Optimizing Hybrid Transactional Memory:  
The Importance of Nonspeculative Operations

Torvald Riegel 1  Patrick Marlier 2  Martin Nowack 1  Pascal Felber 2  Christof Fetzer 1

1 Technische Universität Dresden, Germany  
2 Université de Neuchâtel, Switzerland

Nov 5, 2010

An earlier version of this report has appeared as RR-I-10-08.1 (August 1, 2010). A brief announcement was published at the International Symposium on Distributed Computing (DISC 2010) under the title: Hybrid Time-based Transactional Memory. This report is also available as RR-I-10-11.1 from Université de Neuchâtel.
Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations

Torvald Riegel  Martin Nowack
Christof Fetzer
Technische Universität Dresden
\{firstname.lastname\}@inf.tu-dresden.de

Patrick Marlier  Pascal Felber
Université de Neuchâtel
\{firstname.lastname\}@unine.ch

Abstract

Transactional memory (TM) is a speculative shared-memory synchronization mechanism used to speed up concurrent programs. Most current TM implementations are software-based (STM) and incur noticeable overheads for each transactional memory access. Hardware TM proposals (HTM) address this issue but typically suffer from other restrictions such as limits on the number of data locations that can be accessed in a transaction.

In this paper, we present several new hybrid TM algorithms that can execute HTM and STM transactions concurrently and can thus provide good performance over a large spectrum of workloads. The algorithms exploit the ability of some HTMs to have both speculative and nonspeculative (nontransactional) memory accesses within a transaction to decrease the transactions’ runtime overhead, abort rates, and hardware capacity requirements. We evaluate implementations of these algorithms based on AMD’s Advanced Synchronization Facility, an x86 instruction set extension proposal that has been shown to provide a sound basis for HTM.

1. Introduction

Today’s multicore and manycore CPUs require parallelized software to unfold their full performance potential. Shared-memory synchronization plays a big role in parallel software, either when synchronizing and merging results of parallel tasks, or when parallelizing programs by speculatively executing tasks concurrently.

So far, most concurrent programs have been programmed using lock-based synchronization. Yet, locks are considered difficult to use for the average programmer, especially when locking at a fine granularity to provide scalable performance. This is particularly important when considering that large classes of programs will have to be parallelized by programmers who are not well trained in concurrent programming. Transactional memory (TM) is a promising alternative for synchronization because programmers only need to declare which regions in their program must be atomic - not how atomicity will be implemented. Unfortunately, current software transactional memory (STM) implementations have a relatively large performance overhead. While there is certainly room left for further optimizations, it is believed by many that only hardware transactional memory (HTM) implementations can have a sufficiently good performance for TM to become widely adopted by developers.

Of the many published HTMs, only two designs have been proposed by industry for possible inclusion in high-volume microprocessors: Sun’s Rock TM [9] and AMD’s Advanced Synchronization Facility (ASF) [1]. While these HTMs have notable differences, they are both based on simple designs that provide best-effort HTM in the sense that only a subset of all reasonable transactions are expected to be supported by hardware. They have several limitations (e.g., the number of cache lines that can be accessed in a transaction can be as low as four) and have to be complemented with software fallback solutions that execute in software the transactions that cannot run in hardware. A simple fallback strategy is to execute software transactions serially, i.e., one at a time. However, this approach limits performance when software transactions are frequent. It is therefore desirable to develop hybrid TM (HyTM) in which multiple hardware and software transactions can run concurrently.

In this paper, we present a family of novel HyTM algorithms that use AMD’s ASF as HTM. These algorithms use two state-of-the-art STM algorithms from different research groups, LSA [22] and NOrec [7], for software transactions. LSA and NOrec focus on different workloads in their optimizations, e.g., a higher level of concurrency vs. lower single-thread overheads. The resulting HyTM compositions provide the same guarantees as the respective STMs.

Most previous HyTM proposals have assumed HTMs in which every memory access inside a transaction is speculative, that is, it is transactional, isolated from other threads until transaction commit and will be rolled back on abort. In contrast, ASF provides selective annotation, which means that nonspeculative memory accesses are supported within transactions and speculative memory accesses have to be explicitly marked as such. We make heavy use of this feature to improve the efficiency of our HyTM algorithms by decreasing the runtime overhead, abort rates, and HTM capacity requirements of hardware transactions.

We evaluate the performance of our algorithms on a near-cycle-accurate x86 simulator with support for several implementations of ASF [5] that differ notably in their capacity limits. Our HyTMs are embedded into a full TM software stack for C/C++.

The rest of the paper is organized as follows. In Section 2, we provide background information about ASF and TM in general, and we discuss related work on HyTM designs. We present our new HyTM algorithms in Sections 3 and 4, evaluate them in Section 5, and conclude in Section 6.

2. Background and Related Work

Our objective is to investigate the design of hybrid transactional memory algorithms that exploit hardware facilities for decreasing the overhead of transactions in good cases while composing well with state-of-the-art software transactional memory algorithms. We assume that the TM runtime system is implemented as part of a library with a well-specified interface for starting, committing, and aborting transactions, as well as performing transactional memory accesses (loads and stores).

We focus on C/C++ and we follow the specification of C++ support for TM constructs [13]. Informally, C++ transactions are guaranteed to execute virtually sequentially and in isolation as long as the program is race-free in terms of the upcoming C++ memory model [14] extended with specific rules for transactions.
Algorithm 1 Common transaction start code for all HyTMs.

1: hytm-start(p);
2: if hytm-disabled(p) then
3: goto line 7;
4: r ← SPECULATE
5: if r ≠ 0 then
6: goto line 4;
7: return false
8: if we are in a software transaction then
9: execute STM codepath
10: restore registers, stack, etc. and retry
11: return true

Common transaction start code for all HyTMs.

Algorithm 1) takes care of selecting STM or HTM code at runtime. A transaction first tries to run in hardware mode using a special ASF SPECULATE instruction (line 4). This instruction returns a non-zero value when jumping back after an abort, similarly to setjmp/longjmp in the standard C library. If the transaction aborts and a retry is unlikely to succeed (as determined on line 6, for example, because of capacity limitations or after multiple aborts due to contention), it switches to software mode. After this has been decided, only STM or HTM code will be executed (functions starting with stm- or htm-, respectively) during this attempt to execute the transaction.

In the rest of this section, we give an overview of the hardware TM support used for our hybrid algorithms and we discuss related work.

2.1 AMD’s Advanced Synchronization Facility

AMD’s Advanced Synchronization Facility (ASF) is a proposal [1] of hardware extensions for x86_64 CPUs. It essentially provides hardware support for the speculative execution of regions of code. These speculative regions are similar to transactions in that they take effect atomically. It has been shown in a previous study [5] that ASF can be used as an efficient pure HTM in a realistic TM software stack. The HyTM algorithms that we present in this paper are based on ASF and rely on a similar software stack.

AMD has designed ASF in such a way that it would be feasible to implement ASF in high-volume microprocessors. Hence, ASF comes with a number of limitations [1, 6, 10]. First, the number of disjoint locations that can be accessed in a transaction is limited either by the size of speculation buffers (which are expensive and thus have been designed with small capacity) or by the associativity of caches (when tracking speculative state in caches). Second, ASF transactions are not virtualized and therefore, abort on events such as context switches or page faults. These limitations illustrate that HyTM will be required to build a feature-rich TM for programmers.

In contrast to several other HTM proposals, ASF provides selective annotation for speculative memory accesses. Speculative regions (SRs, the equivalent of transactions) are demarcated with new SPECULATE and COMMIT CPU instructions. In an SR, speculative/protected memory accesses, in the form of ASF-specific LOCK MOV CPU instructions, can be mixed with nonspeculative/unprotected accesses, i.e., ordinary load/store instructions (MOV) as well as atomic instructions such as compare-and-set (CAS). Selective annotation requires more work on the compiler side, but allows the TM to use speculative accesses sparingly and thus preserve precious ASF capacity. Second, the availability of nonspeculative atomic instructions allows us to use common concurrent programming techniques during a transaction, which can reduce the number of transaction aborts due to benign contention (e.g., when updating a shared counter). In an SR, nonspeculative loads are allowed to read state that is speculatively updated in the same SR, but nonspeculative stores must not overlap with previous speculative accesses. Conflict detection for speculative accesses is handled at the granularity of a cache line.

ASF also provides new CPU instructions for monitoring a cache line for concurrent stores (LOCK PREFETCH) or loads and stores (LOCK PREFETCH), for stopping monitoring a cache line (RELEASE), and for aborting a SR and discarding all speculative modifications (ABORT).

Conflict resolution in ASF follows the “requester wins” policy (i.e., existing SRs will be aborted by incoming conflicting memory accesses). Table 1 summarizes how ASF handles contention when CPU A performs an operation while CPU B is in a SR with the cache line protected by ASF [1]. These conflict resolution rules are important for understanding how our HyTM algorithms work and why they perform well.

The ordering guarantees that ASF provides for mixed speculative and nonspeculative accesses are important for the correctness of our algorithms. In short, aborts are instantaneous with respect to the program order of instructions in SRs. For example, aborts are supposed to happen before externally visible effects such as page faults or nonspeculative stores appear. A consequence is that memory lines are monitored early for conflicting accesses (i.e., once the respective instructions are issued in the CPU, which is always before they retire). After an abort, execution is resumed at the SPECULATE instruction. Further, atomic instructions such as compare-and-set or fetch-and-increment retain their ordering guarantees (e.g., a CAS ordered before a COMMIT in a program will become visible before the transaction’s commit). This behavior illustrates why speculative accesses are also referred to as “protected” accesses.

2.2 Previous HyTM Designs

Kumar et al. describe a HyTM [21] based on an HTM with eager conflict detection, support for selective annotation, and requester-wins conflict resolution. Unlike our approach, their STM has an object-based design with indirection via locator objects, uses visible reads, and requires small hardware transactions even for software transactions. Recent research has shown that STM algorithms with invisible reads and no indirection have significantly lower overhead (e.g., [7, 8, 18]).

Damron et al. present a HyTM [19] that combines a best-effort HTM with a word-based STM algorithm that uses visible reads and performs conflict detection based on ownership records. The HTM does not use selective annotation and thus hardware transactions have to monitor application data and TM metadata (i.e., ownership records) for each access, which significantly increases the HTM ca-

### Table 1. Conflict matrix for ASF operations (from [1], §6.2.1).

<table>
<thead>
<tr>
<th>CPU A mode</th>
<th>CPU A operation</th>
<th>CPU B cache line state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speculative region</td>
<td>LOCK MOV (load)</td>
<td>OK</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK MOV (store)</td>
<td>B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK PREFETCH</td>
<td>OK</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK PREFETCH</td>
<td>B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK MOV (load, stores)</td>
<td>B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK MOV (loads, stores)</td>
<td>B aborts</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK PREFETCH</td>
<td>OK</td>
</tr>
<tr>
<td>Speculative region</td>
<td>LOCK PREFETCH</td>
<td>B aborts</td>
</tr>
<tr>
<td>Any</td>
<td>Read operation</td>
<td>OK</td>
</tr>
<tr>
<td>Any</td>
<td>Write operation</td>
<td>B aborts</td>
</tr>
<tr>
<td>Any</td>
<td>Prefetch operation</td>
<td>OK</td>
</tr>
<tr>
<td>Any</td>
<td>PREFETCH</td>
<td>B aborts</td>
</tr>
</tbody>
</table>
Algorithm 2 LSA STM algorithm (encounter-time locking/write-back variant) [11]

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Global state:</td>
</tr>
<tr>
<td>2.</td>
<td>clock ← 0</td>
</tr>
<tr>
<td>3.</td>
<td>oreces: word-sized ownership records, each consisting of:</td>
</tr>
<tr>
<td>4.</td>
<td>locked: bit indicating if orece is locked</td>
</tr>
<tr>
<td>5.</td>
<td>owner: thread owning the orece (if locked)</td>
</tr>
<tr>
<td>6.</td>
<td>version: version number (if → locked)</td>
</tr>
<tr>
<td>7.</td>
<td>State of thread p:</td>
</tr>
<tr>
<td>8.</td>
<td>lb: lower bound of snapshot</td>
</tr>
<tr>
<td>9.</td>
<td>ub: upper bound of snapshot</td>
</tr>
<tr>
<td>10.</td>
<td>r-set: read set of tuples (addr, val, ver)</td>
</tr>
<tr>
<td>11.</td>
<td>w-set: write set of tuples (addr, val)</td>
</tr>
</tbody>
</table>

```
stm-start():
12.   lb ← ub ← clock
13.   r-set ← w-set ← Ø
15.   stm-load(addy):
16.      oreces.val ← (oreces hash(addy)), addy) \post-validated atomic read [11]
17.      if oreces.locked then
18.         if oreces.owner ≠ p then
19.             abort() \orece owned by other thread
20.             if (addy.new-val, s) ∈ w-set then
21.                 val ← new-val \update write set entry
22.             else
23.                 if oreces.version > ub then \try to extend snapshot
24.                     ub ← clock
25.                     if validate() then
26.                         abort() \cannot extend snapshot
27.                     val ← addy
28.                     r-set ← r-set ∪ \{addy, val, oreces.version\} \add to read set
30.     return val
31.     stm-store(addy, val):
32.      oreces ← oreces hash(addy)
33.      if oreces.locked then
34.        if oreces.owner ≠ p then
35.            abort() \orece owned by other thread
36.        else
37.            if (addy, ver) ∈ r-set ∧ ver ≠ oreces.version then
38.                abort() \read different version earlier
39.            if cas(oreces hash(addy), oreces → (true, p)) then
40.                abort() \cannot acquire orece
41.        w-set ← w-set \{addy, val\} \add to write set
42.     stm-commit():
43.      if w-set ≠ Ø then \is transaction read-only?
44.      if ub ≠ lb + 1 then \commit timestamp
45.      if validate() then
46.          abort() \cannot extend snapshot
47.          o-set ← Ø \set of oreces updated by transaction
48.      for all (addy, val) ∈ w-set do \write updates to memory
49.          addy ← val
50.      for all s ∈ o-set do \set of oreces updated by transaction
51.          oreces[s] ← (false, ub) \release oreces
52.     validate():
53.      if for all (addy, val, ver) ∈ r-set do \Are oreces free and version unchanged?
54.         oreces ← oreces hash(addy)
55.      if (oreces.locked ∧ oreces.owner ≠ p) ∨ \{oreces.locked ∧ oreces.version ≠ p\} then
56.         abort() \inconsistent snapshot
```

pace required to successfully run transactions in hardware. Likewise, visible reads result in significant overheads for STMs. This HyTM is also used in a study about the HTM support in Rock [9].

In phased TM [15], the implementation modes for transactions can be switched globally and at once (i.e., only software or hardware transactions are running at a time). The serial irrevocable mode that is present in most current STMs is a special case of the phased approach, as it can be used as a very simple software fallback for HTMs. The phased TM approach is orthogonal to hybrid TM.

The HyTM [12] presented by Hofmann et al. uses a simple global lock as software fallback mechanism instead of an STM that can run several software transactions concurrently. Hardware transactions wait for a software transaction to finish before committing, but are not protected from reading uncommitted and thus potentially inconsistent updates of software transactions. Note that with ASF, hardware transactions are not completely sandboxed. For example, page faults due to inconsistent snapshots will abort speculative regions but will also be visible to the operating system.

The hardware-accelerated STM algorithms (HASTM) [3] by Saha et al. are based on ownership records (like LSA but unlike NOrec). HASTM in cautious mode monitors application data and does read logging, whereas our hybrid LSA algorithms (see Section 3) monitor ownership records and do not log reads. HASTM in aggressive mode monitors both application data and ownership records, thus suffering from higher HTM capacity requirements (evaluated in Section 5). Thus, only our hybrid LSA algorithms can change the ownership-record-to-memory mapping to achieve a larger effective read capacity. Furthermore, HASTM in cautious mode as presented in the paper does not prevent dirty reads, which can crash transactions in unmanaged environments such as C/C++.

Spear et al. propose to use Alert-On-Update (AOU) [17] to accelerate snapshots by reducing the number of necessary software snapshot validations in STMs based on ownership records. How-

**ever,** our LSA STM algorithm already has efficient time-based snapshots due to its use of a global time base, whereas AOU uses a commit counter heuristic, which can suffer from false positives that lead to costly re-validations. The details of the AOU algorithm are not presented, thus it is difficult to assess the remaining HyTM aspects and overheads. Neither AOU nor HASTM use speculative writes for transactional stores, whereas our HyTMs do.

### 3. The Hybrid Lazy Snapshot Algorithms

Our first two algorithms extend the lazy snapshot algorithm (LSA) first presented in [22]. In this section, we first briefly recall the principles of LSA. We then describe its hybrid extensions using eager and lazy conflict detection.

#### 3.1 The LSA Algorithm

LSA is a time-based STM algorithm that uses on-demand validation and a global time base to build a consistent snapshot of the values accessed by a transaction. We base our description on the blocking word-based version of LSA [11] in its basic encounter-time locking, write-back variant. The algorithm is shown in Algorithm 2.

Transaction stores are buffered until commit. The consistency of the snapshot read by the transaction is checked based on versioned locks (ownership records, or oreces for short) and a global time base, which is typically implemented using a shared counter. The orecs protecting a given memory location is determined by hashing the address and looking up the associated entry in a global array of oreces. Note that, in this design, an orece protects multiple memory locations.

To install its updates during commit, a transaction first acquires the locks that cover updated memory locations (line 38) and obtains

1. It first checks the version in an ownership record and then loads data speculatively. Executing these steps in reverse order fixes this problem.
a new commit time from the global time base by incrementing it atomically (line 43). The transaction subsequently validates that the values it has read have not changed (lines 45 and 53–57) and, if so, writes back its updates to shared memory (lines 48–49). Finally, when releasing the locks, the versions of the orecs are set to the commit time (lines 51–52). Reading transactions can thus see the virtual commit time of the updated memory locations and use it to check the consistency of their read set. If all loads did not virtually happen at the same time, the snapshot is inconsistent.

A snapshot can be extended by validating that values previously read are valid at extension time, which is guaranteed if the versions in the associated orecs have not changed. LSA tries to extend the snapshot when reading a value protected by an orec with a version number more recent than the snapshot’s upper bound (line 25), as well as when committing to extend the snapshot up to the commit time, which represents the linearization point of the transaction (line 45).

3.2 The Hybrid LSA Algorithms

Let us now describe the hybrid extensions of LSA, starting with HyLSA-eager (shown in Algorithm 3), which is based on eager conflict detection. Note that the HyTM decides at runtime whether to execute in hardware or software mode, as explained in Section 2 and Algorithm 1.

Transactional loads first perform an ASF-protected load of the associated orec (line 6). This operation monitors the orec for changes and will lead to an abort if the orec is updated by another thread. If the orec is not locked, the transaction uses a nonspeculative load operation (line 9) to read the target value. Note that ASF will start monitoring the orec before loading from the target address (see Section 2.1). If the transaction is not aborted before returning a value, this means that the orecs associated with this address and all previously read addresses have not changed and are not locked, thus creating an atomic snapshot.

Transactional stores proceed as loads, first monitoring the orec and verifying that it is not locked (lines 12–14). The transaction then watches the orec for reads and writes by other transactions (_PREFETCHW on line 15). The operation effectively ensures _eager_ detection of conflicts with concurrent transactions. Finally, the updated memory location is speculatively written (line 16).

Upon commit, an update transaction first acquires a unique commit timestamp from the global time base (line 20), will become visible before the transaction’s commit), speculatively writes all updated orecs (lines 21–22), and finally tries to commit the transaction (line 23). If it commits successfully, then we know that no other transaction performed conflicting accesses to the orecs (representing data conflicts). Thus, the hardware transaction could have equally been a software transaction that acquired read and write locks for its orecs. If the hardware transaction aborts, then it only might have incremented _clock_, which is harmless.

Our second hybrid extension of LSA, HyLSA-lazy, is shown in Algorithm 4. It uses lazy conflict detection: upon store, we do not read nor watch the orec associated with the accessed memory location, but instead we speculatively write to the target location (line 12).

For an LSA update transaction to commit correctly, its commit timestamp must be strictly larger than the value of _clock_ at the time when the transaction had acquired—or, for a hardware transaction, started monitoring—all of the orecs associated with updated locations. Therefore, we start the commit phase of update transactions by speculatively writing to all orecs (lines 17–21).

We use several optimizations to speed up commit in good cases. Similarly to the commit-phase optimizations by Zhang et al. [20], we first install an optimistic commit timestamp (line 17) and next load _clock_ again (line 22) to check the invariant. Because the orec “acquisition” with speculative stores is not bound to a value like for a typical software lock, we can optimize more aggressively than in an STM by updating orecs to the assumed final value (orec.locked is false, line 1 ), which can then potentially allow us to skip the second orec update loop (lines 25–26). Nevertheless, the bigger effect on performance might be that we can potentially skip incrementing _clock_ and share the same commit timestamp between nonconflicting transactions.

---

3 We assume here that ASF starts monitoring the orecs before loading _clock_ (otherwise, we would have to insert a memory barrier before the load).
We use a typical STM quiescence-based protocol to ensure privatization safety for the hybrid LSA algorithms (not shown in the pseudo-code). Basically, update transactions potentially privatize data, so they have to wait until concurrent transactions that might have accessed the updated locations have finished or have extended their snapshot far enough into the future (so that they would have observed the updates). Because hardware transactions will be aborted immediately by conflicting updates, their snapshot is always most recent and we do not need to wait for them.

4. The Hybrid NOrec Algorithms

In this section, we take another algorithm from the literature, NOrec [7], and discuss how to turn it into a scalable HyTM for ASF. Roughly speaking, NOrec uses a single orec (a global versioned lock) and relies on value-based validation in addition to time-based validation. The reason for creating a hybrid extension to NOrec is that this algorithm can potentially provide better performance for low thread counts because it does not have to pay the runtime overheads associated with accessing multiple orecs. On the other hand, LSA is expected to provide better scalability with large thread counts or frequent but disjoint commits of software transactions. Therefore, both algorithms are of practical interest depending on the target architecture and workload.

4.1 The NOrec Algorithm

The NOrec algorithm, shown in Algorithm 5, is quite similar, when ignoring VBV, to timestamp-based TMs like TL2 [8] or LSA [22]. The main difference with the description given in Section 3 is that NOrec uses a single orec (gsl, line 2) and does not acquire the lock before attempting to commit a transaction. As a consequence, it yields a very simple implementation and allows for a few optimizations. In particular, it is not necessary to track which locks are covering loads or stores, and the lock itself can serve as time base (lines 28/32, 11, and 36). However, such a design would not scale well when update transactions commit frequently because timestamp-based validation would also fail frequently (e.g., in the checks on lines 19 and 28). Therefore, NOrec attempts value-based validation (VBV, lines 38–40) whenever timestamp-based validation is not successful (lines 20 and 29).

With VBV, the consistency of a transaction’s read set is verified on the basis of the values that have been loaded instead of the versions of the orecs. The disadvantage of using values is that one has to potentially track more data in the read set because several addresses often map to the same orec. VBV is typically paired with serialized commit phases. In NOrec, this is enforced on lines 12 and 37.

Our implementation of NOrec differs in a few points from the original implementation [7]. Notably, in our implementation, when writing back buffered updates upon commit, we only write to precisely those bytes that were modified by the application, whereas the original implementation always updates the granularity of aligned machine words. This more complex bookkeeping introduces higher runtime overheads but is required for the STM to operate correctly according to the C/C++ TM specification [13].

The NOrec HyTM algorithm shown in Algorithm 6 is based on the informal description by Dalessandro et al. [7], adapted to use ASF as HTM. In the algorithm, the purpose of the additional lock esl is for software transactions to be able to interrupt and stop hardware transactions. The latter monitor this lock (line 25), so that any store to this location will abort them. Also, they will not proceed if the lock is acquired (line 27). Because software transactions only update data when they have acquired esl, hardware transactions never see inconsistent state (e.g., partial software commits). Note that esl is only modified by software transactions because hardware transactions would otherwise abort each other, which is not necessary because they access all data speculatively (lines 20 and 23) and thus ASF will resolve any conflict. To avoid false sharing, gsl and esl are located in separate cache lines in our implementation.

In turn, hardware transactions increase the version number of the global sequence lock gsl (line 30), which will trigger validation in active software transactions. From the perspective of software transactions, committed hardware transactions are thus equivalent to software transactions that committed atomically.

4.2 A Scalable Hybrid NOrec Algorithm

The major problem of Algorithm 6 is that it does not scale well in practice (see Section 5). In what follows, we will construct an algorithm, HyNOrec-2 that performs much better while being no more complex. To better explain and evaluate the different optimizations involved, we additionally show two intermediate algorithms.

Dalessandro et al. assume [7] that the update of the contented gsl by every hardware transaction (line 30 in Algorithm 6) is not a performance problem because it would happen close to the end of
a transaction. However, we observed in experimental evaluation a high rate of aborts and poor overall performance for this algorithm.

Algorithm 7 shows our first (intermediate) NOrec-based HTM, which will serve as the basis for the other two variants. As a first straightforward optimization, a hardware transaction has to update gsl only if it will actually update shared state on commit (line 26).

Second, we do not need to use a small hardware transaction to update both gsl and esl in *stm-commit*. This is not necessary because esl is purely used to notify hardware transactions about software commits and can only be modified by a software transaction that previously acquired gsl (line 7). In contrast to Algorithm 6, this allows hardware transactions to try to commit at a time where gsl has been acquired but esl has not yet been updated (which would have aborted the hardware transaction). However, this case can be handled by just letting the hardware transaction abort if gsl has been locked (line 29).

This second change is not about performance but it allows us to have a software fallback path in the HyTM that does not depend on HTM progress guarantees (e.g., no spurious aborts), which are surprisingly difficult to implement [10]. Also, programs can use the software path in the HyTM as is on hardware that does not support ASF.

Algorithm 7 can still suffer from conflicts on gsl if updating hardware transactions commit frequently. Algorithm 8 shows that we can replace the speculative update of gsl with a nonspeculative atomic fetch-and-increment instruction (line 9), which allows hardware transactions that access disjoint data to not abort each other anymore and makes the algorithm scale better.

To understand why this is possible, consider possible orderings of the hardware transaction’s fetch-and-increment and a software transaction’s compare-and-set (CAS) on gsl. If the increment comes first, the CAS will fail and will cause a software transaction validation. If the software transaction accesses during validation any updates of the hardware transaction before the former can commit, it will abort the hardware transaction, making the situation look like if some transaction committed without updating anything. If in contrast the CAS comes first, the hardware transaction will notice that gsl was locked before it incremented gsl and will abort. The hardware transaction’s update to gsl is harmless because no transaction interprets *gsl clock* if gsl is locked.

the fetch-and-increment yields lower performance according to our experiments.

---

**Algorithm 6 HyNOrec-DSS: HyTM by Dalessandro et al. [7] (extends Algorithm 5)**

1. Global state: ▶ extends state of Algorithm 5
2. esl: extra sequence lock
3. *stm-acquire-locks()*: ▶ start hardware transaction (retry code omitted)
4. SPECULATE: ▶ try to acquire commit lock
5. if *l* = *gsl* then
6. LOCK MOV: *l* ← *gsl* ▶ also acquire extra lock
7. COMMIT ▶ true ▶ locks were acquired atomically
8. return

**Algorithm 7 HyNOrec-0: STM acquires locks separately (extends Algorithm 5)**

1. Global state: ▶ extends state of Algorithm 5
2. esl: extra sequence lock
3. State of thread *p*: ▶ extends state of Algorithm 5
4. update: are we in an update transaction?
5. *stm-commit()*: ▶ replaces function of Algorithm 5
6. if *w-set* ≠ ∅ then ▶ is transaction read-only?
7. while ¬cas(*gsl*, sl → (true, sl.clock)) do ▶ acquire commit lock
8. *sl* ← validate() ▶ also acquire extra lock
9. *sl* ← (true, sl.clock) ▶ also acquire extra lock (no need for cas)
10. for all *(addr, val)* ∈ w-set do ▶ write updates to memory
11. +addr ← val ▶ protected load
12. *esl* ← (false, sl.clock + 1) ▶ release locks and increment clock
13. *gsl* ← (false, sl.clock + 1) ▶ release locks and increment clock
14. if update then ▶ speculative write
15. *esl* ← addr ▶ true ▶ are we in an update transaction
16. return
17. *stm-store(addr, val)*: ▶ speculative write
18. *lock mov*: *l* ← *addr* ▶ protected load
19. return

**Algorithm 8 HyNOrec-1: HTM writes gsl nonspeculatively (extends Algorithm 7)**

1. *stm-start()*: ▶ replaces function of Algorithm 7
2. *lock mov*: *l* ← *esl* ▶ protected load (monitor extra lock)
3. *stm-commit()*: ▶ extra lock available?
4. *lock mov*: *l* ← *gsl* ▶ protected load (monitor extra lock)
5. *lock mov*: *l* ← *esl* ▶ protected load (monitor extra lock)
6. *stm-commit()*: ▶ initially not an update transaction
7. *stm-commit()*: ▶ extra lock available?
8. *stm-commit()*: ▶ protected load
9. *stm-commit()*: ▶ main lock unavailable, we will be aborted anyway
10. *stm-commit()*: ▶ main lock unavailable
11. *stm-commit()*: ▶ main lock unavailable
12. COMMIT ▶ commit hardware transaction
13. COMMIT ▶ commit hardware transaction

---

3As a matter of fact, esl.clock can contain any value as long as the clock bit is updated properly because such an update will abort hardware transactions monitoring esl.

Note that the fetch-and-increment will be ordered before the commit of the transaction. Also, using a typical compare-and-set loop instead of

...
Additionally, hardware transactions spin nonspeculatively if esl is locked before accessing it speculatively to avoid unnecessary aborts (line 2).

The remaining problem of Algorithm 6 is that committing a software transaction aborts all hardware transactions that execute concurrently. One might see this as a minor issue assuming that, typically, software transactions are much longer than hardware transactions, but this is not necessarily the case. There are several reasons why a transaction cannot use ASF, for example because it contains instructions that are not allowed in ASF speculative regions (e.g., rdtscl), or because its access pattern quickly exceeds the associativity of the cache used to track the speculative loads, hence leading to capacity aborts after only few accesses.

Fortunately, software transactions can commit without having to abort nonconflicting hardware transactions. The key insight to understand this second extension is that the monitoring in hardware transactions is like an over-cautious form of continuous value-based validation (any conflicting access to a speculatively accessed cache line will abort a transaction). In NOrec, software transactions tolerate concurrent commits of other transactions by performing value-based validation when necessary.

Our final optimization is shown in Algorithm 9. Hardware transactions do not monitor esl using speculative accesses anymore. The purpose of esl is to prevent hardware transactions from reading inconsistent state such as partial updates by software transactions. To detect such cases and thus still obtain a consistent snapshot, hardware transactions first read the data speculatively (line 4) and then wait until they observe with nonspeculative loads that esl is not locked (line 5). If this succeeds and the transaction reaches line 6 without being aborted, it is guaranteed that it had a consistent snapshot valid at line 5 at a time when there were no concurrent commits by software transactions.

The reasoning for waiting until gsl is not locked on line 10 is similar. Waiting for gsl is as good as waiting for esl because esl will be locked iff gsl is locked (see Algorithm 7).

Note that esl could be removed and replaced by just gsl. A downside of this approach is that it would increase the number of cache misses on line 5 because both hardware and software commits would update the same lock. Therefore, we keep the separation between gsl and esl.

5. Evaluation

To evaluate the performance of our HyTMs, we use a similar experimental setup as in a previous study [5]. We simulate a machine with sixteen x86 CPU cores on a single socket, each having a clock speed of 2.2 GHz. The simulator is near-cycle-accurate (e.g., it simulates details of out-of-order execution). We evaluate three ASF implementations. Two of them, “LLB8” and “LLB256” can track/buffer speculative loads and store in a fully-associative buffer that holds up to 8 or 256 distinct cache lines. “LLB8L1” is a variant which uses only buffers for speculative stores but uses the L1 cache to track transactional loads. We show these ASF implementations because they have different costs when implemented in a microprocessor (e.g., required chip area). LLB8 will have to resort to the STM code path often because most transactions will exceed its capacity. LLB256 is sufficient to run almost all transactions in our benchmarks in the STM code path, but is more expensive. LLB8L1 represents a middle ground. Its capacity for loads is limited by either the cache’s size (1024 lines) or its associativity (2-way).

In order to get the best performance, the compiler links in the TM library statically and optimizes the code by inlining TM functions. The STM implementations that we compare against are a reimplementations of TinySTM [18] (write-through mode, eager conflict detection, ensuring privatization-safety, similar to Algorithm 2, denoted “LSA”) and NOrec [7] (similar to Algorithm 5). The baseline HTM (“HTM”) uses serial-irrevocable mode as simple software fallback. The HyTM implementations have the same names as the respective algorithms (e.g., Algorithm 9 is denoted “HyNOrec-2”) and use the LSA and NOrec implementations for their software code paths.

As benchmarks, we use selected applications from the STAMP TM benchmark suite [4] and the typical integer set microbenchmarks (IntegerSet). The latter are implementations of a sorted set of integers based on a skip list, a red-black tree, a hash table, and a linked list. During runtime, several threads use transactions to repeatedly execute insert, remove, or contains operations on the set (operations and elements are chosen randomly). All set elements are within a certain key range, and the set is initially half full. Table 2 shows the configurations that we consider. In HashTable all transactions are update transactions (insert or remove operations), in all other benchmarks the update rate is 20%. However, these operations only insert (remove) an element if it absent from (part of) the set, so the actual percentage of update transactions can be smaller. We use the Hoard memory allocator [2] in HashTable and glibc 2.10 standard malloc in the other benchmarks.

Because we do not have enough space to show all measurements in the same level of detail, we first present a few interest-

Table 2. IntegerSet microbenchmarks and approximate ratio of HTM commits to total number of commits.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Range</th>
<th>Percentage of commits on hardware code path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LLB8</td>
<td>LLB8L1</td>
</tr>
<tr>
<td>SkipList-Large</td>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>SkipList-Small</td>
<td>1024</td>
<td>&lt; 1% HyLSA: 90–95% HyNOrec: 95–100%</td>
</tr>
<tr>
<td>RB-Tree-Large</td>
<td>8192</td>
<td>0–2% HyLSA: 70–90% HyNOrec: 95–100%</td>
</tr>
<tr>
<td>RB-Tree-Small</td>
<td>1024</td>
<td>2–10% HyLSA: 85–95% HyNOrec: 100%</td>
</tr>
<tr>
<td>HashTable</td>
<td>128000</td>
<td>100% (except HyLSA** on LLB8-L1: 95%)</td>
</tr>
<tr>
<td>LinkedList-Large</td>
<td>512</td>
<td>3%</td>
</tr>
<tr>
<td>LinkedList-Small</td>
<td>25</td>
<td>30–60%</td>
</tr>
</tbody>
</table>

Figure 1. Ratio of HTM commits to total number of commits.
HyLSA-lazy

HyLSA-Eager-SDL

HyLSA-eager

HyNOrec-DSS

HyNOrec-DSSU

HyNOrec-0

HyNOrec-1

HyNOrec-2

Figure 2. Comparison of HyNOrec algorithms. The hardware contention abort rate is the number of aborts due to ASF contention per transaction that commits in hardware or switches to the software codepath.

Figure 3. Comparison of HyNOrec algorithms (HashTable).

Figure 4. Comparison of HyLSA algorithm variants.

Figure 5. Ratio of HTM commits to total number of commits for 8 threads and read-only LinkedList of various sizes (range), when accessing data speculatively (SDL) or not.

ing cases. Table 2 shows which percentage of transaction commits happen on the hardware code path in comparison to the total number of commits. LLB256 provides sufficient capacity to execute all transactions in our IntegerSet configurations in hardware. In contrast, LLB8’s capacity is most often too small. Note that in our implementations, only permanent ASF abort reasons like exceeding ASF’s capacity make the HyTM switch to the software code path. Contention will not result in such a switch unless a transaction suffers from a high number of retries (100 in our experiments). Therefore, the ratio of HTM’s commits that we show is essentially independent of the level of contention in a workload.

Figure 2 shows a comparison between the HyNOrec algorithms for the same SkipList benchmarks but with two different ASF implementations. With LLB8 (left side), all transactions have to fall back to software executions (see Table 2), but interestingly HyNOrec-2 is able to scale better than the other algorithms. The abort rate due to contention shows that this is because hardware transactions in the other HyNOrec variants suffer from contention aborts before they notice a capacity abort, which makes them switch to software execution. Because HyNOrec-2 does not monitor estl, it will not be aborted by commits of nonconflicting software transactions, and will find out quickly that it should switch to software, then taking advantage of STM scalability.

When using LLB8L1 (right side), many transactions can execute in hardware (see Table 2 and Figure 1). HyNOrec-2 also scales much better in this case, showing that its ability to survive commits of nonconflicting software transactions (e.g., in contrast to HyNOrec-1) is beneficial even when the majority of transactions execute in hardware. Furthermore, HyNOrec-DSS suffers from many more aborts than the other TMs. To explain this, we show results for HyNOrec-DSSU, which is like HyNOrec-DSS but only updates gsl when update transactions commit, thus reducing the number of speculative updates to gsl (SkipList has 20% update transactions). HyNOrec-DSSU performs similarly to HyNOrec-0, indicating that this part of the HyNOrec-0 optimizations is crucial. HyNOrec-DSS never performed better than HyNOrec-0 in any of our benchmarks and often performed significantly worse. It does not scale beyond 4 to 6 threads in IntegerSet unless transactions execute the software code path most of the time. Therefore, we discard HyNOrec-DSS from now on.
is Algorithm 4 without such barriers, which shows their runtime overhead. However, scalability remains similar in our benchmarks. HyLSA-lazy can scale slightly better than HyLSA-eager, which is likely due to lazy conflict detection. Because both perform similar in many situations, we will focus on HyLSA-eager.

HyLSA’s capacity requirements are different than those of HyNOrec. HyLSA buffers updates speculatively and thus, for stores, needs ASF capacity for both data and orecs. However, for loads, only orecs are accessed speculatively, and the hash function that maps data to orecs influences capacity requirements. In our implementations, word-sized data are mapped to word-sized orecs (i.e., we discard the lower three bits of an address and select with the remaining bits a slot in an array with $2^{20}$ orecs). Orecs are not cache-line padded because padding would likely increase capacity requirements for HyLSA unless more than one adjacent cache line maps to the same orec. Without padding, hardware transactions detect conflicts on cacheline granularity, whereas STM transactions can detect conflicts on word-size granularity and can thus potentially scale better in high-contention workloads.

Table 2 and Figure 1 show that HyLSA is already more likely to hit capacity limitations than HyNOrec just because it needs twice the capacity for stores, so it is important for HyLSA to read data nonspeculatively. Figure 5 illustrates this point further, showing that when accessing data nonspeculatively (HyLSA-eager-SDL), less transactions can execute the hardware code path.

Figure 6 presents a concluding overview of TM performance with the integer set microbenchmarks. We show configurations that are representative or that highlight interesting properties. HashTable performs similar on all ASF implementations and scales very well, but ultimately suffers from external bottlenecks (e.g., the memory allocator). LLB8 is (on all other benchmarks) not sufficient to run many transactions in hardware, and STMs perform slightly better than HyTMs because the latter try to first execute in hardware (unsuccessfully).

HyNOrec-2 has very good overall performance, especially on scalable workloads. It is not aborted by nonconflicting concurrent commits of software transactions, which is one of the reasons why it performs better than HyNOrec-0 (e.g., in SkipList-Large on LLB8L1, see Figure 1 for HTM ratio). Pure HTM has the lowest overhead but its simple fallback mode (serial execution) can quickly decrease its performance. HyLSA has higher runtime overhead than HyNOrec but typically scales well.

In the small LinkedList on LLB256, all transactions can execute in hardware but HyTMs and HTM do not scale. The reason for this behavior is that ASF’s conflict detection is on the granularity of cache lines, whereas STMs can use smaller granularities (word-sized in LSA, value-based validation in NOrec), which can be beneficial in high-contention workloads with a high level of false sharing. As explained before, HyLSA could use the indirection of the orecs and the memory-to-orec hash function to emulate a smaller granularity for conflict detection. However, this will waste ASF capacity and thus does not seem to be a generally useful strategy. Instead, a HyTM should perhaps switch proactively to software to try to employ a more contention-resistant STM algorithm.

To conclude the evaluation, we show performance results for selected applications from STAMP (see Table 3) in Figure 7. We chose benchmarks that are stable and have parallelism in their workloads, and executed them using STAMP’s standard parameter configurations for simulator environments. LLB256 is again sufficient to execute all transactions in hardware. SSCA2 and KMeans have small transactions, but interestingly HyNOrec-0 seems to require just a little too much capacity (in contrast to the other HyNOrec TMs, it accesses esl and gsl speculatively). Genome on LLB8 also exhibits this behavior. HyLSA’s larger capacity requirements for stores decrease the HTM ratio as well.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LLB8</th>
<th>LLB8-L1</th>
<th>LLB256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>HTM: HyNOrec-2: 65%</td>
<td>HyNOrec: 85-90%</td>
<td>HyLSA: 75%</td>
</tr>
<tr>
<td></td>
<td>HyNOrec-1: 60%</td>
<td>HyNOrec-2: 50%</td>
<td>HyLSA: 38-42%</td>
</tr>
<tr>
<td></td>
<td>HyNOrec: 9-12%</td>
<td>HyNOrec: 9-12%</td>
<td>HyLSA: 3-5%</td>
</tr>
<tr>
<td></td>
<td>HyNOrec: 6-9%</td>
<td>HyLSA: 1-2%</td>
<td></td>
</tr>
<tr>
<td>KMeans-Hi</td>
<td>HTM: 9-12%</td>
<td>HyLSA: 9-12%</td>
<td>HyNOrec: 9-12%</td>
</tr>
<tr>
<td>KMeans-Lo</td>
<td>HTM: 9-12%</td>
<td>HyLSA: 9-12%</td>
<td>HyNOrec: 9-12%</td>
</tr>
<tr>
<td>Vacation-Hi</td>
<td>HTM: 13-14%</td>
<td>HyNOrec: 9-12%</td>
<td>HyLSA: 3-5%</td>
</tr>
<tr>
<td>Vacation-Lo</td>
<td>HTM: 8-11%</td>
<td>HyLSA: 6-9%</td>
<td>HyNOrec: 85-90%</td>
</tr>
<tr>
<td>SSCA2</td>
<td>99-100%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Approximate ratio of HTM commits to total number of commits in STAMP.
HyNOrec-2 performs best among the HyTMs most of the time and is often close to HTM. Its performance suffers in Genome due to its software fallback (NOrec) performing worse than LSA. It often performs much better than HyNOrec-0, thus demonstrating the benefits of our optimizations.

HyLSA has higher runtime overhead than HyNOrec but scales well. Note that we only show HyLSA-eager but HyLSA-lazy performs typically similar to HyLSA-eager according to our observations (both implementations currently have to use memory barriers, as explained previously).

6. Conclusion

In this paper, we have proposed and evaluated novel hybrid software/hardware transactional memory algorithms. They are extensions to two state-of-the-art STM algorithms (LSA and NOrec) and rely on AMD’s ASF hardware extensions for the execution of transactions in hardware. We confirmed through experimental evaluation of the algorithms on a near-cycle-accurate x86 simulator that they provide good scalability, a low runtime overhead, can tolerate concurrent commits of nonconflicting software transactions, and perform significantly better than previous HyTM proposals.

Based on our experience, the key features of ASF that enable implementation of efficient HyTM algorithms are that (1) it allows nonspeculative operations to be used inside of transactions, including atomic operations such as compare-and-set, and (2) that it monitors speculatively accessed locations eagerly for conflicting accesses by other threads. We believe that the techniques that we used in our algorithms apply not just to HyTM but can be useful in general for concurrent algorithms based on new synchronization hardware like ASF.

Acknowledgements We would like to thank Stephan Diestelhorst of AMD for his clarifications regarding the ASF specification. The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement Nº 216852.

References


