Hybrid Transactional Memory Revisited

Wenjia Ruan, Michael Spear

Hybrid Transactional Memory Revisited

Wenjia Ruan and Michael Spear

Department of Computer Science and Engineering
Lehigh University
{wer210,spear}@cse.lehigh.edu

Abstract. Hybrid Transactional Memory (TM) uses available hardware TM resources to execute language-level transactions, and falls back to a software TM implementation for those transactions that cannot complete in hardware. Ideally, a hybrid TM would allow hardware and software transactions to run concurrently, but would not waste hardware TM resources on coordination between the two classes of transactions. In addition, it should scale well, incur little latency, offer strong safety guarantees, and provide some degree of fairness.
We introduce a new hybrid TM algorithm, "Hybrid Cohorts", in which hardware transactions do not modify global metadata, and software transactions have extremely low per-access overhead. The tradeoff is that hardware transactions cannot commit while software transactions are in flight. Evaluation on an 8-thread Intel Haswell CPU shows competitive performance with the current state-of-the-art. Furthermore, it does so while providing acceptable levels of fairness and safety, and offering opportunities for hardware acceleration.

1 Introduction

Since the time when Hybrid Transactional Memory (TM) was first proposed [6], hardware TM (HTM) support has become available in microprocessors from IBM [11, 21] and Intel [10]. These HTM systems are "best effort", meaning that they do not guarantee that they will successfully commit *any* transaction attempt. Failure may arise for many reasons, to include conflicts with other transactions, memory footprints that exceed the HTM capacity, system calls, and timer interrupts. The goal of Hybrid TM (HyTM) is to exploit best-effort HTM whenever possible, and fall back to software TM (STM) when a transaction cannot complete in hardware [6]. This approach promises to scale well and incur low latency when most transactions complete in hardware, with worst-case overhead and scalability comparable to the underlying STM.

The traditional approach to implementing HyTM is to begin with an STM, and try to accelerate it using HTM. Early STM algorithms required interaction with per-location metadata, and hybrid versions of these algorithms wasted limited hardware capacity on this metadata [6, 12, 16]. Worse yet, false sharing of cache lines that held metadata could result in additional HTM aborts, and increased fallback to the STM path. The use of NOrec STM [5] as a baseline enabled HyTM algorithms to avoid per-access overheads. In NOrec-based HyTM [4, 14, 16], a sequence lock serializes the commit of the STM, and all conflicts are detected by comparing the values read by transactions. However, NOrec-based HyTM algorithms suffer from a scalability bottleneck, since

hardware transactions must read, and often write, the sequence lock. Aborts from these accesses could be avoided if the hardware allowed nontransactional accesses [4], but the accesses themselves are necessary. Furthermore, if these accesses are delayed until the end of the transaction [2, 3], the TM ceases to provide the minimum safety requirement of opacity [9], and it can admit erroneous behavior [7]. However, "eager subscription" to the metadata for coordinating hardware and software transactions causes all hardware transactions to abort on any software commit.

The most recent innovation in HyTM is to add hardware acceleration to the STM path, as in Reduced Hardware NOrec (RHNOrec) [14]. The resulting "reduced transaction" technique transforms certain software transactions into hardware transactions, thereby avoiding fallback to a slow STM. The current state of the art achieves performance comparable to Hybrid NOrec, but does not require nontransactional loads.

A common assumption among HyTM algorithms is that STM and HTM transactions should coexist at any time, with neither favored over the other. In contrast, PhaseTM [13], required all transactions to use same mode, whether HTM, STM, or serialized on a single lock. Mode switches were expensive, but in return the HTM mode had no overhead for interacting with STM. The most popular HyTM in practice today is a PhaseTM that switches between HTM mode and a single global lock [24]. If we accept that HTM capacities are more likely to increase than to decrease, then we may assume that STM fall-back will grow increasingly rare. However, as core counts rise, fall-back to a single lock becomes increasingly untenable. These observations motivate our approach. We seek to make the common case (HTM) as fast as possible, by avoiding interaction with (unlikely) concurrent software transactions. When a software transaction is needed, we want it to finish as quickly as possible, to limit its impact on current and future hardware transactions. We also require the HyTM to be opaque.

The innovation we propose is to prioritize software transactions while they are running, by augmenting the Cohorts algorithm [18]. In Cohorts, transactions block at their commit point, until such time as all threads are either (a) ready to commit a transaction, or (b) not executing a transaction. This allows software transactions to avoid any high-latency global metadata accesses during execution. In Hybrid Cohorts (HyCo), we prevent hardware transactions from committing when software transactions are in-flight. We also apply the reduced transaction technique to the Cohorts commit phase, which prevents blocking and eliminates a bottleneck from Cohorts STM. The net effect is an opaque HyTM that scales well and avoids bottlenecks for hardware transactions.

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 HTM resources throughout their execution. In Section 4, we present the results of performance experiments. Section 5 concludes and discusses future research directions.

2 The Hybrid Cohorts Approach

HyTM algorithms that descend from NOrec share two key properties: First, the opacity of each software transaction (STx) is preserved by requiring every hardware transac-

tion (HTx) increment a global counter on hardware transaction commit; the counter is checked by every STx on every read of shared data. (Note that the increment may be elided if it can be proven that no STx exist.) Second, to prevent an HTx from observing inconsistent state, it must not perform reads or commit during the interval when an STx is committing its writes. In Hybrid NOrec, the second property was kept performant via a set of per-thread counters, or via special instrumentation on every load by an HTx.[1] In RHNOrec, the overhead of second property is avoided by executing the suffix of all but the largest STx (the interval between its first write through the completion of their commit) as a hardware transaction.

HyCo takes a different approach to these challenges. First, to ensure the opacity of STx, HTx are not allowed to commit whenever any STx is between its begin and commit points. In the absence of nontransactional loads, this necessitates that HTx abort if they reach their commit point and then observe an active STx. However, it also means that HTx do not need to modify shared variables at commit time in order to notify concurrent STx of the need to validate. Second, to ensure that HTx do not observe inconsistent state, the commit-time validation and writeback of STx is first attempted as a "reduced" hardware transaction. If the reduced transaction cannot commit, the STx waits until all concurrent HTx complete, and then performs serialized writeback. Note that by also blocking STx commits when there are concurrent in-flight STx, validation by in-flight STx is no longer necessary: during any STx execution, memory is immutable. Less instrumentation is required within STx, which reduces STx execution time and should limit the impact of HTx blocking or aborting at their commit point due to concurrent STx execution.

To make the behavior of transactions in HyCo more clear, Figure 1 presents a state machine to describe the behavior of transactions. There are 6 states:

- No STx (NS): This is the default state of the system. In this state, HTx may begin, commit, and abort.
- One Serial Transaction (S): Only one transaction is running, and it cannot be aborted. This allows a transaction to perform I/O [20, 23].
- STx Active (SA): At least one STx has started but has not yet reached its commit point. No transaction is allowed to commit changes to shared state.
- STx Commit Pending (CP): A proper, nonempty subset of STx have reached their commit points. No transaction is allowed to commit changes to shared state. To prevent extended blocking of ready-to-commit STx, new STx may not begin.
- STx Hardware Commit (HC): All STx have reached their commit point, and are attempting to commit via reduced hardware transactions. STx cannot begin, since shared memory may be changed.
- STx Software Commit (SC): A proper, nonempty subset of STx cannot commit via reduced hardware transactions, and are in the process of committing sequentially. No new transactions may begin.

State transitions are triggered by the following transaction events. With the exception of HTx Abort, these events occur on the boundaries of transactions:

- HTx Begin: A thread attempts to begin a transaction in HTx mode.

[1] This instrumentation requires nontransactional loads from within the transaction, which are not supported by TSX.

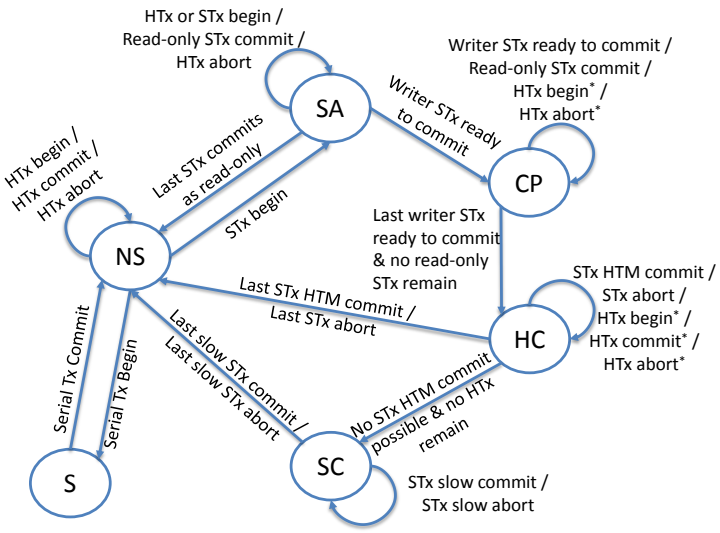


Fig. 1: State transitions of the Hybrid Cohorts algorithm.

- HTx Commit: An HTx commits its changes to shared memory.
- HTx Abort: An HTx fails due to conflicts with other transactions.
- Serial Begin: A thread begins a serial transaction.
- Serial Commit: A serial transaction commits.
- STx Begin: A thread attempts to begin a transaction in STx mode.
- STx Commit (read-only): When an STx reaches its commit point, if it determines that it is read-only, it does not need to block in order to commit, since its commit cannot affect concurrent STx.
- STx Ready to Commit: An STx reaches its commit point, and is not read-only.
- STx Abort: An STx, during its commit operation, validates and determines that it conflicts with a committed transaction.
- STx HC Commit: An STx commits using a reduced hardware transaction.
- STx HC Fail: An STx cannot commit via a reduced hardware transaction (e.g., because it has accessed too many locations).
- STx Slow Commit: An STx commits via the serialized commit protocol.

In Figure 1, 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. Labels marked with an asterisk(*) are optional, and provide more opportunities for HTx to make progress. The order in which transactions are scheduled in the HC and SC modes can be governed by arbitrary contention management policies. For this discussion, we assume that the contention manager randomly chooses the order in which transactions attempt to commit in SC, and transactions attempt to commit immediately upon transition to HC.

Following Dalessandro et al. [4], the underlying STM uses value-based validation. STx do not update memory directly, but instead buffer their writes in a private log,

and then replay them at commit time. These decisions ensure that HTx do not observe uncommitted state during their execution, and need not check per-location metadata on each load and store. 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 global metadata, and a constant overhead for HTx to access that metadata.

Section 3 discusses mechanisms for achieving those transitions that are not obvious. For example, the transition from NS to S can be achieved by either (a) forcibly aborting any in-flight hardware transactions, or (b) setting a flag to prevent subsequent HTx and STx from beginning, and then waiting for the system to be in the NS state with no HTx running. Our goal is to achieve each transition without causing HTx conflicts on state machine metadata. As an example, an HTx can start as long as the system is not in S or SC state, but it need not know the exact system state until it reaches its commit point. If the state changes during HTx execution (e.g., from NS to SA, or even from HC to NS), the HTx should not immediately abort. At commit time, the HTx should be able to quickly check the precise state, and then self abort if necessary (e.g., if the state is CP).

HyCo provides opacity [9] for STx by ensuring that when an STx is in-flight, no concurrent HTx or STx may perform an operation that modifies locations that have been, or may be, read by the in-flight STx. A concurrent STx may reach its commit point, but may not transition to the HC or SC state, and since its writes are buffered, it cannot modify memory. (Note that a read-only STx may commit during this time, but by definition it does not modify shared memory.) Thus no concurrent STx can perform an operation that changes the memory visible to the in-flight STx. Similarly, a concurrent HTx may not transition from SA to HC, where it can complete its transaction. In this case, the HTM provides write buffering for the not-yet-committed HTx.

When the underlying HTM is opaque, Dalessandro et al. established that in a lazy HyTM, an HTx transaction can only experience an opacity violation if it overlaps with an STx commit [4]. Specifically, the 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. 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 a reduced hardware transaction. In the HC state, the reduced transaction validates and performs write-back; consequently the STx cannot expose its partial state: the entire set of updates becomes visible when the hardware transaction commits.

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, every read-only transaction will complete on its first attempt in STx mode, and an STx will not abort unless some other HTx or STx commits. Our implementation 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. Additional scheduling decisions can be made when transitioning out of the CP state (i.e., by allowing a high priority transaction to abort all HTx, transition directly to SC, and commit first).

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
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 if STx in SC mode
cp        : Checkpoint          // checkpoint of thread state, for STx aborts

Global Variables:
threads   : Set<Thread> // For reaching each thread's per-thread variables

started   : Integer     // Count of current active STx transactions
ser_kill  : Boolean     // Allow a Serial transaction to force immediate HTx aborts
stx_kill  : Boolean     // 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 to order STx in SC mode
time      : Integer     // Second counter for STx in SC mode
serial    : Boolean     // Token for transitions to Serial mode
```

3 Implementation

The primary challenge in implementing HyCo is to achieve a low-latency implementation of the state machine from Figure 1. Our solution employs the metadata in Listing 1 to split the state machine into three parts. First, there is a list of `Thread` objects, through which per-thread states for non-transactional (NO), Serial, HTM, and STM modes can be discerned. 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. The code in Algorithm 1 uses these variables in flag-based and Dekker-style coordination.

The default system state is NS, in which HTx may begin and commit. Departing from this state requires an STx or Serial transaction 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 is 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 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

Algorithm 1: The HyCo Algorithm. The parameter to `xbegin` (overridden by `xabort`) indicates the location to jump to when a hardware transaction aborts.

```
function TxBeginHTx()
1   tx_state ← HW    // Announce active HTx
2   _xbegin(5)
    // Detect Serial and SC modes
3   if ser_kill ∨ stx_kill then xabort(5)
4   return
    // Unannounce, wait if Serial or SC mode
5   tx_state ← NO
6   while ser_kill ∨ stx_kill do spin
    // Execute as STx or switch to Serial?
7   if switch_mode() then return
8   goto line 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: SA or CP mode
5   xabort(TxBeginHTx() line 5)

function TxBeginSTx()
1   cp ← make_checkpoint()
    // Increment started only if NS or SA mode
2   while serial ∨ cpending > 0 do spin
3   atomic_incr(started)
    // Double-check NS or SA mode
4   if cpending > 0 ∨ serial then
5       atomic_decr(started)
6       goto line 2
7   tx_state ← SW
    // Lazy cleanup of STx-HC flag
8   if stx_comm then stx_comm ← false

function TxBeginSerial()
    // Acquire serial lock, wait for STx to finish
1   while ¬cas(serial, false, true) do spin
2   tx_state ← S
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)
6   ser_kill ← true    // Abort remaining HTx

function TxCommitSerial()
    // Re-enable HTx, then release serial lock
1   ser_kill ← false
2   serial ← false
3   tx_state ← NO

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

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 spin
    // Move to HC mode, commit STx via HTM
7   if ¬stx_comm then stx_comm ← true;
8   _xbegin(18)
9   if reads.validate() then
10      writes.redo()
11      _xend
12      atomic_decr(started)
13      atomic_decr(cpending)
14      reads ← writes ← ∅
15      tx_state ← NO
16      return
17  else xabort(37)
    // Serialized commit
18  my_order ← atomic_incr(order)
    // Lead thread waits for HC to end
19  if order = 0 then
20      while order < started do spin
        // Optional: allow HTx to complete
21      for tx ∈ {threads − this_thread} do
22          wait_until(tx.tx_state ≠ HW)
23      stx_kill ← true    // Abort remaining HTx
    // Other threads wait their turn
24  else while time ≠ my_order do spin
    // Writeback only if validation succeeds
25  if reads.validate() then writes.redo()
26  else failed ← true
27  time ← time + 1    // Let next STx commit
    // Last thread moves metadata back to NS
28  old ← atomic_decr(started)
29  if old = 1 then
30      stx_kill ← false
31      time ← order ← 0
32  atomic_decr(cpending);
33  tx_state ← NO
34  reads ← writes ← ∅
35  if failed then cp.restore()
36  else return
    // Reachable only on HC validation failure
37  atomic_decr(started);
38  atomic_decr(cpending);
39  reads ← writes ← ∅
40  tx_state ← NO
41  cp.restore()
```

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. This reduces the frequency of reads of the $started$ and $cpending$ counters by HTx. To avoid additional synchronization on STx commit, we defer resetting stx_comm to `TxBeginSTx`. Doing so is immaterial to HTx behavior, since HTx can progress in full from both the NS and HC modes.

The additional transition to SC for serialized commit of STx is expected to be rarest. We employ the same technique as Serial 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 transactions are least common, justifying more overhead whenever one begins. After acquiring the $serial$ token, a transaction will wait for all active STx and HTx to complete. Setting the $serial$ flag first effectively prevents new STx. After allowing some 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.

For completeness, Algorithm 1 also presents the read and write instrumentation for the HyCo 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

We now evaluate the performance of HyCo using microbenchmarks, the STAMP benchmarks [15, 17] and Memcached [19]. Experiments were 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. We compare the following TM implementations:

- **STM_Eager** is the default STM provided with GCC. It is based on write-through TinySTM [8], using per-location ownership records, undo logging, encounter-time locking, and read set validation. The algorithm is opaque, and uses commit-time quiescence [22] to achieve privatization safety.
- **STM_Lazy** is a commit-time locking version of STM_Eager. Writes are stored in a redo log, implemented as a hash table of 64-byte blocks. Ownership records are acquired at commit time. STM_Lazy exposes overheads related to redo logs.
- **HTM** is the default HTM implementation provided with GCC. It is a PhaseTM that falls back to serial mode after two consecutive HTM aborts. **HTM_20** does not fall back to serial until 20 consecutive aborts.
- **HyNOrec:** is the "P-counter" version of Hybrid NOrec, which does not require nontransactional loads [4]. For Memcached, we also report the "2-counter" version.
- **HyNOrec_RH** is the most recent reduced hardware Hybrid NOrec implementation [14]. We did not apply 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.

In the interest of fairness, we observe the following differences among systems:

- **Privatization**: STM_Eager and STM_Lazy require quiescence-based privatization, whereas the HTM and hybrid algorithms do not. This can result in better scalability at high thread counts for hybrids, since they do not require blocking at commit time.
- **Mode Switching**: The Serial modes of our HyTMs are achieved via spin waiting, whereas the GCC-based algorithms use a more heavyweight blocking mechanism.
- **Un-instrumented HTM Loads and Stores**: GCC creates two code paths for transactions: an STM path, in which loads and stores of shared memory are instrumented, and an HTM path in which they are not. HyNOrec_RH and HyCo HTx benefit from this lower-latency code path.

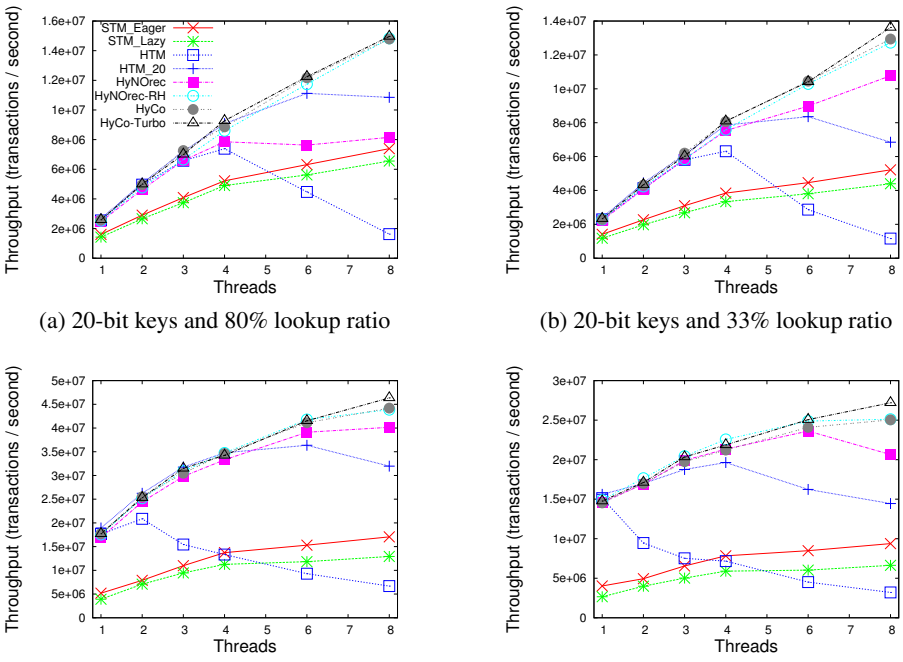
We set HyCo thresholds as follows: An HTx transaction will switch to STx mode after 20 failed attempts to commit. An STx transaction will switch from committing in HC mode to committing in SC mode after 2 failed attempts. Fall-back to Serial mode occurs after 5 failed commit-time validations by an STx transaction. We also present HyCo-Turbo, which eagerly transitions STx to a mode where they run in isolation and perform all updates in-place. An STx can invoke turbo mode by a) confirming it is the only STx, (b) blocking new STx from starting, and (c) aborting all concurrent HTx. A turbo mode STx effectively executes as a Serial transaction.

4.1 Microbenchmark Performance

Figure 2 presents four red-black tree microbenchmarks. The experiments differ in terms of the range of keys present in the tree and the ratio of lookups to inserts and removes (inserts and removes are always performed in equal amount). All trees are pre-populated to 50% full. At one thread, HTM and HyCo performance are identical, and uniformly better than STM. This is expected, since most 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 20 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.

Eager and Lazy STM scale well, and their use of validation affords for fewer aborts than 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 many loads (these costs are only incurred in HyCo's STx mode). Furthermore, STM scales worse than HyCo, due to the overhead of quiescence, and the cost to support irrevocability via mode switching.

To gain a better understanding of the importance of HyTM versus PhaseTM, 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), up to 9% of HTx commit in HC mode. Thus while fallback to serial (for GCC) or STx (for HyCo) is rare, the impact on concurrent HTx can be significant. In HTM and HTM_20, every fallback to STx becomes a fallback to Serial, and all concurrency is lost.



(a) 20-bit keys and 80% lookup ratio

(b) 20-bit keys and 33% lookup ratio

(c) 8-bit keys and 80% lookup ratio

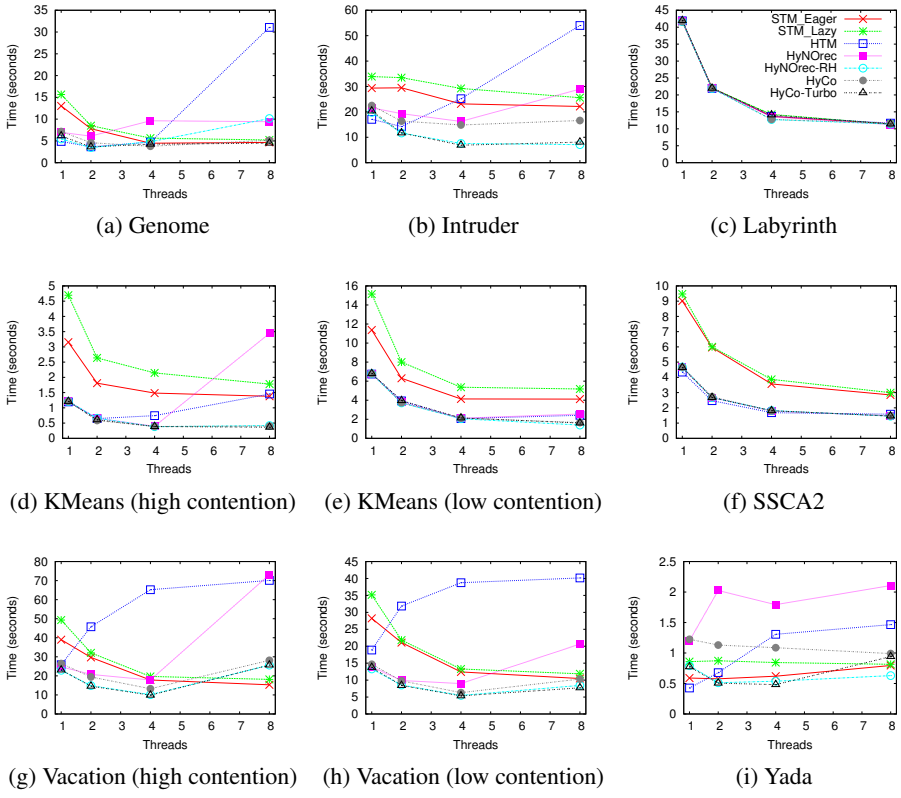
(d) 8-bit keys and 33% lookup ratio

Fig. 2: Red/black tree microbenchmarks

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. We do not report Bayes performance, since it exhibits non-deterministic behavior. We also note that since Labyrinth was rewritten to match the Draft C++ TM Specification [17], it shows little variation among algorithms because transactions no longer comprise a significant portion of execution time.

Among the remaining 8 benchmark configurations, we see two trends emerge. First, on workloads with high contention, such as KMeans and Vacation, 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.



(a) Genome (b) Intruder (c) Labyrinth

(d) KMeans (high contention) (e) KMeans (low contention) (f) SSCA2

(g) Vacation (high contention) (h) Vacation (low contention) (i) Yada

Fig. 3: STAMP performance

The second trend is shown by Genome, Intruder, and Yada, where 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 higher overhead. The problem is most extreme in Yada, where HTx abort late in their execution, fall back to STx, and then incur write set overhead. Similarly, in Genome and Intruder, the frequency of lookups creates overhead.

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 are shared by all of our Hybrid TM implementations.

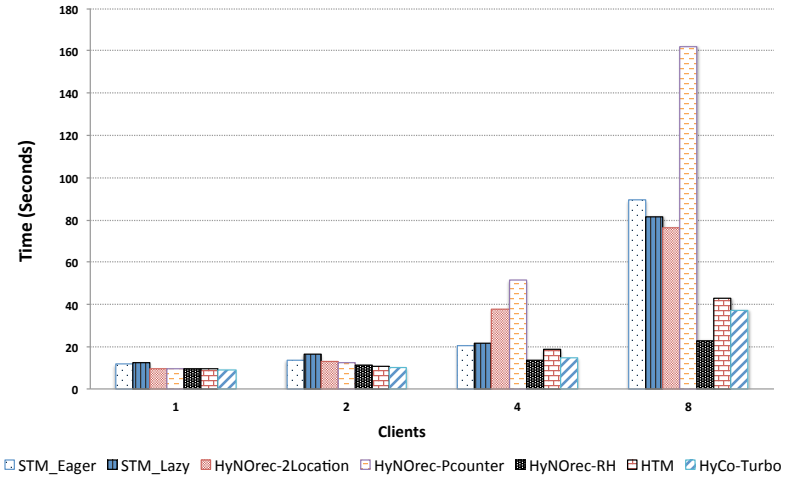


Fig. 4: Memcached Performance

Client Count	NS	HTx:HC	STx:RO	STx:HC	STx:SC	STx:Turbo	Serial
2	7.48M	65	0	85	10	4.4k	0
8	5.44M	544.3K	0	195.7k	72.2k	295.9k	0

Client Count	STx aborts in HC/SC mode	Time spent spinning in TxBegin
2	3.04%	1.55%
8	42.64%	25.97%

Table 1: Frequency of each type of commit for Memcached at different client counts with HyCo-Turbo. Values were reported by a randomly chosen thread.

4.3 Memcached Performance

Lastly, we evaluate all TM implementations on memcached. We followed the experiment configuration of Ruan et.al [19]. The configuration results in a number of operations proportional to the number of threads: flat curves indicate perfect scaling, higher values represent slowdown. The results are presented in Figure 4. Note that the number of transactions per thread is not constant, due to the use of transactions to read shared memory when spin waiting.

Memcached presents a number of interesting behaviors. First, we observed that Hybrid NOrec with P counters, where P is the number of threads, performed worse than the two-counter version. In our prior experiments, P-counter was superior. Second, HTM and HyCo performance were close, and HyNOrec-RH performed best at 8 threads. This was the first instance in which HyCo did not match or outperform HyNOrec-RH. Table 1 provides more detail. The data was collected by sampling one thread's behavior during a multithreaded execution. The top half shows that at high thread counts, a larger number of transactions execute as STx, and that they also cause more HTx to commit

in the HC mode. We also see that the HyCo "turbo mode" optimization is valuable, accounting for more than 5% of commits.

The bottom half of the table shows that some overhead is a direct consequence of the HyCo design. At 8 threads, almost half of STx abort at commit time. These aborts imply wasted work: if STx could detect conflicts earlier, they might not spend as much time on attempts that ultimately failed. Furthermore, a quarter of execution time for the sampled transactions was spent waiting to start transactions. This time was due to both STx that could not start while other STx were in HC, SC, or turbo modes, and HTx that could not start due to STx in SC or turbo modes.

There are a number of solutions to these problems. First, as HTM capacities increase, these problems will naturally diminish as more transactions run as HTx. Second, it is likely that the knobs controlling turbo mode require more intelligence, and perhaps ought to adapt to the thread count and workload. Third, there is clearly an opportunity for advanced transaction scheduling and contention management. When so much time is already spent waiting at transaction begin, techniques such as [1] and [25] should not introduce latency. We leave further exploration of this topic as future work.

5 Conclusions and Future Work

This paper presented the Hybrid Cohorts (HyCo) algorithm. HyCo prioritizes software-mode transactions over hardware-mode transactions, and then employs HTM resources to accelerate the commit phase of software transactions. Hardware transactions do not write to global metadata, and in-flight software transactions do not even read global metadata. Performance is on par with the current state-of-the-art, RHNOrec.

HyCo and RHNOrec represent distinct points in the HyTM design space. Both are informed by Riegel et al. [16] and Dalessandro et al. [4]: they use value-based validation to avoid wasting HTM capacity on per-location metadata, and they avoid serialization when transactions cannot complete in HTM. HyCo adapts the "reduced hardware transaction" innovation from RHNOrec: rather than execute the postfix of transactions in HTM, HyCo commits (validation and writeback) via HTM. Another significant difference is how the algorithms handle the worst case: in HyCo, many transactions can run in the slowest mode simultaneously, but they must validate; in RHNOrec, one transaction can use the slowest mode at a time, but fewer transactions require the slowest mode. Additionally, when a slow transaction runs, HyCo can still allow hardware transactions to commit, whereas RHNOrec does not. HyCo also appears to offer more opportunity for contention management, though we did not evaluate that possibility in this paper.

Which algorithm is "better" is likely to depend on how HTM evolves. Clearly, both systems can benefit from increased HTM capacity, which will allow more transactions to execute fully in hardware. As multi-chip HTM becomes more prevalent, HyCo may benefit more: RHNOrec inherits NOrec's requirement that some committing writers increment a global shared counter, which is known to scale poorly on multi-chip machines; HyCo has no such bottleneck. Another open question is how nontransactional loads might improve the algorithms: HyCo could employ nontransactional reads to allow hardware transactions to spin, rather than abort, when software transactions block their commit. No such opportunity is apparent for RHNOrec.

Unfortunately, we cannot draw stronger conclusions without better TM benchmarks. On microbenchmarks, and on STAMP, RHNOrec and HyCo perform similarly despite significant differences in design (especially in the fall-back path). On the only realistic workload available, Memcached, RHNOrec performs better at high core counts, where symmetric multithreading reduces HTM capacity. We are hopeful that as more programmers use TM, there will be new opportunities to contrast HyTM implementations and draw conclusions about what costs are most important to avoid.

Acknowledgments

We thank our reviewers, and the TRANSACT community, for their excellent advice. 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. H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, Denver, CO, Aug. 2006.
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. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.

10. Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.
20. 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.
21. 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.
22. C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, Mar. 2007.
23. 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.
24. R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.
25. R. Yoo and H.-H. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.