An Opaque Hybrid Transactional Memory

Wenjia Ruan and Michael Spear
Lehigh University
{wer210, spear}@cse.lehigh.edu

Abstract

The arrival of best-effort hardware transactional memory (TM) creates a challenge for designers of transactional memory runtime libraries. On the one hand, using hardware TM can dramatically reduce the latency of transactions. On the other, it is critical to create a fall-back path to handle the cases where hardware TM cannot complete a transaction, and this path ought to be scalable and reasonably fair to all transactions. Additionally, while the hardware-accelerated system is likely to have weaker safety guarantees than a pure hardware TM, it ought not to be weaker than what software TM guarantees.

We propose a new hybrid TM algorithm based on the "Cohorts" software TM algorithm. Our algorithm guarantees opacity by preventing any transaction from observing the un-committed state of any other transaction. It does so via a novel state machine that maximizes the use of hardware TM, while affording opportunity to enforce fairness policies. We present an implementation of our Hybrid Cohorts that prioritizes transactions that fall back to software mode. In this manner, we ensure that long-running transactions do not starve, while still allowing concurrency among hardware and software transactions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Transactional Memory, Synchronization, TSX, Opacity, GCC

1. Introduction

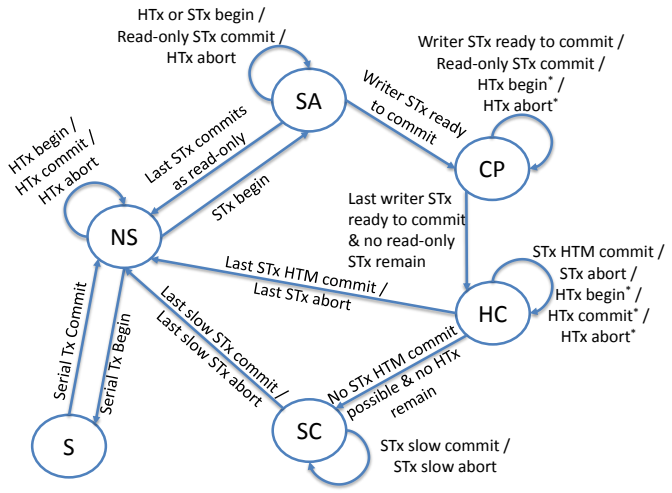
For over 20 years, Transactional Memory (TM) [12] has been viewed as the most promising proposal for simplifying the creation of correct, scalable concurrent programs. The concept behind TM is tantalizingly simple: programmers merely annotate regions of code that must appear to execute atomically, and then a run-time system, augmented with custom hardware, executes those regions concurrently (as "transactions"). During execution, the run-time system tracks memory accesses, detects conflicts, and aborts and retries transactions as necessary to ensure that the program behavior is equivalent to one in which the execution of transactions does not overlap.

The recent addition of TM support to IBM [14, 34] and Intel [13] processors brings the field of concurrent programming much closer to a state in which programmers can eschew locks in favor of transactions. However, first-generation hardware TM systems carry a number of limitations. Most significantly, these implementations are "best effort" [17], in that they do not guarantee that any transaction attempt will commit. In particular, a transaction attempt may fail if it accesses more unique locations than the hardware can support, or if there is an interrupt (e.g., a timer interrupt) during its execution. Consequently, a TM runtime that wishes to use hardware TM must provide a software fall-back path. This fall-back path also provides a means of circumventing the hard-coded conflict resolution strategy ("requester wins" [1]) that the hardware enforces, so as to allow the run-time system to improve the chance that a long-running transaction does not starve.

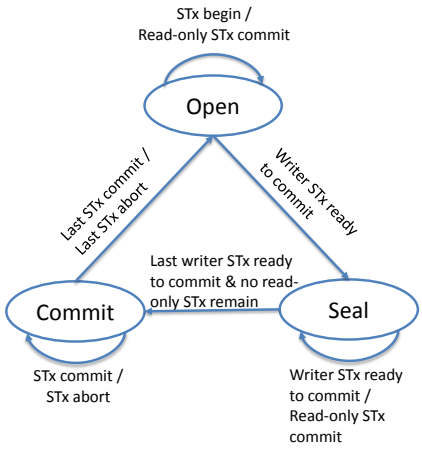
Broadly speaking, TM runtime systems that combine the use of hardware TM with a software fall-back path are called hybrid TM [24]. Existing hybrid TM proposals can be categorized as follows:

- Low-Scalability Fall-back: Lev's PhaseTM [18] was among the earliest hybrid TMs. While it envisioned a variety of different ways to compose hardware and software transactions, it required that all transactions used the same technique at the same time (i.e., all use hardware, or all use a software TM algorithm). Of the many approaches, the combination of hardware TM with a single-lock fall-back was perhaps the most straightforward [4], and has subsequently been improved, e.g., by Calciu et al. [3].
- Scalability Through Non-Transactional Actions: The systems by Dalessandro et al. [5] and Riegel et al. [26] both assumed that the underlying hardware TM would allow non-transactional operations within a transactional context (for reading and writing, respectively). Without this support, these systems degrade roughly to PhaseTM when executed on existing TM hardware.
- Hybrid TM-Specific Hardware: proposals by Minh et al. [23], Shriraman et al. [32], and Saha et al. [30] assumed that the hardware TM would provide a wide API so that a hybrid runtime system could use parts of the hardware (e.g., tracking cache invalidations of specific lines) to accelerate software TM. The hardware for these systems is not currently available, nor does it appear in any product roadmaps.
- Reduced Hardware Capacity: Systems by Kumar et al. [16], Damron et al. [7], and Riegel et al. [26] required all hardware transactions to access the per-location metadata used by the software TM fall-back. This approach can improve the concurrency between hardware and software transactions, but it effectively halves the capacity of the hardware, and is largely viewed as impractical.
- Unsafe Hybrid TM: The Invyswell system [2] reduces the safety of hybrid transactions by sacrificing opacity [11]. The resulting system cannot guarantee correctness in the face of certain patterns [8], but can scale well on existing systems.
- Behavior-Specific Hybrid TM: Reduced Hardware NOrec [20] ensures opacity and is compatible with existing hardware TM. However, its performance relies upon transactions following a specific pattern, in which there is a large read-only prefix before the transaction's first attempted write. While appropriate for data structures, this may not be a suitable approach for realistic applications.

From an architectural perspective, we believe it unlikely that vendors will extend future micro-architectures with hybrid TM features or add non-transactional actions. However, it is likely that future hardware TM may overcome its existing capacity constraints (e.g., by expanding the capacity/associativity of private caches, or



(a) State transitions of the Hybrid Cohorts algorithm. STx refers to a software-mode transaction, and HTx refers to a hardware-mode transaction. The lack of a label on an arc indicates that a transaction behavior is either impossible or not allowed. For example, an HTx is not allowed to commit in the SA or CP states, and it is not possible for an STx to abort in these states.



(b) State transitions of the Cohorts algorithm. The Open, Seal, and Commit states correspond to the SA, CP, and SC states of Hybrid Cohorts.

Figure 1: State transitions for the Hybrid Cohorts (left) and Cohorts (right) algorithms.

by moving conflict tracking structures higher in the cache hierarchy). Thus we believe that the most important qualities of a hybrid TM are to provide a safe programming model, to minimize the use of hardware capacity for tracking metadata, and to emphasize fairness and progress for transactions that fall back to software.

To provide these properties, we introduce the Hybrid Cohorts (HyCo) algorithm. Based on the Cohorts algorithm [28], HyCo uses a state machine to manage the behavior of transactions. By guaranteeing the immutability of memory during any software transaction's execution, and employing hardware TM as broadly as possible, HyCo minimizes instrumentation for all transactions, and eliminates many of the bottlenecks of the original Cohorts algorithm, without sacrificing safety.

The remainder of this paper is organized as follows. In Section 2, we discuss the overall approach of the Hybrid Cohorts algorithm, with a focus on the state machine that governs transaction behavior. Section 3 presents the pseudocode for one implementation of the state machine, which aims to limit the impact on transactions that use hardware TM resources throughout their execution. In Section 4, we present the results of performance experiments. Section 5 concludes and discusses some future research directions.

2. The Hybrid Cohorts Algorithm

The foundation of the HyCo algorithm is a state machine that governs when transactions may begin, as well as when and how they commit. This state machine appears in Figure 1a. For reference, the original Cohorts state machine is provided in Figure 1b.

In the original Cohorts algorithm, the role of the state machine was to ensure that memory remained constant whenever a transaction was in-flight (i.e., between its begin and end points). This entailed blocking writing transactions from committing whenever a transaction was in-flight, and blocking transactions from beginning whenever a transaction was committing. The Cohorts algorithm also assumed that a transaction requiring irrevocability [33, 35] could do so by starting directly in the commit state.

In our new algorithm, we begin by formalizing irrevocability through the addition of a "serial" state (S). We then split the entry state (Cohorts::Open): instead of indicating that software transactions may be active, the split state distinguishes between when at least one software transaction is active (SA), and when no software transactions are running (NS). The commit pending state (CP) is equivalent to the Cohorts::Seal state. Finally, the Cohorts::Commit state is split, so that one-at-a-time slow commit (SC) can be avoided via a HTM-assisted commit phase (HC).

The original Cohorts algorithm also exposed options for how to detect conflicts, to include the use of ownership records [9, 29] or values [6, 25]. In HyCo, we exclusively use value-based conflict detection. To use other metadata would necessitate the use of HTM resources for concurrency control, which would, in turn, reduce the size above which transactions must run in software mode.

The algorithm affords a number of implementation choices and options. For example, the labels marked with an asterisk(*) correspond to a variant in which more hardware-mode transactions (HTx) are allowed. Similarly, there are a variety of ways to choose the order in which transactions perform their slow commit, depending on contention management [31] policies. For this discussion, we assume that the contention manager randomly chooses the order in which transactions attempt to commit.

2.1 Transitions

The initial state of the system is NS, indicating that no software or serial transactions are running. Should a transaction require serial-mode execution, it does so by transitioning from the NS state to the S state. This transition may entail either (a) forcibly aborting any in-flight hardware transactions, or (b) setting a flag to prevent subsequent HTx and STx transactions from beginning, and then waiting for the system to be in the NS state with no HTx transactions running. Implementation details for achieving this transition appear in Section 3.

When a transaction is in serial mode, it is not allowed to abort, and no other transactions may execute. When the transaction commits, the system transitions back to the NS state.

In the NS state, there are no STx transactions running. Thus as long as the system remains in NS state, HTx transactions may

execute in their entirety, either committing or aborting and retrying. However, as soon as an STx begins, the system transitions from NS to SA. In SA, new hardware and software transactions may begin. However, hardware transactions may not commit: they must abort or wait if they reach their commit point while the system is in the SA state. STx transactions accumulate their reads and writes in thread-private logs, with all writes buffered until commit time. As in the Cohorts algorithm, a read-only STx (detected by its empty write log) can commit directly from the SA state, since it does not modify memory. This may transition the system back to NS, if it results in all remaining transactions being STx.

From the SA state, as soon as the first writing STx is ready to commit, the system transitions to the Commit Pending (CP) state. From this state, additional read-only STx may commit, writing STx may announce that they are ready to commit, and HTx may begin or abort. Note that none of these transaction behaviors can affect the in-flight STx, since these behaviors do not affect shared memory.

When the last STx reaches the CP state, HyCo transitions to the HTM-assisted Commit state (HC). Any in-flight HTx transactions are permitted to commit immediately; all STx transactions use a hardware transaction to first validate their read set, and if it has not changed, to replay all writes from the thread-private log. Note that when an HTx transaction aborts, it can retry immediately, as can a hardware transaction attempting to commit the STx. However, if the STx validation fails, then the STx does not retry until the system returns to the NS state.

If all STx can commit or abort from the HC state, then the system transitions back to NS, even if HTx transactions are still executing. However, if any STx cannot commit via HTM (e.g., due to its read and write sets being too large to traverse and replay in a hardware transaction), then once there are no further STx attempting to commit in HTM, and no remaining HTx, the system transitions from HC to SC, where STx transactions commit sequentially. As in the S state, some effort is needed to block HTx transactions from beginning, or else this transition may be delayed indefinitely. Once the transition occurs, the remaining STx are guaranteed that (a) no new transactions can start, and (b) no other transactions are attempting to commit. Thus the STx can, in turn, validate their read sets and then either abort or write-back their updates. Once all pending STx have done so, the system returns to the NS state.

2.2 Key Properties

Earlier, we argued that a hybrid TM should ensure safety, limit use of hardware capacity for tracking metadata, and should enable some sort of fairness and progress for STx transactions. We briefly discuss each of these points in relation to the HyCo algorithm below:

Safety: The HyCo algorithm provides opacity [11] for all transactions. In Cohorts, opacity is achieved by ensuring that all shared memory is immutable whenever a transaction is in-flight. In HyCo, where there are two flavors of transaction, we modify this criteria: when a STx is in-flight, no concurrent HTx or STx transaction may perform an operation that modifies locations that have been, or may be, read by the in-flight STx. A concurrent STx transaction may progress up to its commit point, and may create pending changes to memory via the `TxWrite` function (as in Algorithm 5). However, it may not transition to the HC or SC state. Thus the concurrent STx cannot perform an operation that changes the memory visible to the in-flight STx. In this case, the property is achieved through the write buffering performed by STx. Similarly, a concurrent HTx may not transition to the HC state, where it can complete its transaction. Since HTx writes are buffered by the hardware until the commit point, the HTx cannot affect the behavior of the concurrent STx.

Now let us turn to an HTx transaction. Dalessandro et al. established that in a lazy Hybrid TM, an HTx transaction can experience an opacity violation if it overlaps with a concurrent STx commit [5]. The specific issue they identified is that a lazy STx might perform a partial write-back concurrent with the HTx, so that the HTx reads some of the STx's committed state, but not all of it. More generally, a sufficient condition is to prevent incomplete STx write-back from being visible to an HTx execution. In HyCo, this is achieved by (a) forbidding an STx from reaching the SC state until there are no concurrent HTx, and (b) attempting to commit STx in the HC state. In the HC state, the STx uses a hardware transaction to both validate and perform write-back; consequently the STx cannot expose its partial state: the entire set of updates becomes visible when the hardware transaction commits.

Metadata: As discussed above, HyCo does not use per-location metadata. Instead, it tracks the values read by a STx, and then validates those values directly. In this manner, it does not spend precious HTM resources tracking metadata. As a result, only a constant amount of metadata is needed for any HTx or STx transaction. As we will show in Section 3, the state machine can be implemented in a variety of ways, but the only global metadata for HyCo is related to the state machine, and it is only accessed at transaction boundaries. This results in a constant amount of metadata, and a constant overhead to access that metadata.

Fairness and Progress: HyCo supports a variety of approaches to ensuring fairness and progress. A few properties are relatively obvious: any transaction can be guaranteed to complete if it executes in Serial mode, and every read-only transaction will complete on its first attempt if it executes in STx mode. Beyond this, HyCo increases fairness by limiting the conditions in which a transaction cannot make progress. In particular, we have taken care to allow HTx to begin and commit when an STx is committing via HTM (HC state). Coupled with the simple existence of HC state, this limits the situations in which the system serializes. In addition, HyCo exposes two knobs for tuning progress. The first is a count of the number of HTx aborts before falling back to STx mode. The second is a count of the number of STx aborts before falling back to Serial mode. When combined with optional contention management at the beginning of the HC and SC states, there is ample opportunity to ensure that the most advantageous transactions are given priority.

3. Implementation

The primary challenge in implementing HyCo is to achieve a low-latency implementation of the state machine from Figure 1a. The most natural solution is to track each thread's state in a thread-private variable. However, doing so results in high latency in the common case: an HTx must check $O(\#Threads)$ locations at begin time. On the other hand, implementing each state as a counter is also a poor choice, since certain counters become contention hotspots.

Our solution, presented in Listing 1, is to split the state machine into three parts. First, there is a list of `Thread` objects, through which per-thread states for non-transactional, Serial, HTM, and STM mode can be discerned. This list is employed by all transactions. Second, we use an Integer and three Booleans to control when HTx can begin, and when they must immediately abort. Finally, three Integers and one Boolean are used to manage the states of STx and Serial transactions.

HTx Behavior: Algorithms 1- 3 describe how HTx, STx, and Serial transactions use these variables to safely transition among states. The default state is NS, in which HTx may begin and commit. Departing from this state requires an STx or Serial transac-

Listing 1: Hybrid Cohorts metadata. Global variables are clustered according to whether they assist in (a) coordinating all transactions, (b) coordinating HTx transactions, or (c) coordinating STx transactions.

```
Thread Variable Type:
  tx_state   : Enum{NO, S, HW, SW}   // state of thread's transaction (nontransactional, serial, HTx, STx)
  writes     : Map<addr,val>         // write set if this transaction is in STx mode
  reads      : Set<addr,val>         // read set if this transaction is in STx mode
  my_order   : Integer               // commit order of this transaction if it is in STx mode and using serial commit (SC)
  cp         : Checkpoint            // checkpoint of thread state, for retrying after STx aborts.
Global Variables:
  threads    : Set<Thread>   // A way of reaching each thread's per-thread vars

  started    : Integer       // Count of current active STx transactions
  ser_kill   : Boolean       // Flag to allow a Serial transaction to force immediate HTx aborts
  stx_kill   : Boolean       // Flag to allow an STx in SC mode to force immediate HTx aborts
  stx_comm   : Boolean       // Indicate that all STx are ready to commit

  cpending   : Integer       // Count of STx that are in the CP state
  order      : Integer       // Counter for ordering any STx that require SC mode to commit
  time       : Integer       // Second counter for STx that require SC mode to commit
  serial     : Boolean       // Token for granting a transaction permission to run in Serial mode
```

Algorithm 1: Begin and end instrumentation for HTx transactions. Parameters to `xabort` indicate the line to jump to after canceling a transaction attempt.

```
function TxBeginHTx()
    // Announce active HTx
1   tx_state ← HW
2   xbegin
    // Detect Serial and STx-SC transactions
3   if ser_kill ∨ stx_kill then
4       xabort(6)
5   return
    // Wait until no Serial or STx-SC transactions
6   tx_state ← NO
7   while ser_kill ∨ stx_kill do spin
    // Note: option to change to STx or Serial would go here
8   goto 1

function TxCommitHTx()
    // Commit if all STx in HC mode or no STx
1   if stx_comm ∨ started = 0 then
2       xend
3       tx_state ← NO
4       return
    // Cannot commit: in-flight STx or STx in SC mode
5   xabort(TxBeginHTx :: 6)
```

Algorithm 2: Begin instrumentation for STx transactions.

```
function TxBeginSTx()
1   cp ← make_checkpoint()
    // Try to set started while ¬serial and cpending = 0
2   if ¬serial then
        // Wait for committing STx, then announce self
3       while cpending > 0 do spin
4       atomic_incr(started)
        // Double-check that it's safe to start
5       if cpending > 0 ∨ serial then
6           atomic_decr(started)
7           goto 2
8       tx_state ← SW
        // Lazy cleanup of STx-SC flag
9       if stx_comm then stx_comm ← false
10  else goto 2
```

tion to begin. To keep overheads low for HTx, we subscribe to the ser_kill flag when an HTx begins. After becoming serial, but before accessing shared memory, a Serial transaction sets this flag to immediately abort all HTx. By optionally using the $threads$ set first (`TxBeginSerial` lines 4-5), we can opt to prioritize running HTx over new Serial transactions.

Since HTx can execute concurrently with STx, we do not repeat this behavior when STx begin. Instead, we must ensure that HTx do not commit when either (a) STx are between their begin and end, or (b) STx are performing serial commit. The stx_kill flag expresses condition (b). To handle condition (a), we use the $started$ and $cpending$ counters. When they are equal, every STx transaction has reached its commit point, and are trying to commit using HTM. In this case, HTx can commit, since the HTM will mediate conflicts. However, if they differ, then the HTx must abort.

STx Behavior: STx are expected to be less frequent than HTx, and also to be longer-running. Thus we tolerate some contention over metadata, since it reduces the number of locations that HTx must check. Specifically, we use the $started$ counter to track the number of STx that are not yet committed, and $cpending$ to track the number of STx that have reached their commit point. The $order$ and $time$ counters are used only for SC commits, to enforce one-at-a-time commit of large STx.

To maximize HTx concurrency with STx, we do not eagerly inform HTx of transitions between NS, SA, CP, and HC. Instead, we use the stx_comm flag, which indicates that STx have moved to HC state. While two values are needed to manage the SA-CP-HC transition, this specific pattern avoids aborts for HTx, since $started$ changes infrequently when stx_comm is set.

The additional transition to SC for serialized commit of STx is expected to be rarest. We employ the same technique as Serial

Algorithm 3: End instrumentation for STx transactions.

```
   function TxCommitSTx()
       // Read-only fast path
1      if writes = ∅ then
2          atomic_decr(started)
3          reads ← ∅
4          return
       // Wait until all STx ready to commit
5      atomic_incr(cpending)
6      while cpending < started do wait
       // STx will try to commit via HTM
7      if ¬stx_comm then stx_comm ← true;
8      xbegin
9      if reads.validate() then
10         writes.writeback()
11         xend
12         atomic_decr(started)
13         atomic_decr(cpending)
14         reads ← writes ← ∅
15         tx_state ← NO
16         return
17     else xabort(38)
18     // STx couldn't commit via HTM. Use serialized commit
19     my_order ← atomic_incr(order)
       // Lead thread waits for HC phase to end, others wait
       their turn
20     if order = 0 then
21         while order < started do spin
           // Optional: allow HTx to complete
22         for tx ∈ {threads − this_thread} do
23             wait_until(tx.tx_state ≠ HW)
           // Interrupt remaining HTx
24         stx_kill ← true
25     else while time ≠ my_order do spin

       // Writeback only if validation succeeds
26     if reads.validate() then writes.writeback()
27     else failed ← true
       // Let next STx commit
28     time ← time + 1
       // Clean up SC metadata; extra work for last thread
29     old ← atomic_decr(started)
30     if old = 1 then
31         stx_kill ← false
32         time ← order ← 0
33     atomic_decr(cpending);
34     tx_state ← NO
35     reads ← writes ← ∅
36     if failed then cp.restore()
37     else return
       // Reachable only on HC validation failure
38     atomic_decr(started);
39     atomic_decr(cpending);
40     reads ← writes ← ∅
41     tx_state ← NO
42     cp.restore()
```

Algorithm 4: Begin and end instrumentation for Serial transactions

```
   function TxBeginSerial()
       // Acquire serial lock
1      while ¬bool_cas(serial, false, true) do spin
2      tx_state ← S
       // Wait for committing STx
3      while started > 0 do spin
       // Optional: allow HTx to complete
4      for tx ∈ {threads − this_thread} do
5          wait_until(tx.tx_state = NO)
       // Interrupt remaining HTx
6      ser_kill ← true

   function TxCommitSerial()
       // Release lock, re-enable HTx
1      ser_kill ← false
2      serial ← false
3      tx_state ← NO
```

transactions, where a flag (stx_kill) is coupled with a traversal of the $threads$ set (`TxCommitStx` lines 22-23) to allow HTx to complete before serial STx.

A final complication is that, for the sake of fairness, we do not allow new STx to begin once any STx is ready to commit writes. This necessitates care in `TxBeginSTx`, since we must double-check $cpending$ after incrementing $started$.

Serial Behavior: Serial transactions are expected to be least common, and thus we are willing to incur overhead whenever one begins. In particular, after acquiring the $serial$ token, a transaction will wait for all active STx and HTx to complete. By setting the $serial$ flag first, it effectively prevents new STx. After allowing HTx to complete, it sets ser_kill to prevent additional HTx, at which point it can begin. Both flags are cleared when the transaction completes.

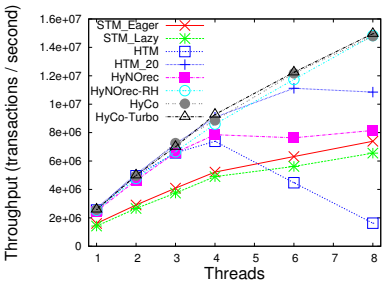
Per-Access Instrumentation: For completeness, Algorithm 5 presents the read and write instrumentation for the HyCo algorithm. As in the original Cohorts algorithm, per-access instrumentation is minimal, entailing neither metadata access nor memory fences. This is because (a) memory is immutable during STx execution, (b) Serial transactions execute in the absence of concurrency, and (c) HTx conflicts are mediated through the HTM, not through metadata.

4. Evaluation

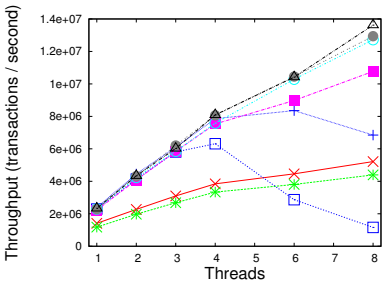
In this section, we evaluate the performance of HyCo. We consider microbenchmarks and the STAMP benchmark suite [22, 27]. Experiments are conducted on a machine with single-chip 3.40GHz Intel Core i7-4770 with 4 cores / 8 threads, running Ubuntu Linux 13.04, kernel 3.8.0-21, and a 4.9 GCC compiler with O3 and m64 flags. Results are the average of 5 trials.

To compare with HyCo, we consider the following TM implementations:

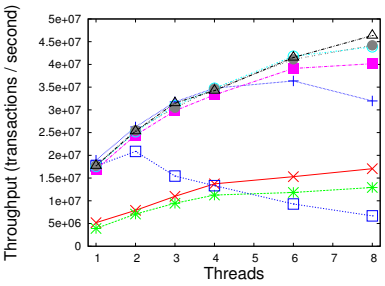
- **STM_Eager** is the default STM implementation provided with GCC. It is based on TinySTM's write-through algorithm [10]: write locks are acquired eagerly upon first write access to a location, undo logs track changes made by transactions, in case of an abort, and reads check the version number of locks. Conflicts are detected via validation, and a global counter is used



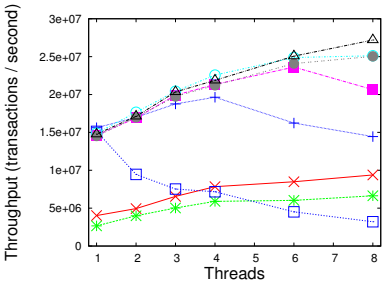
(a) Red/black tree microbenchmark with 20-bit keys and 80% lookup ratio



(b) Red/black tree microbenchmark with 20-bit keys and 33% lookup ratio



(c) Red/black tree microbenchmark with 8-bit keys and 80% lookup ratio



(d) Red/black tree microbenchmark with 8-bit keys and 33% lookup ratio

Microbenchmark	NS	HTx:HC	STx:RO	STx:HC	STx:SC	Serial
20-bit / 33% lookup	1.96M	50.4K	44	37	9	0
20-bit / 80% lookup	2.09M	7.6K	72	8	7	0
8-bit / 33% lookup	5.00M	381K	569	180	70	28
8-bit / 80% lookup	9.32M	25K	1.98K	344	330	14

(e) Frequency of each type of commit for four microbenchmarks. Data is taken from a one-second execution with four threads. The workload was heterogeneous, and values were reported by a randomly chosen thread.

Figure 2: Microbenchmark performance

Algorithm 5: Hybrid Cohorts read and write instrumentation

```
function TxRead(addr)
    // Serial and HTM fast-path
1   if tx_state ∈ {S, HW} then
2       return *addr
    // Handle read-after-write
3   if addr ∈ writes then
4       return writes[addr]
    // Read the value, and log it for commit-time validation
5   v ← *addr
6   reads ← reads ∪ {⟨addr, v⟩}
7   return v

function TxWrite(addr, val)
    // Serial and HTM fast-path
1   if tx_state ∈ {S, HW} then *addr = val
    // Buffer the write until commit time
2   else writes ← writes ∪ {⟨addr, v⟩}
```

to avoid most validation during transaction execution. Writer transactions use *quiescence* to achieve privatization safety.

- **STM_lazy** is a commit-time locking version of STM_Eager. Writes are stored in a redo log, which is implemented as a hash table of 64-byte blocks. Write locks are acquired at commit time. In all other regards, the implementation is the same as STM_Eager. The main value of STM_lazy in our experiments is in identifying overheads related to redo logs.
- **HTM:** a) **HTM** is the default HTM implementation provided with GCC. Transactions attempt to run using Intel RTM, and fall back to a serial execution mode after two consecutive HTM aborts. b) **HTM_20** modifies the above HTM implementation so that fallback to serial mode occurs after 20 attempts.
- **HyNOrec:** There are two suggestted implementations that do not require non-transactional reads in the original HyNOrec proposal [5]. We present the P-counter version in Microbenchmark and STAMP suite, as it outforms the 2-location version.
- **HyNOrec_RH** is the most recent reduced hardware Hybrid NOrec implementation adopted from [21]. We did not apply the complier static analysis to reduce the instrumentation of read-only hardware transactions, for fair comparison with other TM implementations, which could all benefit from such analysis.

Our version of HyCo employs the following optimizations:

- **Lightweight Privatization Safety**: Since writer transactions either (a) commit via HTM, or (b) commit during the serialized (SC) phase, there is no need for out-of-band privatization safety. Our HyCo implementation thus skips GCC's quiescence mechanism.
- **Lightweight Irrevocability**: GCC achieves serial execution via adaptivity. In contrast, Serial mode is a first-class behavior within HyCo, and thus we can avoid interaction with a custom readers/writer futex on every transaction.
- **Un-instrumented HTM Loads and Stores**: GCC creates two code paths for transactions: one suitable for STM, in which loads and stores of shared memory are transformed into function calls, and one suitable for HTM, in which loads and stores are not instrumented. Given the lightweight instrumentation in Algorithm 5, HyCo is able to use the latter approach for HTx.

We also present HyCo-Turbo, which additionally provides a lightweight software path. Since it is natural for a STx to be aware of the number of existing software transactions, a STx can turn into turbo mode by sealing the cohort early if it a) confirms that no other STx is running and b) successfully aborts all running HTx. A turbo mode STx does not require further instrumentation on reads/writes, or validation at commit time.

We set HyCo thresholds as follows: An HTx transaction will switch to STx mode after 20 failed attempts to commit. An STx transaction will switch from committing in HC mode to committing in SC mode after 2 failed attempts. Fall-back to Serial mode occurs after 5 failed commit-time validations by an STx transaction.

4.1 Microbenchmark Performance

We begin our evaluation by looking at microbenchmark performance. We consider four configurations of a red-black tree test, taken from the RSTM library [19]. Configurations differ in terms of the range of keys present in the tree, and the ratio of lookups to inserts and removes (insert and remove operations are always performed in equal amounts). In all cases, the tree is pre-populated to 50% full. The charts in Figure 2 present throughput as the average over five trials.

At one thread, htm and HyCo performance are identical, and uniformly better than STM. This is expected, since transactions are small enough to complete without exceeding hardware capacity. As we increase the thread count, and contention increases, we see a significant shift: the rapid fall-back to serial mode hurts htm, both because it is too early, and because it limits concurrency. Even htm_20, our version of the GCC htm that retries 20 times before falling back to serial mode, cannot keep up with HyCo: the opportunity cost of serialization, even after 10 failed attempts, is simply too high. This is especially true for the highest contention configuration (8-bit keys, 33% lookup), where htm_20 performance degrades beyond 4 threads.

The performance of eager and lazy STM was also surprising in this experiment. As expected, both scale well, and their use of validation affords for fewer aborts than the "requester wins" conflict resolution strategy [1] of HTM. However, latency is high: they incur a function call on every load and store, and lazy pays even more due to accesses to the write log on every load and store. Furthermore, STM scales worse than HyCo. There are two causes: the overhead of quiescence, and the cost to support irrevocability via mode switching.

To gain a better understanding of why HyCo scales so much better than GCC's htm, we measured the frequency of each type of commit for the HyCo execution of the benchmarks. While the majority of transactions can commit using HTM (NS state), there are nontrivial instances in which transactions fall back to STx mode. While STx transactions are rare, the number of HTx transactions that commit concurrently with STx (i.e., when the STx is in HC mode) is high (indicated by HTx:HC). This confirms that the opportunity cost of serializing is high: in htm and htm_20, every fallback to STx becomes a fallback to Serial, and all concurrency among HTx:HC, STx:RO, and STx:HC is lost. Worse, these often result in a cascade of transactions that fall back to serial mode. This is most unfortunate for read-only STx, which otherwise are concurrent.[1]

4.2 STAMP Performance

STAMP performance is shown in Figure 3. Unlike microbenchmark experiments, STAMP performance is shown as total time. The expectation is that more threads will result in a decreased execution time.

As in previous work [27], we observe that the Labyrinth benchmark shows little variation among algorithms. This is a consequence of the benchmark being rewritten to match the Draft C++ TM Specification: transactions no longer comprise a significant portion of execution time. As has become standard practice, we do not report Bayes performance, since the benchmark exhibits nondeterministic behavior.

Among the remaining 8 benchmark configurations, we see two trends emerge. First, on workloads with high contention, such as KMeans-HC and Vacation-HC, HTM performs best at one thread, but its performance degrades as the thread count increases, due to its reliance on serialization to ensure progress after repeated aborts. In contrast, HyCo manages to maintain its performance as contention increases, by falling back to STx. This trend peters out to some degree at 8 threads for Vacation-HC, due hardware multithreading effects: with four cores and 8 hardware threads, transaction write capacities are effectively halved at 8 threads. The low-contention variants of KMeans and Vacation show that as contention decreases, HTM is able to perform on-par with HyCo, but HyCo remains a superior choice overall. The same is true for SSCA2, where small transactions run bottleneck-free in HyCo and HTM.

The second trend is shown by Genome, Intruder, and Yada. In these benchmarks, HyCo incurs higher latency than HTM in order to interact with its write set. Recall that for STx transactions, HyCo must perform a lookup on each read, and must buffer its writes in a manner compatible with lookup. This necessitates a more complex data structure (hash of blocks with masks) than the undo log used by eager STM and the HTM fall-back. Consequently, we see that STM_lazy is a constant factor slower than STM_eager, and that HyCo similarly incurs high overhead. The problem is most extreme in Yada, where the combination of (a) aborting as HTx before falling back to STx; and (b) incurring write set overhead; results in an insurmountable slowdown at all thread levels. Similarly, in Genome and Intruder, the frequency of lookups creates a significant overhead.

In effect, these results confirm claims made by Kestor et al. [15]. In their work, they showed that a proper implementation of lazy STM in GCC incurred higher constant overhead than previously believed. While we believe our lazy TM implementations to be more optimal than theirs, the problem remains: the baseline for lazy STM is worse than eager, especially in unmanaged languages.

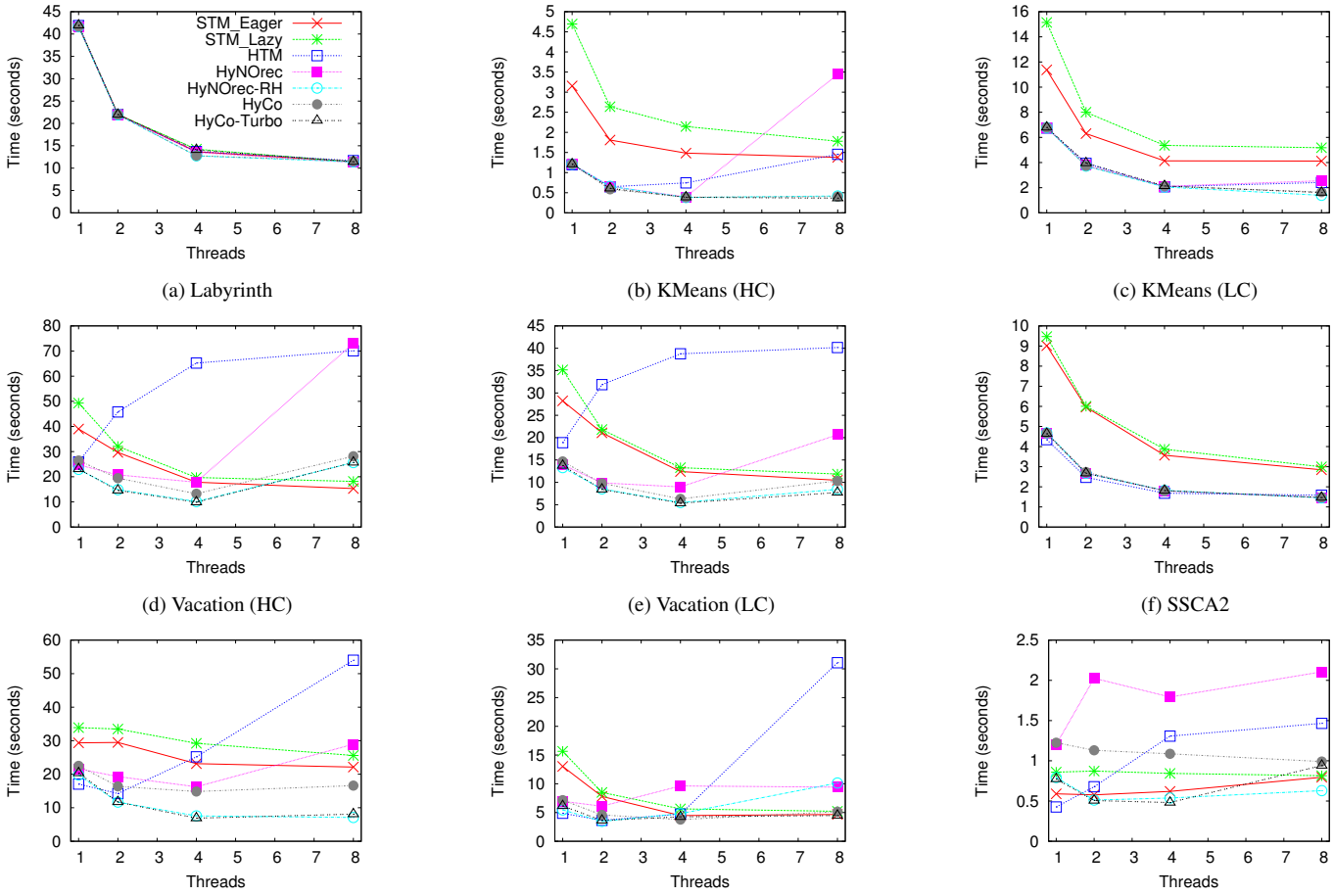
On this last point, we conducted experiments with two different write set implementations: a hash table and an unbalanced BST. These tests showed that the data structure itself was not the slowdown. Rather, the cost came from manipulating bit masks in order to handle the case where a byte is accessed as part of multiple accesses of varying granularity (e.g., the byte is written, and then the enclosing word is read). These costs may be fundamental to lazy hybrid TM. If so, prior work that did not assume a compiler interface to the hybrid TM, such as Hybrid NOrec [5] may need to be reevaluated.

5. Conclusions and Future Work

In this paper, we presented the Hybrid Cohorts (HyCo) algorithm. The foundation of HyCo is a state machine that governs when transactions may begin, and when they may attempt to commit. This state machine ensures correctness (opacity) by presenting each transaction with the illusion that memory is immutable during program execution. Given such a guarantee, hardware-mode transactions can run virtually instrumentation-free, all inter-thread coordination is constrained in the transaction begin and commit functions, and even software-mode transactions can use hardware TM support to accelerate their commit operations.

Due to the relative newness of Hybrid TM for real-world systems, there is no common testbed for comparing different algorithms. We hope that such a platform will emerge soon, and that it will be built atop GCC so as to address real-world concerns. Indeed, our experiments show that redo logs can have a higher cost than previously reported, and this cost affects HyCo transactions

[1] Note that we do not use compiler information to identify read-only transactions; had we used this information, an optimized STx:RO fastpath would be possible. In the tree workloads, many read-only transactions are not statically identifiable (e.g., an insert of a key that already exists), and thus such an optimization would have less value than it might otherwise seem.



(a) Labyrinth (b) KMeans (HC) (c) KMeans (LC)

(d) Vacation (HC) (e) Vacation (LC) (f) SSCA2

(g) Intruder (h) Genome (i) Yada

Figure 3: STAMP performance. HC and LC refer to high- and low-contention command-line configurations.

that fall back to software mode. Whether this cost can be mitigated, and how this cost might affect previously published hybrid TM, are exciting areas for future research.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[2] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, AB, Canada, Aug. 2014.

[3] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[4] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[5] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[6] L. Dalessandro, M. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[8] D. Dice, T. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of Lazy Subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory*, Paris, France, July 2014.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of*

the 13th ACM Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, Feb. 2008.

[11] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[12] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[13] Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.

[14] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[15] G. Kestor, L. Dalessandro, A. Cristal, M. Scott, and O. Unsal. Interchangable Back Ends for STM Compilers. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[16] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[17] Y. Lev and J.-W. Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[18] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[19] V. J. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, June 2006.

[20] A. Matveev and N. Shavit. Reduced Hardware NOREC: An Opaque Obstruction-Free and Privatizing HyTM. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[21] A. Matveev and N. Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, Mar. 2015.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[23] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[24] M. Moir. Unpublished Manuscript, July 2005.

[25] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[26] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[27] W. Ruan, Y. Liu, and M. Spear. STAMP Need Not Be Considered Harmful. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[28] W. Ruan, Y. Liu, C. Wang, and M. Spear. On the Platform Specificity of STM Instrumentation Mechanisms. In *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, Shenzhen, China, Feb. 2013.

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

[30] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, Dec. 2006.

[31] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[32] A. Shriraman, M. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[33] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[34] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[35] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.