Reduced Hardware NOREC: An Opaque Obstruction-Free and Privatizing HyTM

Alexander Matveev
Tel-Aviv University
matveeva@post.tau.ac.il

Nir Shavit
MIT
shanir@csail.mit.edu

Abstract

This paper presents a reduced-hardware (RH) version of the promising NORec Hybrid TM algorithm. Instead of an all-software slow path, in RH transactions, part of the slow-path is executed using a short hardware transaction. The purpose of this hardware component is not to speed up the slow-path (though this is a side effect). Rather, using it we are able to eliminate virtually all of the instrumentation from the common hardware fast-path, requiring it to only access the shared "clock" of the NORec STM at the end of the hardware transaction. This improves on all prior work, including our own prior RH work, by presenting for the first time a Hybrid Transactional Memory that provides opacity with low hardware abort rates. Moreover, the "mostly software" slow-path is obstruction-free (no locking), privatizing, allows complete concurrency between hardware and software transactions, and uses the short hardware transactions only to write values during the software commit. We provide a simple slow-slow path in the unlikely case that both the hardware and mostly software paths fail.

For the concurrency levels we are able to test (a 4-core 8-way Haswell chip) the new algorithm exhibits promising performance.

1. Introduction

IBM and Intel have recently announced hardware support for best-effort hardware transactional memory (HTM) in upcoming processors [12, 13]. Best-effort HTMs impose limits on hardware transactions, but eliminate the overheads associated with loads and stores in software transactional memory (STM) implementations. Because it is possible for HTM transactions to fail for various reasons, a hybrid transactional memory (HyTM) approach has been studied extensively in the literature. It supports a best effort attempt to execute transactions in hardware, yet always falls back to slower all-software transactions in order to provide better progress guarantees and the ability to execute various systems calls and protected instructions that are not allowed in hardware transactions.

The first HyTM [6, 8] algorithms supported concurrent execution of hardware and software transactions by instrumenting the hardware transactions' shared reads and writes to check for changes in the STM's metadata. Riegel et al. [10] provide an excellent survey of HyTM algorithms to date, and the various proposals on how to reduce the instrumentation overheads in the frequently executed hardware fast-path: the key to good HyTM performance.

In the past two years, Dalessandro et al. [4] and Riegel et al. [10] have proposed Hybrid TMs based on the NORec STM.These are the most promising Hybrid TMs to date because they allow to limit the overall need to instrument instructions in the algorithms all hardware fast-path.

The first proposal, Hybrid NORec [4], is a hybrid version of the efficient NORec STM [5]. In it, write transactions' commits are executed sequentially and a global clock is used to notify concurrent read transactions about the updates to memory. The write commits trigger the necessary re-validations and aborts of the concurrently executing transactions. The great benefit of the NORec HyTM scheme over classic HyTM proposals is that no metadata per memory location is required and instrumentation costs are reduced significantly. However, in order to provide opacity in the hardware transactions, the global clock of the NORec STM must be read at the start of the hardware transaction, which adds it to the transaction's tracking set and causes high level of fast-path aborts. Using sandboxing in place of opacity is not a viable solution as it can compromise the correctness of transactional code.

The second proposal, by Riegel et al. [10], effectively reduces the instrumentation overhead of hardware transactions in HyTM algorithms based on both the LSA [11] and NORec [5] STMs. It does so by using non-speculative operations inside the hardware transactions to provide opacity. Unfortunately, these operations are supported by AMD's proposed ASF transactional hardware [2] but are not supported in the best-effort HTMs that IBM and Intel are bringing to the marketplace.

In a recent paper we presented the *reduced hardware* (RH) [9] approach to designing HyTM algorithms. Instead of an all-software slow path, in RH transactions, part of the slow-path is executed using a smaller hardware transaction, so what we have is actually a "mixed" hardware-software slow path. The purpose of this hardware component is not to speed up the slow-path (though this is a side effect). Rather, using it we were able to eliminate a large part of the instrumentation from the common hardware fast-path. If the mixed slow path fails one defaults to a slow slow path based on a global lock.

In [9] we presented an RH version of the TL2 STM, that eliminated the instrumentation overhead of reads in the fast-path instrumentation, and provided an efficient fast path for the HyTM, improving on all prior algorithms. However, there still remained several key problems with this algorithm:

- The fast-path avoided instrumenting reads but still instrumented writes. Since reads are typically 4 times more frequent than writes this is an improvement, but not as fast as a pure hardware transaction.

- In order to provide opacity and obstruction-freedom, the small hardware transaction in the slow path, needed to include a verification phase of the read locations. This meant that the chances of the short hardware transaction failing were still high as it included both the read and write sets.
- The slow path was not privatizing, a problem if the algorithm were to really be deployed in a commercial setting.
- The default slow-slow path was blocking, limiting the concurrency between hardware and software transactions.

In this paper we present for the first time an RH version of the NORec Hybrid TM algorithm. The key idea is a way of applying the RH approach to the NORec algorithm, so that unlike prior NORec-based HyTM proposals, reading the shared global clock of the NORec STM happens only at the end of the hardware transaction. The short hardware transactions are used only to write the write-set values during the software commit, and we show how to default to an all software NORec "slow-slow path" in the rare case where both the fast and slow paths repeatedly fail to commit.

This new algorithm overcomes the drawbacks of all prior HyTM algorithms, including the prior Hybrid NORec proposals, and improves our own prior work by providing:

- The first HyTM that has a fast-path with no instrumentation at all, not of reads or of writes.
- The first HyTM that has a fast-path and slow-path that are all opaque, obstruction-free, privatizing, and allow complete concurrency between hardware and software transactions.
- The first HyTM that uses only a short hardware transaction in the mixed path, consisting only of the write set, and can thus be attempted repeatedly until it succeeds without a need for revalidation.

Our empirical testing of the new RH NORec algorithm on a state-of-the-art 4-core Haswell chip provides encouraging evidence of the potential effectiveness of the reduced hardware approach [1] We believe that our encouraging results and the complete obstruction-freedom of the new algorithm's fast and slow paths raise interesting questions about the tradeoffs of using a combination of hardware and software in the previously software-only slow-path of Hybrid TMs, and more generally, in the cost of the opacity property itself.

The paper is organized as follows. We begin by an overview of our new hybrid protocol, and then get into the details and the limitations of the algorithm. We then show a version of the algorithm that takes advantage of IBM's proposed new supend-resume operations. Finally, we show the results of our empirical testing.

2. Reduced Hardware NORec

Here, in a nutshell, is how our new hybrid TM protocol works. The RH NORec protocol has a multi-level fallback mechanism: for any transaction it first tries a pure hardware fast path. If this fails it tries a new *mixed slow-path*, and if this fails, it tries an all software *slow-slow-path*.

On the slow-path, RH NORec executes the original NORec STM transaction [5]. The transaction body is executed purely in software. It collects a read-set and a write-set, postpones the actual data writes to the commit phase, and performs current read-set value-based revalidation on every NORec global clock change. The key new element in RH NORec is that the commit-time write-back of the new values is executed within a single speculative hardware transaction. The commit saves the current global clock value, starts read-set value-based revalidation and then initiates a small hardware transaction, which first verifies that the current global clock is equal to the saved one. This clock check verifies that the read-set revalidation that was just performed is still valid within the hardware transaction. Then the small hardware transaction performs the writes of the write-set and updates the global clock. Unlike the original Hybrid NORec, there are no locks, and the slow-path transaction is obstruction-free. Moreover, the short hardware transaction can be repeated several times until it succeeds without any loss of correctness.

This change in the slow-path, turning it into a mixed slow-path, allows us to implement the hardware fast-path transactions without reading the NORec global clock on every fast-path transaction start. Instead, the fast-path is only required to update the global clock upon every fast-path commit of a transaction [2]. As a result, the RH NORec avoids many of the original false aborts that limited Hybrid NORec's scalability (see the analysis in Section 4). Intuitively, this update only during the commit suffices because for any slow-path transaction, concurrent hardware transactions will either see all the new values written, or all the old ones, but will fail if they read both new and old versions because this means they overlapped with the slow-path's hardware commit.

How likely-to-fail is the hardware part of the mixed slow-path transaction? Because in the slow-path, the transaction body is executed purely in software, any system calls and protected instructions that might have failed the original hardware transaction can now complete in software before the commit point. In the commit point, the small hardware transaction performs only the actual writes, so the hardware requirements are reduced to be only the write-set locations, and there is no requirement to speculate on the read-set locations. Still, the commit write-back may fail due to hardware capacity limitations, because the write-set is too large; but these cases are usually rare, and if they happen the algorithm will, as we explain later, fallback to a slow-slow mode, where concurrent hardware and software transactions run the original Hybrid NORec.

Algorithm 1 RH NORec fast-path transaction

```
1:  function FAST_PATH_START(ctx)
        ▷ no instrumentation - only start the hardware transaction
2:      HTM_Start()
3:  end function
4:
5:  function FAST_PATH_WRITE(ctx, addr, value)
        ▷ no instrumentation - simply write the location
6:      store(addr, value)
7:  end function
8:
9:  function FAST_PATH_READ(ctx, addr)
        ▷ no instrumentation - simply read the location
10:     return load(addr)
11: end function
12:
13: function FAST_PATH_COMMIT(ctx)
        ▷ increment the global clock to notify other transactions about
    possible update to the memory
14:     global_clock ← global_clock + 1
15:     HTM_Commit()
16: end function
```

[1] This is the first such empirical proof as our earlier paper [9] was a simulation published before Haswell chips were available. Obviously it would be interesting to test the approach when larger Haswell chips are available.

[2] In actuality it only needs to update it for transactions that write.

Algorithm 2 RH NORec mixed slow-path transaction

```
1: function SLOW_PATH_START(ctx)
2:     ctx.tx_version ← global_clock
3: end function
4:
5: function SLOW_PATH_WRITE(ctx, addr, value)
       ▷ add to write-set
6:     ctx.write_set ← ctx.write_set ∪ {addr, value}
7: end function
8:
9: function SLOW_PATH_READ(ctx, addr)
       ▷ check if the location is in the write-set
10:    if addr ∈ ctx.write_set then
11:        return the value from the write-set
12:    end if
13:    cur_value ← load(addr)
       ▷ log the read and revalidate if required
14:    ctx.read_set ← ctx.read_set ∪ {addr, cur_value}
15:    if ctx.tx_version ≠ global_clock then
16:        ctx.tx_version ← global_clock
17:        if ¬revalidate_read_set_value_based(ctx) then
18:            stm_abort(ctx)
19:        end if
20:    end if
21:    return cur_value
22: end function
23:
24: function SLOW_PATH_COMMIT(ctx)
       ▷ read-set revalidation
25:    label: start_revalidate
26:    if ctx.tx_version ≠ global_clock then
27:        ctx.tx_version ← global_clock
28:        if ¬revalidate_read_set_value_based(ctx) then
29:            stm_abort(ctx)
30:        end if
31:    end if
32:    HTM_Start()
       ▷ write the values
33:    for addr, new_value ∈ ctx.write_set do
34:        store(addr, new_value)
35:    end for
       ▷ verify that read-set revalidation is still valid and update the clock
36:    if ctx.tx_version ≠ global_clock then
37:        htm_abort(ctx)
38:    end if
39:    global_clock ← global_clock + 1
40:    HTM_Commit()
41:    if the HTM failed NOT due to capacity then
42:        goto start_revalidate
43:    else
44:        fallback to slow-slow mode
45:    end if
46: end function
```

Algorithm 3 RH NORec modifications for the all-software slow-slow fallback

```
1: function FAST_PATH_START(ctx)
2:     HTM_Start()
       ▷ Verify that there are no concurrent all-software write-backs in the process
3:     if is_taken(is_all_soft_lock) then
4:         HTM_Abort()
5:     end if
6: end function
7:
8: function SLOW_PATH_READ(ctx, addr)
9:     .....
10:    cur_value ← load(addr)
       ▷ Wait for concurrent all-software write-backs to finish
11:    while is_taken(is_all_soft_lock) do
12:        do nothing
13:    end while
14:    .....
15: end function
16:
17: function SLOW_PATH_COMMIT(ctx)
18:    .....
19:    HTM_Start()
20:    .....
       ▷ verifies that there is no all-software fallbacks
21:    if is_taken(is_all_soft_lock) then
22:        HTM_Abort()
23:    end if
24:    .....
25:    HTM_Commit()
26:    if the HTM failed NOT due to capacity then
27:        goto start_revalidate
28:    else
       ▷ execute all-software commit
29:        Slow_Slow_commit(ctx)
30:    end if
31: end function
32:
33: function SLOW_SLOW_COMMIT(ctx)
34:    lock_acquire(is_all_soft_lock)
35:    if ¬revalidate_read_set_value_based(ctx) then
36:        lock_release(is_all_soft_lock)
37:        stm_abort(ctx)
38:    end if
39:    for addr, new_value ∈ ctx.write_set do
40:        store(addr, new_value)
41:    end for
42:    global_clock ← global_clock + 1
43:    lock_release(is_all_soft_lock)
44: end function
45:
```

3. Algorithm Details

Algorithm 1 shows the RH NORec fast-path implementation. On start, it initiates a hardware transaction (line 2), and during the execution performs completely pure reads and writes (line 10 and 6) without any instrumentation. On commit, it increments the global clock and commits the hardware transaction (lines 14-15). Note that for RH hybrid correctness, the global clock update at the fast-path commit is only required for a fast-path transaction that made a write, and only when there is a concurrent slow-path transaction.

Algorithm 2 shows the RH NORec mixed slow-path implementation. On start, it reads the global clock to a local variable called *tx_version* (line 2). During the execution, the transaction performs its writes to a local write-set buffer (line 6) , and on reads, it scans the write-set for the read locations (lines 10-11). If the read location is found in the write-set, then it returns its value from there. Otherwise it reads the read location from the memory, adds it to a read-set buffer, and verifies that the global clock has not been changed, by comparing it to the *tx_version* local variable. In case it detects a clock change, it triggers a read-set revalidation, and upon a successful read-set pass, the *tx_version* variable is updated to the new clock value (lines 13-21). On commit, the transaction samples the global clock to a *local_global_clock* local variable, and executes the read-set revalidation (lines 26-31). Then, it starts a small hardware transaction that verifies that the clock has not been changed, performs the actual writes, and increments the clock by 1 (lines 32-40). If the short hardware transaction fails, the transaction restarts the commit (lines 41-44). It is possible to restart as long as there is no real conflict (revalidation failure) or no real hardware limitation (capacity problem).

Algorithm 4 RH NORec optimization - using non-speculative operations

```
 1: function SLOW_PATH_OPT_COMMIT(ctx)
        ▷ STEP 1: put the write-set locations into speculation (to be
    monitored by HTM), and suspend HTM
 2:     HTM_Start
 3:     for addr ∈ ctx.write_set do
 4:         cur_value ← load(addr)
 5:         store(addr, cur_value)
 6:     end for
 7:     HTM_Suspend
        ▷ STEP 2: perform read-set revalidation outside HTM
 8:     if ctx.tx_version ≠ global_clock then
 9:         ctx.tx_version ← global_clock
10:         if ¬revalidate_read_set_value_based(ctx) then
11:             stm_abort(ctx)
12:         end if
13:     end if
        ▷ STEP 3: resume HTM and finish
14:     HTM_Resume
15:     for addr, new_value ∈ ctx.write_set do
16:         store(addr, new_value)
17:     end for
18:     global_clock ← global_clock + 1
19:     HTM_Commit
20:     if the HTM failed NOT due to capacity then
21:         goto STEP 1
22:     else
23:         fallback to slow-slow mode
24:     end if
25: end function
26:
```

They key point of this design is that the hardware fast-path performs the global clock update only at the commit. This is possible due to the new mixed slow-path commit-time atomic write-back, which is done by using a small hardware transaction. The atomic slow-path write-back hides the intermediate updates, and exposes only all of the writes or none of them to the concurrent fast-path transactions. As a result, fast-path transactions cannot see partial updates of the slow-paths, which involve some new and some old values, and can see only all of the new values or all of the old ones. In contrast, the original Hybrid NORec slow-path commit write-back is executed piecemeal, write after a write, so it is possible for the fast-paths to see slow-paths partial updates, and it is necessary for the fast-paths to read the global clock on start, so that they will immediately detect and abort upon a slow-path update initiation.

3.1 Fallback to an all-software slow-slow path

RH NORec uses a small hardware transaction to perform the slow-path commit write-back atomically. This is crucial for the correctness of the hybrid protocol, and reduces the hardware requirements to be only the set of the write locations, not including the set of the read locations. Therefore, a constant failure of this small hardware transaction blocks the slow-path transaction from progress. This may happen due to some hardware limitation, for example when the set of the write locations cannot fit into the L1 cache. We expect this situations to be rare, but still it may happen, and in this case we provide a *slow-slow* mode fallback for the RH-NORec protocol.

Algorithm 3 shows the code modifications required to support the all-software slow-slow mode. The slow-slow mode fallback is similar to the original HY-NORec implementation that uses a global lock. When the slow-path commit detects a constant failure of the small hardware transaction (lines 26-29), it retries in a slow-slow commit mode where it acquires the global lock. Then, while the lock is taken, it performs the read-set revalidation, the write-back with global clock update, and the global lock release (lines 34-43). The hardware fast-path transactions read this global lock variable on the start and verify that it is free (lines 3-4). Since this variable is cached and we expect execution of the slow-slow mode to be rare, the cost of reading this lock variable is negligible. The fast-path hardware transactions abort upon a first fallback to the slow-slow commit, and wait for it to finish. In addition, the mixed-path reads inspect the global lock immediately after the read of the location, and if the lock is acquired, then spin-loop on it till its free (lines 11-12). Also, we disallow concurrent slow-path commits while there is slow-slow mode commit by making the slow-path commit small hardware transaction verify that the global lock is not taken (lines 21-22).

3.2 Algorithm optimization for HTM with non-speculative operations

The new IBM Power 8 ISA transactional memory specification [1] defines a hardware transactional memory system with *suspend-resume* operations. They allow to suspend a hardware transaction, so that a non-transactional code can execute, and then resume the transaction execution. An RH NORec algorithm based on this feature can have an improved slow-path commit implementation that completely eliminates the potential global clock abort window.

RH NORec slow-path commit performs the following steps: (1) samples the global clock, (2) revalidates the read-set, (3) executes a small hardware transaction that writes the write-set locations atomically, and (4) revalidates that the current global clock is equal to the one it has read before (in step 1). As a result, the slow-path commit will restart itself, if the global clock changes between steps (1) and (4). We can reduce this abort window if the hardware allows non-speculative (non transactional) memory operations inside a hardware transaction.

The main idea behind the new slow-path commit is to use the hardware speculation as a protection for the write locations. The new slow-path commit starts by executing a small hardware transaction that writes to every write location its current value and then suspends itself. This puts the write locations into hardware monitoring, and now it performs the read-set value-based revalidation outside of the hardware transaction. Upon revalidation success, it resumes the small hardware transaction, writes the new values to the write locations, and commits it. If the hardware fails to commit, it restarts the slow-path commit procedure. The whole slow-path transaction is restarted only when there is a real conflict (revalidation failure) or a real hardware limitation (capacity problem).

The new slow-path commit is correct because any concurrent read or write of any of the monitored write locations will disallow the small hardware transaction from committing successfully. The behavior is similar to locking the write locations, with the difference that if someone touches a write location then it proceeds and aborts the committing transaction. As a result, there is no need for global clock reads and checks during the commit procedure, and the only requirement is to increment the global clock on the commit finish.

Algorithm 4 shows the new slow-path commit implementation. The fast-path and the rest of the functions remain the same for this version of RH NORec.

4. Performance Evaluation

We evaluated the performance of our new RH NORec algorithm on an 8-way Intel Haswell chip with 4 cores, each multiplexing 2 hardware threads (HyperThreading). For our testing we used a red-black tree benchmark. The algorithms we benchmarked were:

HTM *Hardware TM:* Transactions execute as pure hardware transactions using the Intel Haswell RTM mechanism [13], and on

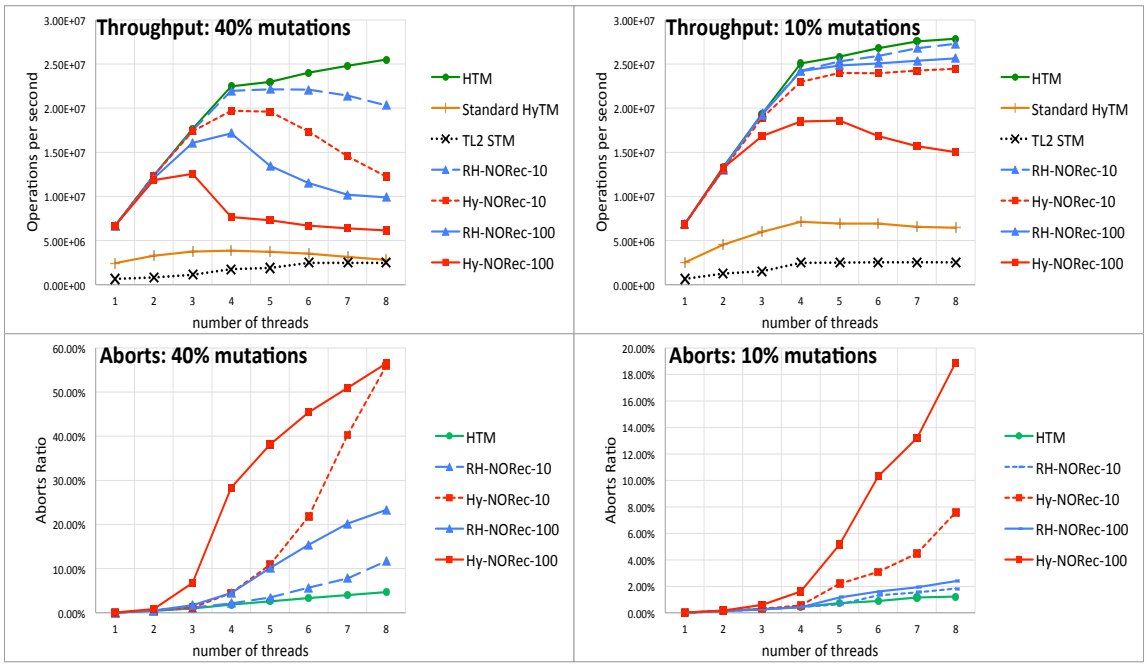

Figure 1. Red-Black tree results. The upper graphs show the throughput for 40% and 10% mutations in the tree, and the bottom graphs show the corresponding abort rates. We can see a clear performance advantage of RH-NORec over HY-NORec, and from the lower graphs it is clear that this is completely correlated with the higher rate of HY-NORec aborts due to reading the global clock into the HTM transactional at the start of each transaction. It is also encouraging to see that RH-NORec delivers performance that is very close to that of pure HTM.

failure restart as pure hardware transactions. This indicates the best performance that can be achieved by the HTM mechanism.

Standard HyTM *The Standard Hybrid Transactional Memory:* A state-of-the-art hybrid TL2-Style TMs [10]. The software slow-path executes a TL2 STM and the hardware fast-path reads and writes inspect the per location metadata (using a single "if" condition check).

TL2 A TL2 STM implementation [7], that uses a GV6 global clock.

HY-NORec *Original Hybrid Norec:* The algorithm of Dalessandro et. al [4]. The hardware fast-path reads the global clock on start and increments it on commit. The software slow-path executes the NOrec STM. There are two variants: HY-NORec-10 and HY-NORec-100. The first executes 10% of the fast-path aborted transactions in the mixed slow-path and the remaining 90% retry the fast-path, while the second executes 100% of the aborts in the slow-path.

RH-NORec *Reduced Hardware NORec:* This is our new hybrid TM. The hardware fast-path only updates the global clock at the end of the transaction during the hardware commit, the mixed software slow-path executes the transaction body in pure software, and the transaction commit writes by using a small hardware transaction. In a similar way to HY-NORec, there are two variants: RH-NORec-10 and RH-NORec-100. The first executes 10% of the fast-path aborts in the mixed slow-path and the remaining 90% of transactions retry in the fast-path; the second executes 100% of the aborts in the slow-path.

The red-black tree we use was derived from the *java.util.TreeMap* implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from Cormen et al. [3]. The red-black tree implementation exposes a key-value pair interface of *put*, *delete*, and *get* operations. If the key is not present in the data structure, put will put a new element describing the key-value pair. If the key is already present in the data structure, put will simply insert the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure.

The benchmark allows us to control the tree size and the fraction of write transactions executed, called mutation ratio. We execute every run for 10 seconds, and report the average number of operations completed per second.

Figure 1 shows the results for a red-black tree with 1K nodes and 40% and 10% rates of mutation respectively. The top two graphs show the throughput results, and bottom two graphs show the abort ratios. We performed the same benchmarking for larger trees, with 5K-10K nodes, and saw similar results. Increasing the tree size beyond 10K nodes makes the hardware fast-path abort too often, so that most of the time slow-paths execute, and the advantage of using an HTM is lost.

Looking at Figure 1, we note that pure hardware transactions executed using the Intel RTM hardware transactional mechanism have a performance deterioration after 4 threads. The reason for this is the HyperThreading mechanism that multiplexes additional new threads, so 2 threads run on every core, from 5 to 8 threads. This makes every 2 threads on the same core share an L1 cache, on which they conflict often.

Looking at the results of our benchmark, we can see that there is a big advantage of HTM over the TL2 STM, and that the Standard HyTM algorithms eliminate almost all of this advantage due to

their need to inspect metadata on each read or write. Standard HyTM performance is close to that of the TL2 STM and is very far from the HTM's potential. The HY-NORec and the RH-NORec algorithms eliminate the Standard HyTM instrumentation from the fast-path hardware transactions, and accordingly achieve a better performance.

In the 40% mutation benchmark (upper left graph) we have two types of executions for the RH-NORec and HY-NORec. One that forwards 10% of the hardware fast-path aborts to the mixed slow-path and the remaining 90% retry again in the fast-path, and another that forwards all of the 100% of the aborts to the slow-path. This percentage is indicated by the line name. We can see that RH-NORec-10 outperforms HY-NORec-10 by a factor of 1.7, and RH-NORec-100 outperforms HY-NORec-100 by a factor of 2.4. Overall, RH-NORec is able to get very close to the HTM's performance, and we can see this with RH-NORec-10.

The performance difference is perhaps mostly explained by the difference in the algorithm's abort rates. Analysis of the abort ratios for the 40% mutation case (bottom left graph) shows us that there is a significant difference in the aborts between the RH-NORec and HY-NORec. The lines correspond to the algorithms in the upper left graph of throughput. For the 10% slow-paths case, HY-NORec suffers a 5 times higher abort rate compared to the RH-NORec, and for the 100% slow-path its abort rate is still 2 times higher. The main reason for this is the fact that HY-NORec reads the global clock on the hardware fast-path start. As a result, a HY-NORec slow-path update of the global clock triggers an abort of all current hardware transactions, which introduces unnecessary aborts. In contrast, the RH-NORec fast-paths access the global clock only at the commit point, and therefore avoids all of these aborts.

In the 10% mutation benchmark (upper right graph), we can see that there is almost no difference between RH-NORec-10 and RH-NORec-100. Both of them exhibit a very low abort ratio. But, there is a difference for HY-NORec-10 and HY-NORec-100, where HY-NORec-10 is able to get close to RH-NORec performance. This is due to the sensitivity of the HY-NORec to slow-path aborts that may result in a system-wide abort of all hardware transactions. The HY-NORec-10 exhibits 7% aborts in total (look at the bottom right part of the graph), while the HY-NORec-100 exhibits as high as 19% aborts; this makes a big difference in the performance.

Analysis of the aborts for the 10% mutation case shows the same behavior as for the 40% mutation. RH-NORec incurs approximately 2% aborts in general, and HY-NORec incurs 7% and 19%, which is a 3 - 10 times difference. As before, we can see a correlation between the aborts and the resulting throughput.

References

[1] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *ISCA*, pages 225–236, 2013.

[2] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne 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.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition edition, 2001.

[4] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. *SIGPLAN Not.*, 46(3):39–52, March 2011.

[5] Luke Dalessandro, Michael F. Spear, and Michael 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.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, October 2006.

[7] 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.

[8] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony 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.

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

[10] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof 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.

[11] P. Felber T. Riegel and C. Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[12] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[13] Web. Intel tsx http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell, 2012.