# Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory

Alexander Matveev

MIT

amatveev@csail.mit.edu

Nir Shavit

MIT

shanir@csail.mit.edu

## Abstract

Because of hardware TM limitations, software fallbacks are the only way to make TM algorithms guarantee progress. Nevertheless, all known software fallbacks to date, from simple locks to sophisticated versions of the NOrec Hybrid TM algorithm, have either limited scalability or weakened semantics.

We propose a novel reduced-hardware (RH) version of the NOrec HyTM algorithm. Instead of an all-software slow path, in our RH NOrec the slow-path is a "mix" of hardware and software: one short hardware transaction executes a maximal amount of initial reads in the hardware, and the second executes all of the writes. This novel combination of the RH approach and the NOrec algorithm delivers the first Hybrid TM that scales while fully preserving the hardware's original semantics of opacity and privatization.

Our GCC implementation of RH NOrec is promising in that it shows improved performance relative to all prior methods, at the concurrency levels we could test today.

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

***General Terms*** Algorithms, Design

***Keywords*** Transactional Memory

## 1. Introduction

Transactional memory (TM) is an alternative to using locks for shared-memory synchronization. In TM, the programmer marks code regions as transactions, and the TMs internal implementation ensures a consistent execution of those regions. Transactions provide *external consistency*, that is,

a transaction can successfully commit only when it can be totally ordered relative to other (possibly concurrent) transactions without violating their real time order. This is the safety property called serializability. Transactions also provide *internal consistency*: a transaction cannot see inconsistent views of the system state, that is, ones that result from intermediate modifications of another transaction, since this may cause the current transaction to fail unexpectedly or enter an infinite loop. Internal consistency is captured by the safety properties *opacity* [13] and *privatization* [17]. Opacity, which we will refer to many times in this paper, is the property that every transaction sees a snapshot of all the variables it accesses at any time. Given internal and external consistency, the programmer can think of a transaction as one "atomic" (i.e. indivisible) sequence of operations.

Hardware transactional memory (HTM) is currently supported by the mainstream Intel Haswell [14, 21] and the IBM Power8 [3] processors. Intel and IBM provide *best-effort* HTM implementations that are much faster than software transactions, providing serializability, opacity, and privatization, but not providing any progress guarantee. CPU cache capacity limitations, interrupts, page faults, system calls or unsupported instructions, may all cause a hardware transaction to repeatedly fail. To overcome this problem, a software fallback (a *slow-path*) is provided for the cases when the hardware transaction (the *fast-path*) fails. The ability to fall back to the slow-path allows one to provide some form of progress guarantee. The simplest of these software fallbacks is a lock, which provides progress and provides all of the hardware's internal and external consistency properties. However, the lock serializes all of the software and hardware attempts to complete the transaction, and therefore does not scale.

Thus, developers have been looking for alternative software slow-paths that will scale while providing all the consistency properties of the hardware. The approach of choice is hybrid transactional memory (HyTM), in which the slow path is a software transaction that can run concurrently with the hardware ones. Preserving internal consistency, in particular opacity and privatization, in the software slow-path, requires complex coordination protocols that impose non-

trivial penalties on the more common hardware fast-path. This is the limitation recent HyTM research is trying to overcome, and for which our new algorithm provides what we believe is the most complete and scalable solution to date.

## 1.1 HyTM Background

The first HyTM [8, 15] algorithms supported concurrent execution of hardware and software transactions by instrumenting the hardware transactions' shared reads and writes to check for changes in the software path's metadata. This approach, which is the basis of all the generally applicable HyTM proposals, imposes severe overheads on the hardware transaction.

One way to overcome these overheads is through a PhasedTM [16] approach: transactions are executed in phases, each of which is either all hardware or all software. This approach performs well when all hardware transactions are successful, but has poor performance if even a single transaction needs to be executed in software because it must switch all transactions to a slower all-software mode of execution. This is a good approach for some workloads, but in others it is not clear how to overcome frequent switches between phases.

The leading HyTM algorithm to-date is the Hybrid NOrec HyTM of Dalessandro et al. [7], a hybrid version of the Norec STM of Dalessandro, Spear, and Scott [6]. In this algorithm, the commits of write transactions are executed sequentially, and a shared global clock is used to notify concurrent transactions about the updates to memory. The write commits trigger the necessary re-validations and aborts of the concurrently executing transactions. The great benefit the NOrec HyTM scheme over prior HyTM proposals is that no metadata per memory location is required and instrumentation costs are reduced significantly. However, it suffers from limited scalability because it uses a shared clock to ensure serializability and opacity. All hardware transactions must read the clock when they start, which adds this clock location to the HTMs tracking set and causes them to frequently abort; essentially every time the clock gets updated by a software slow-path transaction that writes.

Riegel et al. [19] were aware of this drawback of the Hybrid NOrec algorithm and suggested a way to reduce these overheads by using non-speculative operations inside the hardware transactions. These operations are supported by AMD's proposed ASF transactional hardware [5] but are not supported in the best-effort HTMs that Intel and IBM are bringing to the marketplace [1] .

The original authors of the NOrec HyTM, and the recent work by Calciu et al. [4], both state that it is a challenging task to improve the scalability of the NOrec HyTM on existing architectures unless one gives up opacity. They thus propose to sacrifice opacity by implementing efficient but unsafe HyTM protocols that allow non-opaque hardware transactions, ones that may read inconsistent system states. The idea is to use the built-in HTM sandboxing mechanisms to protect non-opaque hardware transactions from runtime errors. If the inconsistent hardware transaction performs a "bad" action that causes it to abort, the hardware together with the runtime can detect and recover from the error. The problem with HTM sandboxing is that it does not protect from *all* possible errors. The authors identify a set of possible scenarios in which transactions might still incorrectly commit in an inconsistent state, and suggest how they can be handled by extending compiler and runtime support.

However, recent work by Dice et al. [10] identifies many additional scenarios of bad executions that escape the HTM sandboxing mechanism. These additional cases rise concerns about the HTM sandboxing approach since on existing hardware they would require a much more complex compiler and runtime support to be handled, overshadowing the advantages of using HTM sandboxing in the first place. Figuring out what hardware modifications and new algorithms would make sandboxing a viable approach is an interesting topic for future research.

We henceforth focus on the performance of HyTM algorithms that can provide the same consistency properties as pure hardware transactions on todays new commodity multicore architectures.

## 1.2 The RH Approach

In a recent paper, we proposed the reduced hardware approach to HyTM design [18] as a way of minimizing the overhead of the hardware fast-path without restricting concurrency between hardware and software transactions. The idea was that instead of the all-software slow-path used in HyTMs, part of the slow-path will be executed using a reduced size hardware transaction, making the slow path a software-hardware *mixed slow-path*. To show the benefit of this approach [18], we proposed a reduced hardware version of the classic TL2 STM of Dice, Shalev and Shavit [9]. The RH-TL2 algorithm eliminated the instrumentation overheads of the shared reads from the hardware fast-path, and therefore provided a more efficient HyTM fast-path which improved on all prior TL2-style HyTM algorithms. However, there still remain several fundamental problems with the RH-TL2 algorithm which this paper sets out to solve:

- **The RH-TL2 fast-path required instrumentation:** There was no need to instrument the hardware fast-path reads, but there was a need to instrument its writes and update the software metadata for every write location of the fast-path before it performs the HTM commit. Though reads are more frequent than writes, this was still far from the performance of having a fully pure uninstrumented hardware transaction.

---

[1] IBM Power8 supports a suspend-resume feature that allows to execute non-speculative operations in hardware transactions. The problem is that suspend-resume is intended for debugging and is expensive to use. It cannot be used frequently, for every shared read and write.

- **The RH-TL2 mixed slow-path had high chances of failure:** The mixed slow-path small hardware transaction required a verification phase of the read locations before their write-back. As a result, the chances of the small hardware transaction failing were still high since it included both the read and write locations in HTM.

- **The RH-TL2 algorithm did not provide privatization:** Privatization [17] is a feature provided by hardware transactions, guaranteeing that the modifications of ongoing transactions are not visible to correctly synchronized non-transactional operations. This would be a serious limitation if the algorithm were to be deployed in a commercial setting.

### 1.3 Our New Reduced Hardware NOrec Algorithm

In this paper, we show how to apply the RH approach of [18] to the NOrec HyTM algorithm, resulting in a new HyTM algorithm that overcomes all the drawbacks of RH TL2 mentioned above, and more importantly, all the scalability limitations of the NOrec HyTM. It provides the first scalable HyTM that has complete internal and external consistency; the same as a pure hardware TM.

Our RH NOrec algorithm, which we describe in detail in the next section, uses two short hardware transactions in the mixed slow-path, to eliminate NOrec HyTM scalability bottlenecks. Both of these bottlenecks are caused by reading of the shared clock too early. In one case too early in the hardware fast-path, and in the other too early in the software slow-path. We execute two hardware transactions, in the mixed slow path, to overcome these problems. One hardware transaction encapsulates all the slow path writes at commit point and executes them all together. This change to the mixed software slow-path enables the fast-path hardware transaction to read the shared NOrec clock at the end of the transaction, just before committing, instead of reading it when it starts, avoiding the frequent false-aborts of the original Hybrid NOrec. The other hardware transaction, in the mixed slow path, executes the largest possible prefix of slow-path reads in a hardware transaction. In other words, it starts the mixed slow-path with a hardware transaction, and as long as it does not encounter a write it keeps on executing in hardware. This hardware prefix allows us to defer the reading of the global clock *in the mixed slow-path itself* to the end of the sequence of reads, which as we will explain, significantly reduces the chances it will have to restart. We believe it is algorithmically surprising that with the hardware support in the mixed path, moving the clock-reads back preserves opacity, enables a pure uninstrumented fast-path, and allows full privatization as in the original Hybrid NOrec algorithm. Finally, we note that if the mixed slow-path fails to commit, the algorithm reverts to the original Hybrid NOrec.

We integrated our new RH NOrec HyTM into the GCC compiler, and performed empirical testing on 16-way Intel Haswell processor. Though one must be cautious given the limited concurrency we could test, our results are rather encouraging: we show that RH NOrec is able to reduce the number of HTM conflicts from 8-20 fold compared to NOrec HyTM for the RBTree microbenchmark, and 2-5 fold for the STAMP applications. Though the reduction in HTM conflicts does not always lead to better throughput results (sometimes the dominant factor is the level of HTM capacity aborts, and sometimes the HTM conflicts are already very low, so reducing them more does not make too much of a difference), in benchmarks that suffer from high HTM conflict rates, RH-NOrec shows a significant improvement over the best previous schemes.

## 2. Reduced Hardware NOrec

We begin by explaining the Hybrid NOrec algorithm on which RH NOrec is based.

### 2.1 Hybrid NOrec Overview

The original NOrec STM, that is, the all software one, coordinates transactions based on a global clock that is updated every time a write transaction commits, and also serves as a global lock. It keeps a read set and a write set, that is, logs of all the locations read and written, and their values. A transaction reads the global clock value when it starts, and stores it in a local clock variable. To perform a read, the transaction (1) reads the location, (2) logs its value to a read-set log, and (3) performs a consistency check that ensures opacity: it checks that the local clock value is the same as the global clock value. If the clocks differ, then it means that some writer has committed and maybe even overwrote the current transaction. Thus, the current transaction must perform a full read-set revalidation in order to rule out a possible overwrite. To execute a write, the NOrec algorithm can work in two possible ways: (1) write to a local write-set, deferring the writes to the commit, or (2) write directly to the memory. In both cases, to ensure opacity, the NOrec algorithm locks the global clock before making the actual memory writes (when they are encountered or eventually when commiting), and releases it after the actual writes are done. While the global clock is locked, other transactions are delayed, spinning until it is released.

The original Hybrid NOrec has each transaction start to execute in hardware, and if it repeatedly fails to commit, it falls back to executing the NOrec STM in software. Of course, one must coordinate among the hardware and software to guarantee serializability and opacity. Figure 1 presents a problematic scenario in a hybrid TM execution that one must solve in order to provide opacity. The scenario is a concurrent execution of one hardware fast-path transaction and one software slow-path transaction. We omit many of the hybrid protocol's details and focus only on direct memory reads and writes. In the scenario, the slow-path updates shared variables X and Y and the fast-path reads X and Y. Now, if the slow-path updates X, and before it updates
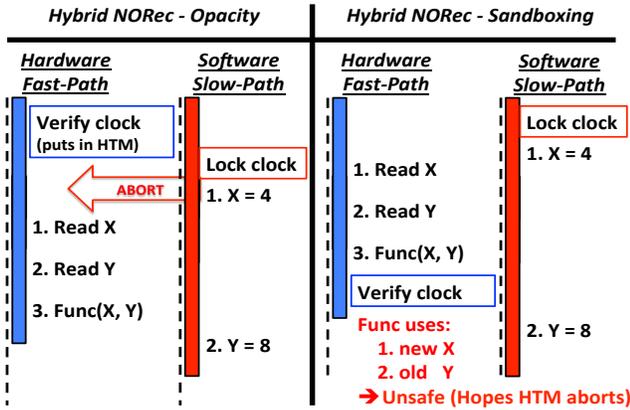
**Figure 1.** The problematic scenario that a hybrid protocol must handle to provide opacity.



**Figure 2.** The RH NOrec provides opacity and concurrency between the fast-path and the slow-path by encapsulating the writes in the HTM.

Y, the fast-path reads X and Y, then the fast-path has now a view of a new X and an old Y relative to the slow-path transaction, which is an inconsistent memory view. To avoid this problem, in Hybrid NOrec, when a hardware fast-path transaction starts, it reads the global clock in hardware and verifies it is not locked. Thus, the hardware tracks the clock location from the very start of its execution and will abort whenever any software slow-path that writes to memory locks the global clock to perform its actual memory writes. This creates many false-aborts: aborts by software threads writing to completely unrelated locations.

The way to overcome this problem is to move the verification of the global clock from the start of the hardware fast-path to its end, eliminating the excessive false-aborts. However, as mentioned earlier, doing so in the Hybrid NOrec algorithm requires perfect HTM sandboxing to protect against the inconsistent executions that may corrupt memory and crash the program. Such sandboxing unfortunately does not exist.

## 2.2 RH NOrec Overview

We now explain our new RH NOrec algorithm. In this algorithm we will move the access and verification of the global clock back in both the hardware and software transactions, while maintaining consistency. We will do so by adding two small hardware transactions to the software slow-path of the Hybrid NOrec. The new slow-path executes using both hardware and software, so we call it the *mixed slow-path*.

The first hardware transaction is *HTM postfix*. It encapsulates all the mixed slow path writes and executes them in one hardware transaction. Executing the writes as an atomic hardware transaction guarantees the fast-path hardware transaction cannot see partial writes of the mixed slow-path, and this enables the fast-path to read the shared NOrec clock at the end of the transaction, just before committing, instead of reading it when it starts. Figure 2 depicts this
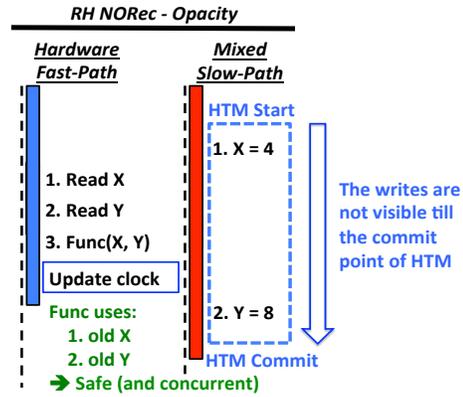
change and shows how it solves the problematic scenario of the previous Figure 1. Recall, the issue is that the the slow-path may update X, and before it updates Y, the fast-path may read X and Y, where X is new and Y is old relative to the slow-path execution. In the new algorithm, the mixed slow-path writes X and Y together in the HTM postfix transaction, so any concurrent hardware fast-path transaction will see a consistent view.

The second hardware transaction is *HTM prefix*. It tries to execute the largest possible prefix of mixed slow-path reads in a single hardware transaction. In other words, it starts the mixed slow-path with a hardware transaction, and as long as it does not encounter a write it keeps on executing in hardware. The HTM prefix uses the HTM conflict-detection mechanism to detect overwrites of the slow-path reads. This replaces the reading of the shared NOrec clock on the mixed slow-path and validating that it is still the same for each read. In this way, the RH NOrec algorithm defers the read of the shared NOrec clock to the commit point of the HTM prefix, significantly reducing the execution window during which it is possible to abort the mixed slow-path. To make this approach efficient and reduce the false-abort window as much as possible, the length of the HTM prefix adjusts dynamically during the runtime, based on the HTM abort rate feedback, so that the largest amount of mixed slow-path reads can execute in the hardware.

Figure 3 shows an execution example of the RH NOrec mixed slow-path with the two small hardware transactions. The figure presents two fast-path hardware transactions and a one mixed slow-path transaction that are able to execute concurrently (assume no real conflicts) and preserve opacity. The mixed slow-path starts by executing its first reads in the HTM prefix hardware transaction. The first fast-path transaction has this advantage: it executes concurrently with the mixed slow-path HTM prefix, and performs a commit and update of the global clock without forcing the mixed
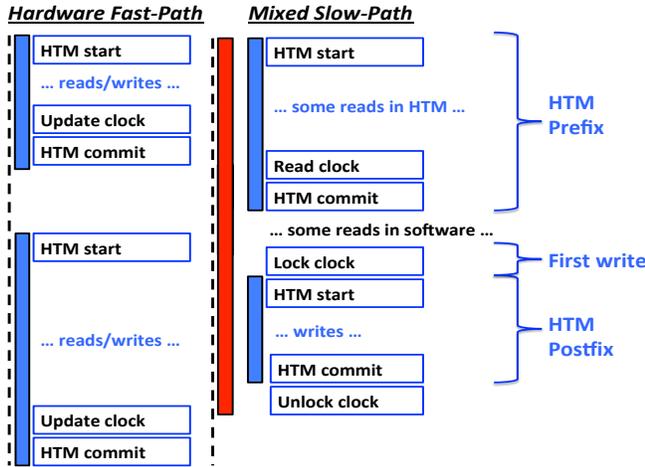
**Figure 3.** Execution example of RH NOrec that reveals the concurrency advantages of using small hardware transactions in the mixed slow-path: the HTM prefix, for the first reads, and the HTM postfix, for all of the writes. The figure presents two fast-path hardware transactions and a one mixed slow-path transaction, that are able to execute concurrently and preserve opacity.

slow-path to restart. When the HTM prefix commits, it reads the global clock to a local clock variable and proceeds to execute the rest of the reads in the software. Upon the first mixed slow-path write, it starts the HTM postfix hardware transaction, and this executes concurrently with the second fast-path transaction. When the HTM postfix commits, it updates the global clock without aborting the second fast-path, and later the fast-path commits successfully.

Finally, when some mixed slow-path small hardware transaction repeatedly fails to commit, the RH NOrec mixed slow-path reverts back to the Hybrid NOrec full software slow-path counterpart. For the HTM prefix, it reads the shared NOrec clock on the slow-path start and validates it is still the same for each read. For the HTM postfix, it aborts all current hardware transactions and executes the writes in the software.

## 2.3 RH NOrec Algorithm

For clarity, in this section we present a mixed slow-path that uses only the HTM postfix transaction, that encapsulates the writes. In the next section, we add the necessary modifications to support the second, HTM prefix short hardware transaction.

Algorithm 1 and Algorithm 2 show the pseudo-code for the hardware fast-path and the mixed slow-path.

The hybrid protocol coordinates fast-paths and slow-paths by using three global variables:

1. *global_clock:* Implements the conflict-detection mechanism. Every hardware fast-path increments the clock be-

---

**Algorithm 1** RH NOrec hardware fast-path transaction

```
 1: function FAST_PATH_START(ctx)
 2:     label: retry
 3:     HTM_Start()
 4:     if HTM failed then
                ▷ Handle HTM abort
 5:         ... retry policy ...
 6:         if no retry then
 7:             fallback to mixed slow-path
 8:         end if
 9:         goto retry
10:     end if
            ▷ HTM started; subscribe to HTM lock
11:     if global_htm_lock ≠ 0 then
12:         HTM_Abort()
13:     end if
14: end function
15:
16: function FAST_PATH_READ(ctx, addr)
            ▷ no instrumentation - simple load
17:     return load(addr)
18: end function
19:
20: function FAST_PATH_WRITE(ctx, addr, value)
            ▷ no instrumentation - simple store
21:     store(addr, value)
22: end function
23:
24: function FAST_PATH_COMMIT(ctx)
25:     if HTM is read only then
                ▷ Detected by compiler static analysis
26:         HTM_Commit()
27:         return
28:     end if
29:     if num_of_fallbacks > 0 then
                ▷ Notify mixed slow-paths about the update
30:         if is_locked(global_clock) then
31:             HTM_Abort()
32:         end if
33:         global_clock ← global_clock + 1
34:     end if
35:     HTM_Commit()
36: end function
```

fore it commits to notify mixed slow-paths about the memory update, and every mixed slow-path does the same to notify other mixed slow-paths.

2. *global_htm_lock:* Allows to abort all hardware fast-paths when the hardware component of the mixed slow-path fails. Every fast-path subscribes to the htm lock by reading it and verifying it is free on its start. This allows a failed mixed slow-path to abort all of them by setting the htm lock, and execute the mixed slow-path writes as is (in full software) while still ensuring opacity.

3. *num_of_fallbacks:* Represents the current number of mixed slow-paths. Hardware fast-paths use this counter

**Algorithm 2** RH NOrec mixed slow-path transaction that supports only HTM postfix.

```
 1: function MIXED_SLOW_PATH_START(ctx)
 2:     atomic_fetch_and_add(num_of_fallbacks, 1)
 3:     ctx.is_write_detected ← 0
 4:     ctx.tx_version ← global_clock
 5:     if is_locked(ctx.tx_version) then
 6:         restart(ctx)
 7:     end if
 8: end function
 9:
10: function MIXED_SLOW_PATH_READ(ctx, addr)
11:     cur_value ← load(addr)
12:     if ctx.tx_version ≠ global_clock then
13:         restart(ctx)
14:     end if
15:     return cur_value
16: end function
17:
18: function MIXED_SLOW_PATH_WRITE(ctx, addr, value)
19:     if ¬ctx.is_write_detected then
20:         handle_first_write(ctx)
21:     end if
22:     store(addr, value)
23: end function
24:
25: function HANDLE_FIRST_WRITE(ctx)
        ▷ Lock the clock and start HTM. If the HTM fails, then set
          the HTM lock to abort all hardware fast-paths.
26:     ctx.is_write_detected ← 1
27:     acquire_clock_lock(ctx)
28:     if ¬ start_rh_htm_postfix(ctx) then
29:         global_htm_lock ← 1
30:     end if
31: end function
32:
33: function START_RH_HTM_POSTFIX(ctx)
34:     label: retry
35:     HTM_Start()
36:     if HTM failed then
37:         ... retry policy ...

38:         if no retry then
39:             return false
40:         end if
41:         goto retry
42:     end if
43:     ctx.is_rh_active ← true
44:     return true
45: end function
46:
47: function ACQUIRE_CLOCK_LOCK(ctx)
48:     new ← set_lock_bit(ctx.tx_version)
49:     expected ← ctx.tx_version
50:     is_locked ← atomic_CAS(global_clock, expected, new)
51:     if ¬is_locked then
52:         restart(ctx)
53:     end if
54:     ctx.tx_version ← new
55:     return true
56: end function
57:
58: function MIXED_SLOW_PATH_COMMIT(ctx)
        ▷ If read-only, then do nothing
59:     if ¬ctx.is_write_detected then
60:         atomic_fetch_and_sub(num_of_fallbacks, 1)
61:         return
62:     end if
        ▷ If RH is on, then commit it
63:     if ctx.is_rh_active then
64:         HTM_Commit()
65:         ctx.is_rh_active ← false
66:     end if
        ▷ If HTM lock is taken, then release it
67:     if global_htm_lock ≠ 0 then
68:         global_htm_lock ← 0
69:     end if
        ▷ Update the clock: clear the lock bit and increment
70:     global_clock ← clear_lock_bit(global_clock) + 1
71:     atomic_fetch_and_sub(num_of_fallbacks, 1)
72: end function
```

to avoid updating the global clock when there is no slow-path executing.

Algorithm 1 shows the hardware fast-path implementation. On start, it initiates a hardware transaction (line 3) and subscribes to the global htm lock (line 11). If the HTM fails, the abort handling code (lines 4 - 9) executes and decides when to perform the fallback. During the HTM execution, reads and writes execute without any instrumentation (line 17 and 21), making them as efficient as pure HTM. Upon commit, a read-only HTM commits immediately. Otherwise, the HTM checks if there are mixed slow-path fallbacks, and if there are, then it increments the global clock to notify the fallbacks about the writer. It then commits the hardware transaction (lines 29-35). The detection of read-

only fast-paths is currently based on the GCC compiler static analysis that provides this information to the GCC TM library.

Algorithm 2 shows the mixed slow-path implementation that uses only the HTM postfix short hardware transaction. Upon start, it increments the fallback counter, sets the writer detection flag to 0, and reads the global clock to a local variable called *tx_version* (lines 2 - 7). During the mixed slow-path execution, the reads and writes are performed directly to/from the memory. To execute a read, it loads the memory location, and then verifies that global clock version is still the same as the local version. If there is a change, then this means that some write transaction committed, possibly overwriting the current transaction, and therefore the current transaction restarts (lines 11 - 15). To execute a write, it first

**Algorithm 3** RH NOrec mixed slow-path transaction that supports both HTM prefix and HTM postfix.

```
 1: function MIXED_SLOW_PATH_START(ctx)
 2:     if start_rh_htm_prefix(ctx) then
 3:         return
 4:     else
 5:         ... original code (of Algorithm 2) ...
 6:     end if
 7: end function
 8:
 9: function START_RH_HTM_PREFIX(ctx)
10:     label: retry
11:     HTM_Start()
12:     if HTM failed then
13:         ... retry policy and prefix length adjustment ...
14:         if no retry then
15:             return false
16:         end if
17:         goto retry
18:     end if
19:     ctx.is_rh_prefix_active ← true
            ▷ HTM prefix started; subscribe to HTM lock
20:     if global_htm_lock ≠ 0 then
21:         HTM_Abort()
22:     end if
            ▷ Set the prefix expected length
23:     ctx.max_reads ← ctx.expected_length
24:     ctx.prefix_reads ← 0
25:     return true
26: end function
27:
28: function MIXED_SLOW_PATH_READ(ctx, addr)
29:     if ctx.is_rh_prefix_active then
30:         ctx.prefix_reads ← ctx.prefix_reads + 1
31:         if ctx.prefix_reads < ctx.max_reads then
32:             return load(addr)
33:         else
34:             commit_rh_htm_prefix(ctx)
35:         end if
36:     end if
37:     ... original code (of Algorithm 2) ...
38: end function
39:
40: function MIXED_SLOW_PATH_WRITE(ctx, addr, value)
41:     if ctx.is_rh_prefix_active then
42:         commit_rh_htm_prefix(ctx)
43:     end if
44:     ... original code (of Algorithm 2) ...
45: end function
46:
47: function COMMIT_RH_HTM_PREFIX(ctx)
48:     num_of_fallbacks ← num_of_fallbacks + 1
49:     ctx.is_write_detected ← 0
50:     ctx.tx_version ← global_clock
51:     if is_locked(ctx.tx_version) then
52:         HTM_Abort()
53:     end if
54:     HTM_Commit()
55:     ctx.is_rh_prefix_active ← false
56: end function
57:
58: function MIXED_SLOW_PATH_COMMIT(ctx)
59:     if ctx.is_rh_prefix_active then
60:         HTM_Commit()
61:         return
62:     end if
63:     ... original code (of Algorithm 2) ...
64: end function
```

checks if this is the first write (line 19). If so, it sets the writer detected flag to 1, and then locks the global clock by *atomically* (1) setting its lock-bit to 1 and (2) verifying that the global clock is still the same as the local clock (line 26 and lines 48 - 55). Then, it initiates a hardware transaction that is going to perform the rest of the transaction, which includes all of its writes. If this hardware transaction fails to commit, then it sets the *global htm lock*, aborting all current hardware fast-paths, and executes the rest of the transaction as is, with the software NORec algorithm providing full opacity (lines 28 - 30). Upon commit, it checks if the transaction is read-only. If so, then it decrements the fallback counter and it is done (lines 59 - 62). Otherwise, it commits the writes' hardware transaction if it is active, and then releases all locks being held and decrements the fallback counter (lines 63 - 71).

## 2.4 Reducing Mixed Slow-Path Restarts

In the mixed slow-path using an eager approach (that writes in place as opposed to maintaing a write set that is written upon commit) is lightweight and efficient: It has no read-set or write-set logging and performs direct memory reads and writes. However, eliminating the write-set logs forces a mixed slow-path restart every time the global clock is updated by a concurrent write transaction. In this section, we significantly reduce the amount of these restarts by extending the mixed slow-path to use the second, HTM prefix, short hardware transaction.

A mixed slow-path may perform a restart in any point between the start of the transaction and its first shared memory write. This is because the mixed slow-path reads the global clock upon starting, and only locks it on the first write. So the initial reads, the ones before the first write, execute with a consistency check that may result in a restart. The idea of the HTM prefix short hardware transaction is to execute maximal amount of these reads in the HTM, which effectively eliminates the need to read the global clock on the start and allows to defer it to the commit point of the HTM prefix. To make this effective, the length of the HTM prefix adjusts dynamically based on the HTM abort feedback: at first, the

algorithm tries to execute a long prefix, and if this fails to commit in the hardware, then it reduces the length until it finds the right length that will commit with high probability (the adjustment algorithm is similar to that used in [2]).

Algorithm 3 shows modified mixed slow-path that supports the HTM prefix hardware transaction. Upon start, it initiates the prefix hardware transaction (lines 2 - 3). If it fails, then it executes the original start code (of Algorithm 2), and otherwise it skips the original code and proceeds in the HTM. The prefix initiation procedure is implemented in the *start_rh_htm_prefix* function. It starts a hardware transaction, sets the *is_rh_prefix_active* flag to 1, subscribes to the global htm lock to ensure opacity (in a similar way to the fast-paths), and initializes the prefix length parameters (lines 10 - 25). The algorithm adjusts the prefix maximum length (line 13), according to the HTM aborts feedback, and enforces it by counting the number of reads executed (lines 29 - 36). When the maximum number of reads is reached, or a first write is encountered (lines 41 - 43), the HTM prefix executes the commit procedure (lines 48 - 55). This essentially performs the original start code including the initialization of the local version variable. Finally, if none of the threshold conditions are met, and the HTM prefix reaches the mixed slow-path commit, then it simply commits the transaction and returns (lines 59 - 62).

## 3. Performance Evaluation

We integrated the new RH NOrec algorithm and the original Hybrid NOrec algorithm into the GCC 4.8 compiler. GCC 4.8 provides language-level support for transactional memory (GCC TM) based on the draft specification for C++ of Adl-Tabatabai et. al [1].

The GCC TM compiler generates two execution paths for each GCC transaction: (1) a pure uninstrumented fast-path and (2) a fully instrumented slow-path. In the second path, every shared read and write is replaced with a dynamic function call to the *libitm* runtime library that provides an implementation for the various TM algorithms. In addition, the compiler instrumentation provides the libitm library with runtime hints about the code read-write, write-write and write-read dependencies, and allows the libitm library to identify static read-only transactions. In our implementations, we use all these hints to improve the performance of the Hybrid TMs in the same way as the original libitm library uses the hints to improve the performance of the STMs.

### 3.1 TM Algorithms

We tested the performance of the following GCC TM algorithms:

**Lock Elision:** Transactions execute in pure hardware, using the RTM Haswell mechanism [14], and if a transaction fails to commit in hardware, then it acquires the global lock, which aborts all hardware transactions and serializes the execution to ensure progress.

**NOrec:** All transactions execute as software NOrec [6] transactions with eager encounter-time writes. Upon a first write, a transaction locks the global clock and performs the subsequent writes directly to the memory. In order to perform a read, a transaction reads from the memory and verifies consistency. In this version of NOrec, there are no read-set or write-set logging procedures, so a transaction must restart when a write transaction commits. We also implemented the lazy design of NOrec that does require read-set and write-set logging, but we found that for the low concurrency in our benchmarks, the eager NOrec design delivers better performance.

**TL2:** All transactions execute in software following the TL2 [9] algorithm with eager encounter-time writes. To write, a transaction locks the metadata of the write location, and performs a direct memory write. To read, it logs the location in the read-set, reads it from the memory, and verifies the associated metadata. Upon commit, it revalidates the read-set, and if successful, releases the locks while advancing the global clock. The TL2 STM imposes high constant overheads, but provides better scalability than the NOrec STM because it uses per location metadata for conflict-detection.

**HY-NOrec:** The NOrec HyTM algorithm of Dalessandro et. al [7]. The coordination between the hardware fast-path and the software slow-path tries to avoid as many false-aborts as possible. To achieve this, the coordination protocol uses 3 global variables: (1) global htm lock, (2) global NOrec clock, and (3) fallback count. A hardware fast-path reads the global htm lock upon start and verifies it is free. In the fast-path commit, it increments the global clock only if the fallback count is not 0. The software slow-path executes the eager encounter-time NOrec STM, which aborts all of the fast-paths only when the first write is encountered, by setting the global htm lock. As explained above in the case of NOrec, we found this eager HyTM design outperforms the lazy HyTM design for the low concurrency levels available in our benchmarks.

**RH-NOrec:** This is our new hybrid TM as described in Section 2.

### 3.2 Execution Environment

In our benchmarks, we use the 16-way Intel Core i7-5960X Haswell processor: It has 8-cores and provides support for HyperThreading of two hardware threads per core. The new Haswell chip RTM system provides support for hardware transactions that can fit into the capacity of the L1 cache. As an optimization, the hardware internally uses bloom filters for the read-sets and write-sets, so it can also provide larger *read-only* hardware transactions that can fit into the capacity of the L2 cache. It is important to notice that the HyperThreading reduces the L1 cache capacity for HTM by

a factor of 2, since it executes two hardware threads on the same core (same L1 cache). As a result, in many benchmarks there are significant penalties above the limit of 8 threads, where the HyperThreading executes and generates an increased amount of HTM capacity aborts.

The operating system we use is a Debian OS 3.14 x86_64 with GCC 4.8. All of the benchmarks are compiled with an O3 optimization level and inlining enabled. We found that the malloc/free library provided with the system does not scale and imposes high overheads and many false aborts on the HTM mechanism. We thus switched to the scalable *tc-malloc* [12] memory allocator that maintains per thread memory pools and provides good performance inside the HTM transactions. The GCC libitm library was modified to include the new Hybrid TMs and compiled with the default parameters.

### 3.3 Retry Policy

Both Hybrid NOrec and RH Norec use the same static retry policy, that we found to perform well for most of the benchmarks. In the next paragraphs, we describe the retry algorithm for the fast-path and the slow-path. Note that for Lock Elision only the fast-path applies.

*Fast-path:* When a hardware fast-path transaction aborts, the protocol checks the HTM abort flags to see the reason for the abort. If the HTM flags indicate that retrying in the hardware may help (the flag NO_RETRY is false), then the transaction restarts in the hardware. Otherwise it falls back to the software slow-path. The maximum number of hardware restarts we allow is 10, after which the transaction falls back to the software. The effect of this policy is that capacity aborts immediately go to the software, while conflict aborts retry many times in the hardware, giving the hardware a chance to complete as many transactions as possible.

*Slow-path:* The slow-path of the Hybrid TMs may starve when there are many frequent fast-path write transaction commits: each such commit updates the clock, and this forces a slow-path restart. Note that this starvation scenario is less likely for the RH NOrec, but still may happen when the HTM prefix fails. Therefore, to avoid this and ensure progress, we use a special *serial lock* to serialize the execution of the problematic slow-paths: each slow-path that restarts too often acquires this special lock when it starts, and releases it when it is done. In turn, the HTM fast-paths, that need to increment the global clock (writers), check this lock and abort when it is taken. We set the slow-path restart limit to be 10, since we found this limit to provide the best overall results for most of the benchmarks.

A recent paper [11] presents a dynamic-adaptive retry policy for lock elision schemes. Implementing a dynamic retry policy for Hybrid TMs is an interesting and important topic for future work.

### 3.4 Mixed Slow-Path HTM Details

The RH NOrec mixed slow-path uses two small hardware transactions: one is the *HTM prefix* that executes the first part of the mixed slow-path and reduces the software restarts, and another is the *HTM postfix* that performs the memory writes and allows the hardware fast-paths to execute concurrently. We found out that the best retry policy for these small HTMs is to try each one only once before using the full software counterpart. This is because many retries increase the execution time of the mixed slow-path, and generate more cache pressure on the HTM fast-paths.

### 3.5 Red-Black Tree Microbenchmark

The red-black tree we use is derived from the *java.util.TreeMap* implementation found in the Java 6.0 JDK. The red-black tree benchmark exposes a key-value pair interface of *put*, *delete*, and *get* operations, and allows to control the (1) tree size and the (2) mutation ratio (the fraction of write transactions). Each run executes for 10 seconds, and reports the total number of operations completed.

The left to right columns in Figure 4 present results for a 10,000 node red-black tree with 4%, 10% and 40% mutation ratios. The first row (from the top) shows the throughput of the various algorithms, while the next rows present an in-depth execution analysis of the Hybrid NOrec and the RH NOrec. The second row shows (1) the average number of HTM conflict and capacity aborts per completed operation. The third row shows the (2) average number of restarts per slow-path transaction. The fourth row shows the (3) slow-path ratio: the relative number of transactions that failed in the fast-path and made it fall back to the slow-path. The fifth row shows (4) the success ratios of the two small hardware transactions that are used by the RH NOrec mixed slow-path: the HTM prefix that reduces the slow-path restarts, and the HTM postfix that includes the memory writes to allow concurrent fast-path executions.

In Figure 4, we can see that the high instrumentation costs of STMs make them much slower than the HTM-based schemes. However, the performance of Lock Elision and Hybrid NOrec drop when the slow-path fallback ratio increases above 8 threads due to the HyperThreading effect. This performance drop becomes significant with more write transactions and the TL2 STM becomes faster than the Hybrid NOrec for the 40% mutation ratio. In contrast, the new RH NOrec software-hardware mixed fallback path is able to maintain the HTM advantages: it is 1.2, 2.4 and 5.0 times faster than Hybrid NOrec for the respective 4%, 10% and 40% mutation ratios. The execution analysis reveals a significant reduction in the HTM conflict aborts and slow-path restarts for the RH NOrec compared to Hybrid NOrec. This reduction generates a lower slow-path ratio, and is a result of the high success ratio of the HTM prefix and HTM postfix small hardware transactions that the RH NOrec uses in its mixed slow-path. Note that in the 40% mutation case, the
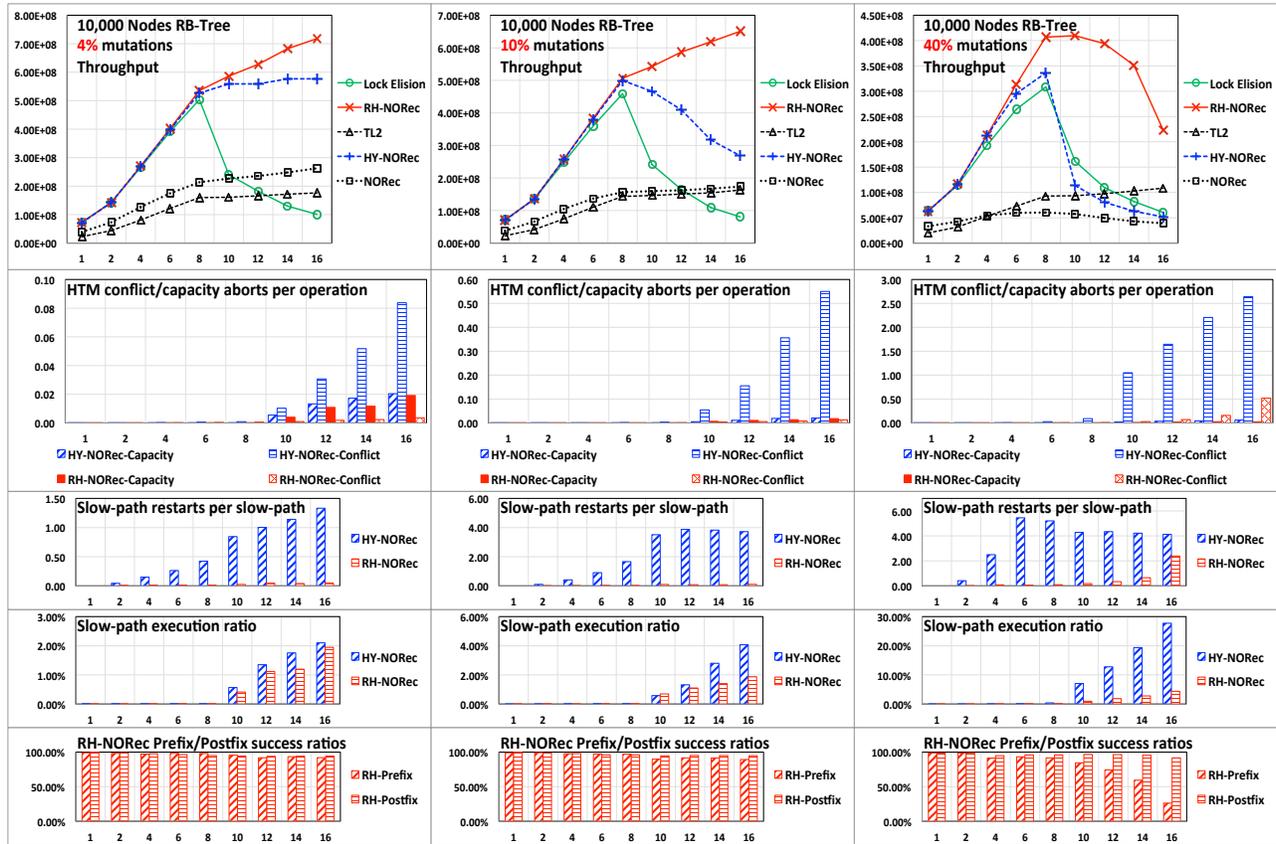
**Figure 4.** Red-Black tree benchmark. The first (top) row shows the throughput for 4%, 10% and 40% mutation ratios, and the next rows show the execution analysis for the Hybrid TMs.

HTM prefix has a lower success ratio and this aversly affects the RH NOrec scalability.

An interesting side-effect that can be seen in the RBTree analysis is a reduction in HTM capacity aborts for the RH NOrec. The reason for this side-effect is the "sensitivity" of this specific benchmark to the software fallbacks: it executes many "cache-friendly" read-only operations, and therefore the relative L1 cache pressure of the fallbacks is higher compared to benchmarks that execute many write operations. This is typical of the STAMP benchmarks for which this side-effect is small to neligable.
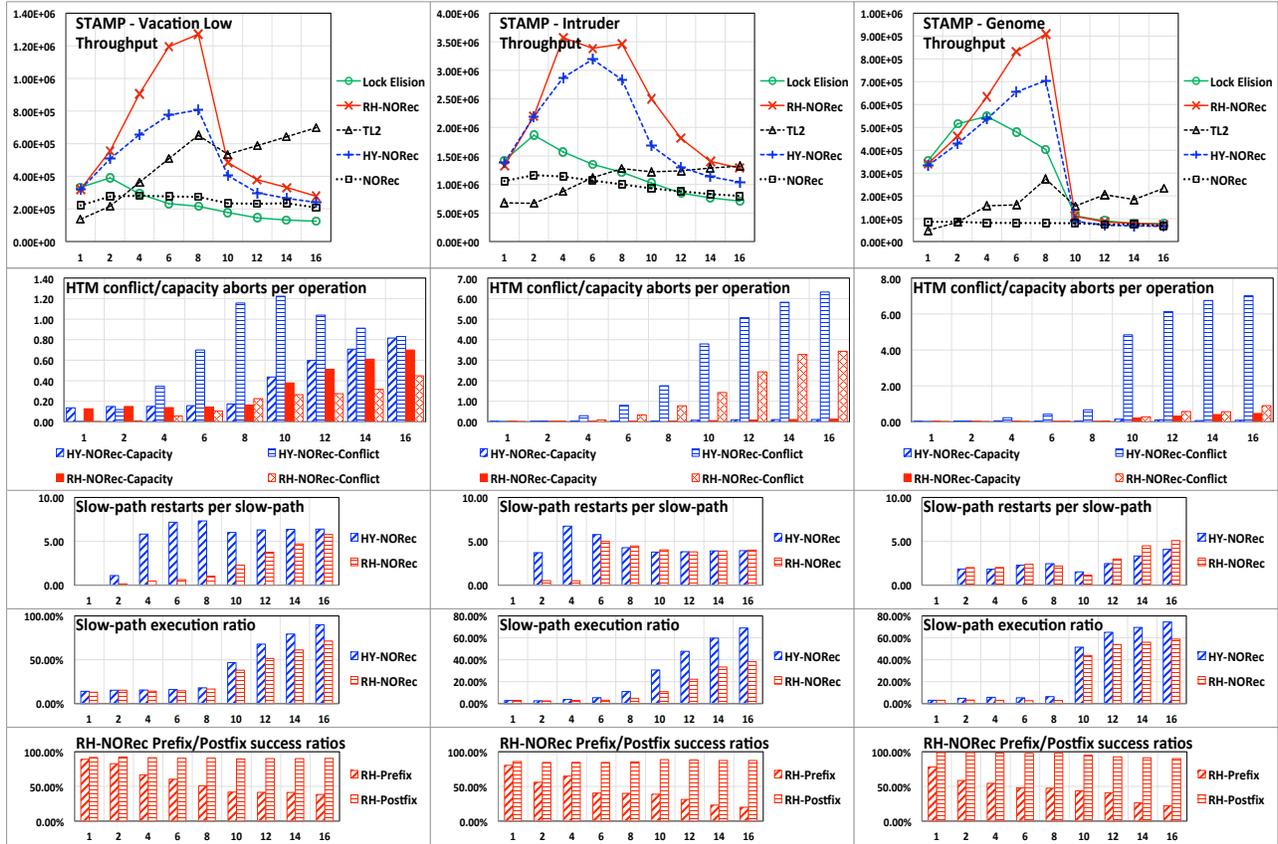
### 3.6 STAMP Applications

The STAMP suite we use is the most recent version [20] of Ruan et. al, and it includes various real-world applications with different access patterns and behaviors.

The left to right columns in Figure 5 present results for Vacation-Low, Intruder and Genome STAMP applications. The format of the presentation is exactly the same as in Figure 4.

Vacation-Low simulates online transaction processing, and it has a good potential to scale since it generates moderately long transactions with low contention. For this appli-

cation, the RH NOrec is twice as fast as the Hybrid NOrec for 8 threads, and the analysis shows that this is because the reduction in HTM conflicts. In this case, there is a significant reduction in the HTM conflicts and slow-path restarts just as for the RBTree, but there is no significant difference in the number of slow-path fallbacks. Still RH NOrec is 1.6 times faster than Hybrid NOrec for 8 threads since the overall number of fast-path and slow-path restarts is much lower. Above 8 threads, RH NOrec still reduces the HTM conflicts, slow-path restarts, and even the slow-path fallback ratio, but the HTM capacity aborts increase (due to HyperThreading) and dominate the performance: they increase the slow-path fallback ratio to the point where the RH NOrec reductions are not significant.

Intruder uses transactions to replace coarse-grained synchronization in a simulated network packet analyzer. This workload generates a large amount of short to moderate transactions with high contention, and the poor scalability of the TL2 STM is an indication of this internal behavior. Despite this, the Hybrid TMs significantly improve over the STMs, and RH NOrec is 1.2 and 1.5 times faster than the Hybrid NOrec for 8 and 10 threads. The RH NOrec behavior for 8-16 threads is the opposite of Vacation-Low: there is

**Figure 5.** Results for Vacation-Low, Intruder and Genome STAMP applications. The first (top) row shows the throughput, and the next rows show the execution analysis for the Hybrid TMs.

no difference in the number of slow-path restarts, but there is much lower slow-path fallback ratio. So in this case the advantage comes from the HTM postfix small hardware transaction that is able to reduce the HTM conflicts by a factor of 2, and not from the HTM prefix that mostly fails due to high contention. Note that there is a "penalty spike" for RH NOrec at 6 threads. The analysis shows that for more than 4 threads, RH NOrec is not able to reduce slow-path restarts as before, and this adds the additional overhead that results in a spike for 6 threads.

The Genome benchmark employs string matching to reconstruct a genome sequence from a set of DNA segments. This execution generates mostly moderate transactions with a low to moderate contention level, but the instrumentation costs in this benchmark are very high. As a result, the HTM-based schemes that eliminate most of the instrumentation improve significantly over the STMs for 1-8 threads. RH NOrec is 1.3 times faster than Hybrid NOrec for 8 threads, and above 8, the HyperThreading effect significantly increases the HTM capacity aborts, to the point that they dominate the slow-path ratio and eliminate the advantages of the Hybrid TMs. The analysis shows that RH NOrec has no significant effect on the slow-path restarts due to the low success ratio of the HTM prefix small hardware transactions. However, RH NOrec reduces the HTM conflicts by an order of magnitude, and this provides a lower slow-path fallback ratio compared to Hybrid NOrec.

Figure 6 presents the additional STAMP applications: Vacation-High, SSCA2 and Yada. We omit Kmeans and Labyrinth results, since they are similar to SSCA2, and omit Bayes due to its inconsistent behavior (as was also noted by [21]).

Now we explain the results for Figure 6 (from left to right).

Vacation-High is the more contended version of Vacation-Low. It generates additional heavier and slower transactions with moderate contention levels. As a result, all of the HTM-based schemes suffer penalties. However, RH NOrec behaves in the same way as for Vacation-Low: it reduces the HTM conflicts and slow-path restarts, and this allows the RH Norec to be 1.3 times faster than the Hybrid NOrec for 8 threads. In this case, RH NOrec provides less advantage than for Vacation-Low, and the reason for this is the higher number of HTM capacity aborts.

The SSCA2 kernel performs mostly uncontended small read-modify-write operations in order to build a directed,
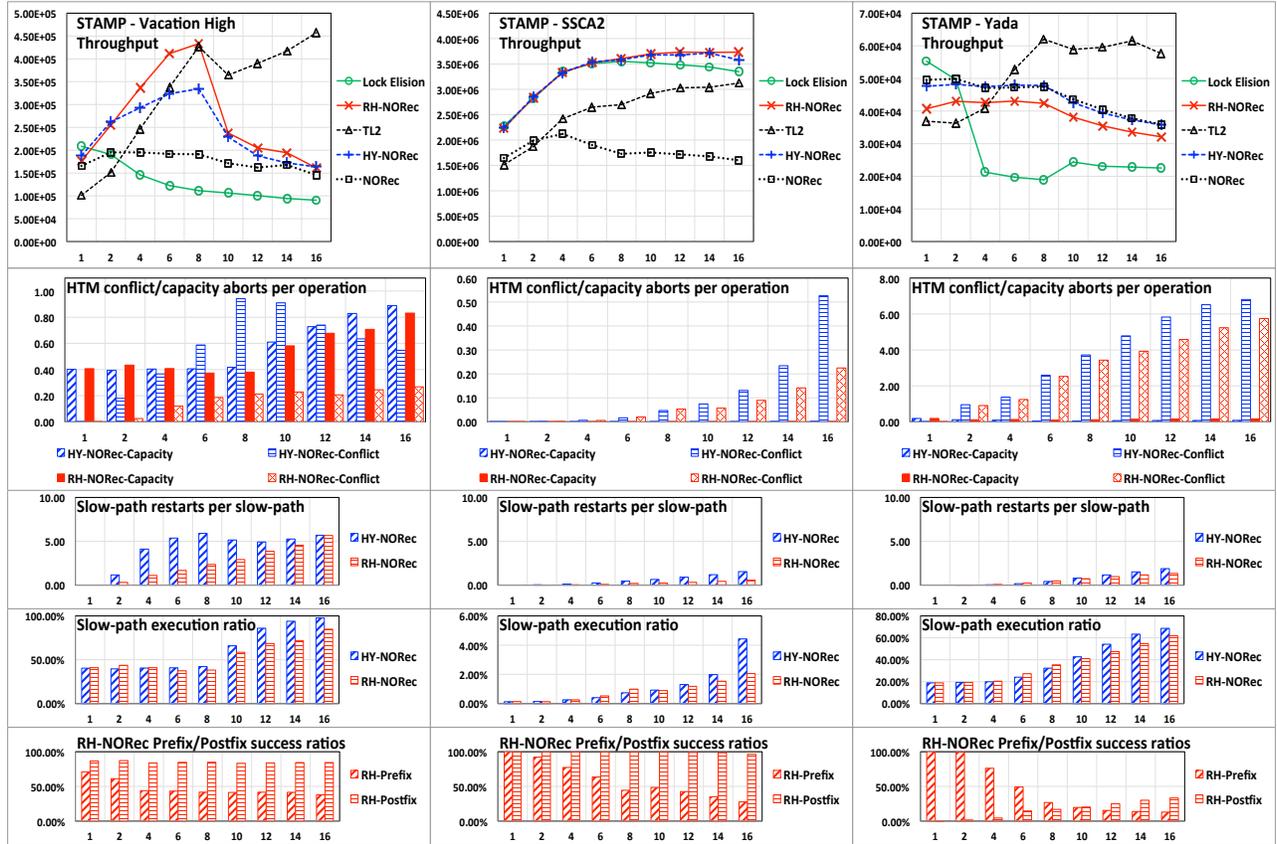
**Figure 6.** Results for Vacation-High, SSCA2 and Yada STAMP applications. The first (top) row shows the throughput, and the next rows show the execution analysis for the Hybrid TMs.

weighted multi-graph. In this case, the slow-path fallback ratio is very low, so all of the HTM-based schemes provide good results and scalability, and RH NORec's reduction in HTM conflicts and slow-path restarts is not significant for the overall performance. However, note that the reduction in HTM conflicts and slow-path restarts significantly increases when more threads are added, so for higher concurrency levels RH NORec should outperform the Hybrid NORec.

In Yada, threads cooperate to perform a Delaunay refinement, using Ruppert's algorithm. An initial mesh is provided as input, and threads iterate through the triangles of the mesh to identify those whose minimum angle is below some threshold (i.e., they find triangles that are "too skinny"). When such a triangle is found, the thread adds a new point to the mesh, and then re-triangulates a region around that point to produce a more visually appealing mesh. In this benchmark, most of the transactions are heavy, long and contending, so the HTM-based schemes suffer from huge overheads, and the RH-NORec small hardware transactions only introduce additional overheads, manifested in the low success ratio of the HTM prefix and postfix small hardware transactions.

## 4. Conclusion

We presented the RH NOrec hybrid TM, a new version of Hybrid NOrec that provides safety and improved scalability at the same time. The RH NOrec HTM fast-paths are fully safe (or opaque), and do not require special code to detect and handle runtime errors.

The core idea of RH NOrec is the mixed NORec slow-path that uses two short hardware transactions, the HTM postfix and HTM prefix, to reduce the HTM conflicts and the slow-path restarts. Of course, because of the limited concurrency available on the Haswell chips and the HyperThreading penalties, one must take our positive results here only as an indicator of the potential of RH NOrec to scale. Our hope is that as higher concurrency processors with HTM support are introduced, this potential will indeed pan out.

## Acknowledgments

# References

[1] A. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for c++, 2012. URL `https://sites.google.com/site/tmforcplusplus`.

[2] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: an automated transactional approach to concurrent memory reclamation. In D. C. A. Bulterman, H. Bos, A. I. T. Rowstron, and P. Druschel, editors, *EuroSys*, page 25. ACM, 2014. ISBN 978-1-4503-2704-6.

[3] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *ISCA*, pages 225–236, 2013.

[4] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In J. N. Amaral and J. Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 187–200. ACM, 2014. ISBN 978-1-4503-2809-8. . URL `http://doi.acm.org/10.1145/2628071.2628086`.

[5] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40, New York, NY, USA, 2010. ACM.

[6] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. . URL `http://doi.acm.org/10.1145/1693453.1693464`.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 46(3):39–52, Mar. 2011. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/1961296.1950373`.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/1168918.1168900`.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[10] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.

[11] N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-11-9.

[12] google-perftools. Tcmalloc : Thread-caching malloc, 2014. URL `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[13] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. .

[14] Intel. Transactional synchronization in haswell, 2012. URL `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`.

[15] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. . URL `http://doi.acm.org/10.1145/1122971.1123003`.

[16] Y. Lev, M. Moir, and D. Nussbaum. Phtm: Phased transactional memory. In *In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/ TRANSACT2007PhTM.pdf*, 2007.

[17] V. Marathe, M. Spear, and M. Scott. Scalable techniques for transparent privatization in software transactional memory. *Parallel Processing, International Conference on*, 0:67–74, 2008. ISSN 0190-3918. .

[18] A. Matveev and N.Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *SPAA*, pages 11–22, 2013.

[19] 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*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. . URL `http://doi.acm.org/10.1145/1989493.1989501`.

[20] W. Ruan, Y. Liu, and M. Spear. Stamp need not be considered harmful. In *TRANSACT 2014 Workshop, Salt Lake City, Utah, USA*, 2014.

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. . URL `http://doi.acm.org/10.1145/2503210.2503232`.