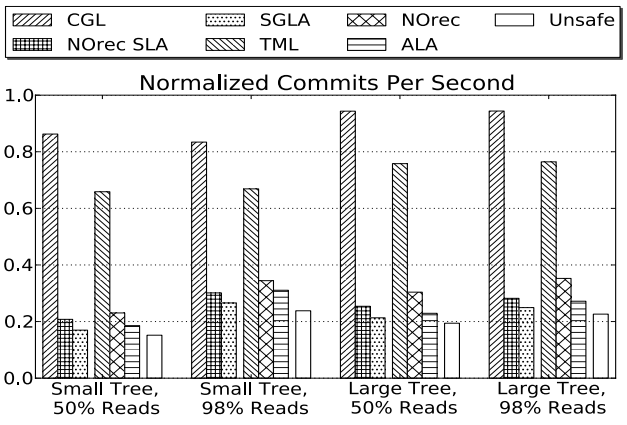

Figure 1. Performance results for single-threaded execution, normalized to an uninstrumented baseline. Bars are clustered in three groups, CGL, NOrec SLA, and SGLA provide SLA semantics, while TML, NOrec, and ALA provide ALA semantics. Unsafe provides no privatization safety and is thus in a class of its own.

and taken-branch counts. These numbers were all consistent with the results presented in Figure 1 (after modifying code layout to avoid instruction cache misses and taken branches).

The overhead of CGL is one atomic compare-and-swap per transaction, which can be nontrivial for small transactions, but is amortized in larger transactions. TML can elide this for read-only transactions, but requires a volatile load and branch per transactional memory operation—a large cost that cannot be amortized.

NOrec is consistently faster than the canonical ALA implementation by a small amount. In a single-threaded workload these algorithms have similar behavior. The main extra source of overhead in ALA is that it loops at commit in order to acquire each of its orecs with an atomic instruction, leading to extra atomics, larger instruction counts, and more taken branches than NOrec. This more than balances the fact that NOrec has slightly larger cache pressure than ALA, as the tested configurations do not result in substantial data cache misses. NOrec SLA is faster than the SGLA implementation by similar amounts, for similar reasons.

Write buffering and logging for validation have clear overheads, but these are the price of writer concurrency and privatization safety. When combined with lazy conflict detection, they also eliminate livelock in practice, and dramatically simplify contention management [35].

4.2 Scalability

We use the same RBTree benchmark as in Section 4.1 to assess scalability in the same four benchmark configurations, however we do not test CGL as it does not scale. The OS scheduler is configured to place threads round-robin on the cores within a chip, so the first 8 threads are on separate cores, the 9th is back on the first core, and so on. Executing a fixed number of transactions per thread leads to load imbalance, so we instead run each configuration for five seconds. All thread counts between 1 and 64 are measured directly. Figure 2 summarizes these results.

We expected TML to perform well for read-mostly workloads. We were somewhat surprised to see its resilience in conditions where there are many writers as well. Even in the 50% writer case, TML outperforms all alternatives on our microbenchmark with fewer than 4–8 active threads. Our STAMP tests (Section 4.3) show this same pattern of behavior.

Of the privatization-safe implementations that admit write-write and read-write concurrency (and thus have better scalability), the NOrec options perform the best in most of our tests, even out to 64 threads. Their scalability and throughput, in fact, are better than Unsafe in 3 of the 4 experiments, even though we selected Unsafe for its high scalability. This is perhaps not as surprising as it first appears: Given a set of concurrent active transactions, NOrec and Unsafe allow the same potential commit orderings, while SGLA and ALA are more restrictive, for the sake of publication safety. NOrec has lower fixed and per-location overheads, but Unsafe, given its lack of privatization safety, allows its writebacks to occur in parallel, and its completions to occur in any order.

As expected, SGLA scales poorly overall. Its pipeline-style concurrency is quickly swamped by the probability that most transactions are waiting on transactions with earlier start-linearization stamps. In addition L1 data cache misses are higher due to its reliance on quiescence tables that are always written remotely (see Figure 3 for details). The behavior of ALA is similarly disappointing. Its pessimistic publication orderings limit its scalability in read-mostly circumstances, while its privatization safety becomes a bottleneck when writers are more common.

The NOrec TMs show very little separation in workloads with many writes. The primary difference between the algorithms is that NOrec can avoid validation on commit, while NOrec SLA validates every transaction. In workloads where roughly 50% of the transactions are writers this additional validation costs virtually nothing, since it is likely that all transactions will validate at least once. Read-dominated workloads, however, result in noticeable separation between the algorithms: NOrec may commit many transactions without any validation, while NOrec SLA must validate every transaction at least once. In the large-tree, read-mostly case (top right graph in Figure 3), NOrec SLA suffers approximately 20 extra cache misses per transaction due to this validation.

NOrec has two main weaknesses. It has a single-writer commit protocol, which clearly limits its scalability in workloads with many writers. This is a fundamental property of the underlying algorithm, and is independent of system architecture. The Niagara2 clearly demonstrates NOrec's second weakness: its read set entries are twice as large as the corresponding orec-based implementations. Where Unsafe, ALA, and SGLA all store the address of an orec in their read sets, NOrec and NOrec SLA must store *both* the address of the read location and the value that was seen.

During validation, each entry in the read set is brought into the L1 cache. NOrec will begin to suffer excessive L1 cache evictions before the orec-based algorithms, which is exactly what we see in our large trees with frequent writers configuration (Figure 3). RBTree rotations in this configuration read large numbers of locations, enough to trigger evictions in the small L1 cache shared by threads on a given core. The fact that each committing transaction triggers a validation merely serves to exacerbate the situation in this write-heavy configuration.

In configurations where NOrec does not suffer from this L1 miss pathology we see the expected L1 data cache behavior. SGLA must access two quiescence tables that are written by remote threads, giving it the largest cache-miss profile. Unsafe and ALA use similar timestamp schemes and thus have similar shapes, but Unsafe's extendable timestamps, while making it less pessimistic about ALA safety, also induce more cache misses per successful transaction. TML, NOrec, and NOrec SLA have better data locality than their orec-based cousins, resulting in the lowest overall miss rates. Other performance counter results show that L1 data cache misses are the primary cause of lost scalability specific to the Niagara2.

4.3 STAMP

In 5 of the 8 STAMP benchmarks (Figure 4: intruder, ssca2, yada, labyrinth, kmeans), NOrec and NOrec SLA are either the fastest or roughly tied for fastest among the various runtimes, across the full range of thread counts. In genome, only Unsafe outperforms

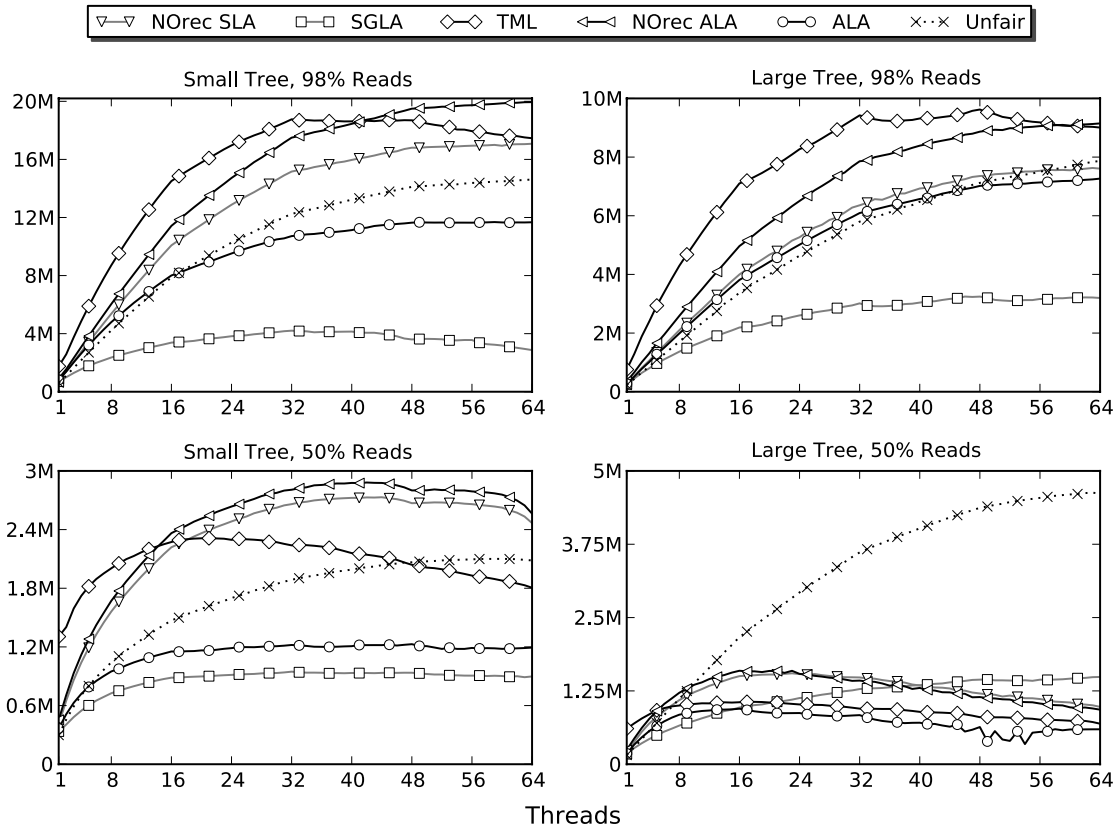

Figure 2. Microbenchmark scalability results. The Y axis shows commits per second: higher is better. NOrec SLA and SGLA support SLA semantics. TML, NOrec ALA, and ALA support ALA semantics. Unsafe provides no publication safety.

them. As in the RBTree experiments, these results demonstrate remarkable scalability for NOrec, and suggest that its privatization and publication safety are not a significant obstacle to throughput.

In general, SGLA performs better in STAMP than it does in RBTree. The main exception is intruder, which more closely resembles the RBTree results. Unsurprisingly, SGLA outperforms the NOrecs in vacation, whose principal shared data structure is in fact a red-black tree, presumably due to a relatively high occurrence of large writer transactions.

Base (ALA) NOrec outperforms NOrec SLA by a significant amount in intruder, but in most other benchmarks the two are very nearly tied. NOrec SLA appears to outperform NOrec in bayes, but this benchmark is nondeterministic, and displays irregular performance; it's hard to draw firm conclusions.

The clear loser here is TML: for workloads with many large, conflicting transactions, its lack of writer concurrency prevents it from scaling acceptably.

5. Special Uses

The distinctive properties of NOrec make it appealing for several special purposes. In this section we describe three such purposes, which we have begun to explore as part of ongoing work.

5.1 Closed Nesting

Closed nesting, where inner nested transactions can abort and restart separately from their outer parents, is expected to improve performance in certain applications [28]. Sadly, TL2, the prior canonical ALA-publication-safe STM, does not support closed nesting. Any read that causes an inner transaction to abort, because the corresponding orec has a recent timestamp, will also necessarily cause outer transactions to abort, since their start times can be no later than those of their children.

Both the ALA and SLA versions of NOrec appear well suited to closed nesting. We would expect to introduce a scope data structure that stores information necessary to detect and handle an inner nested transaction abort.

NOrec's append-only read log makes it simple to detect the proper nesting level at which to abort a transaction. As we enter each scope we can record the tail pointer of the log. Validation then scans the log in order. If it finds an invalid read, the record of tail pointers indicates the nesting level at which to start aborting, and the point at which to truncate the log.

The write set is more difficult. A nested abort requires that we restore the set of writes that was active at the start of the nested transaction. An indexed write set complicates this process, as we would normally maintain only the unique, latest write to any location in the set. Closed nesting requires that we maintain a unique, latest write per nested scope.

There are several ways to implement such an augmented indexed write set. The simplest is to make the write set an append-only log, much like the read set, with an index for fast lookup of the most recently written value for an address. A nested abort can truncate both logs, then re-scan the write log to rebuild the index.

A slight variant of this technique would overwrite the most recent entry for a given location in the write set, if it is being written at the same nesting level. This trades potential improved space overhead when the same location is written multiple times for

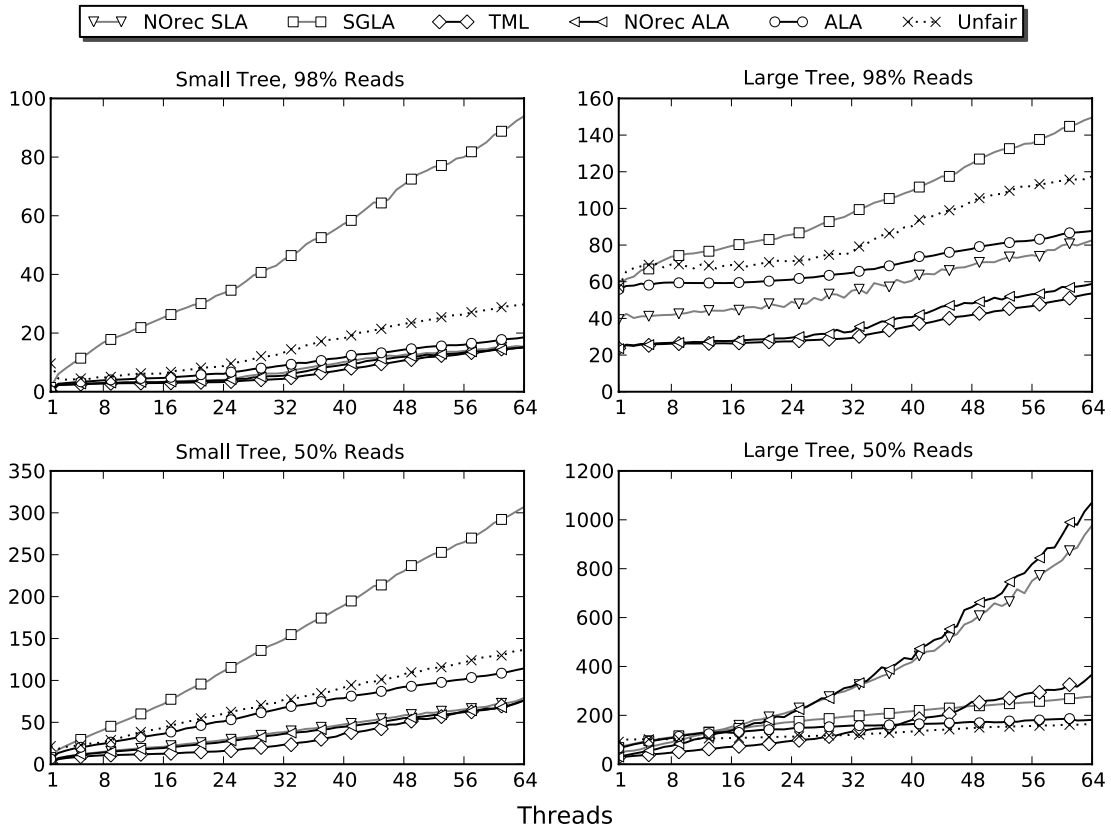

Figure 3. L1 Data cache misses per commit.

an additional branch on each write barrier. We could also maintain the information needed to undo each write, in order to patch the index rather than rebuild it, or we could store a forwarding pointer as part of each log entry. The latter option would allow the write barrier to lazily detect an individual invalidation and patch itself piecemeal by finding the corresponding active value. Finally, we could keep a stack of scope/value pairs at each log location and scan through the log during abort, popping stacks as necessary. The choice among these options will depend on further experiments, and may be workload dependent.

5.2 OS Safety

NOrec is appealing as a kernel-level STM for several reasons, including low space overhead, a narrow window when locks are held, and a lack of false conflicts due to orec or Bloom filter hash collisions. As in most STMs, however, we must deal with the possibility that a doomed transaction may read a datum that has been deallocated by a concurrent transaction. In particular, if the datum's page is removed from the address space of the doomed transaction, a segmentation fault may occur. In user code, this problem can be avoided with nonfaulting loads [7], garbage collection, a custom signal handler [11], or epoch-based deferred memory reclamation [12, 16]. In the OS, the latter option is most common, but typically defers reclamation for extended periods (as in read-copy-update (RCU) algorithms [24]).

RCU employs extended epochs because there is generally no precise way to tell when no outstanding references to a location remain. In TM, it suffices to ensure that every transaction that was active during T2's execution has either committed or aborted. However, within the OS there is no clear bound on the number of possibly active threads, especially if a hardware interrupt can trigger a new transaction.

To address the problem, we require a solution to the long-lived k-renaming problem [27]. One simple heuristic approach is to maintain a table of integers (1024, say), each initialized to $\perp$. When an OS transaction begins, it randomly probes the table until it finds an entry that equals the initialized value and atomically sets the value to its cache of the NOrec sequence lock. At commit time, the transaction resets its entry to $\perp$. This table can be used in the exact same manner as the static tables of Hudson [16] and Fraser [12]; it differs only in that it requires an extra CAS at the beginning of each transaction to select an epoch slot.

5.3 Hardware Integration

Future multicore processors (e.g., Sun's Rock [4]) are likely to provide some form of best-effort hardware transactional memory, in which overflow of hardware resources causes fall-back to STM. Interaction with traditional STM implementations is complicated by the need for the hardware to be aware of and interact with the STM metadata. In HyTM, for example [6], read and write barriers are inserted in hardware transactions in order to inspect orecs. This bloats both the number of instructions executed and the cache and read set footprint of the hardware transaction, increasing its likelihood of aborting. In addition, the STM uses semivisible readers to communicate with the HTM, a potential performance bottleneck. A more recent variant, SkySTM [20], incorporates several performance improvements, but still faces the same basic issues.

NOrec has only a single word of shared metadata for an HTM implementation to contend with—the global sequence lock. We can leverage this fact to build a hybrid TM system in which hardware