Reduced Hardware NOREC

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

Nir Shavit
MIT and Tel-Aviv University
shanir@csail.mit.edu

Abstract

The most promising Hybrid TM algorithms to date have been those designed around the NORec STM, because they allow to limit the overall need to instrument instructions in the algorithms all hardware fast-path. However, in order to provide opacity in the hardware transactions, the shared "clock" of the NORec STM must be read at the start of the hardware transaction, which adds it to the track set and causes high level of fast-path aborts. Solutions to this problem have required sandboxing or the use of special non-speculative instructions that are not available on current hardware.

In a recent paper we presented the *reduced hardware* (RH) approach to designing Hybrid TM algorithms. Instead of an all-software slow path, in RH transactions, part of the slow-path is executed using a smaller 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 one can eliminate a large part of the instrumentation from the common hardware fast-path.

In this paper we present an RH version of the NORec Hybrid TM algorithm. It only reads the shared "clock" of the NORec STM at the end of the hardware transaction, thus providing opacity with low hardware abort rates. Moreover, the "mostly software" slow-path is obstruction-free (no locking), allows complete concurrency between hardware and software transactions, and uses the short hardware transactions only to write values during the software commit.

The performance benefits and complete obstruction-freedom of the new RH NORec algorithm raise interesting questions about the tradeoffs of using a combination of hardware and software in the previously software-only slow-path of Hybrid TMs.

1 Introduction

IBM and Intel have recently announced hardware support for best-effort hardware transactional memory (HTM) in upcoming processors [7, 8]. 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.

Riegel et al. [5] 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. [2] and Riegel et al. [5] 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 [2], is a hybrid version of the efficient NORec STM [3]. 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 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 (and thus avoid the need for sandboxing), 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.

The second proposal, by Riegel et al. [5], effectively reduces the instrumentation overhead of hardware transactions in HyTM algorithms based on both the LSA [6] and NORec [3] STMs. It does so by using non-speculative operations inside the hardware transactions. Unfortunately, these operations are supported by AMD's proposed ASF transactional hardware [1] 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) approach to designing Hybrid TM algorithms. Instead of an all-software slow path, in RH transactions, part of the slow-path is executed using a smaller 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 a large part of the instrumentation from the common hardware fast-path.

In this paper we present for the first time an RH version of the NORec Hybrid TM algorithm. It overcomes the drawbacks of the above Hybrid NORec proposals by reading the shared global clock of the NORec STM only at the end of the hardware transaction, thus providing opacity with low hardware abort rates. Moreover, the "mostly software" slow-path is obstruction-free (no locking), allows complete concurrency between hardware and software transactions, and uses the short hardware transactions only to write values during the software commit, which is different from our previous RH design, where the small hardware transactions were required to speculate on the read locations' metadata. The result, as we show, is the best performing HyTM to date.

We believe that the performance and complete obstruction-freedom of the new RH NORec algorithm 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.

2 Reduced Hardware NORec

Algorithm 1 RH NORec fast-path transaction implementation

```
1:  function RH_NOREC_FP_START(ctx)
        ▷ no instrumentation - only start the hardware transaction
2:      HTM_Start()
3:  end function
4:
5:  function RH_NOREC_FP_WRITE(ctx, addr, value)
        ▷ no instrumentation - simply write the location
6:      store(addr, value)
7:  end function
8:
9:  function RH_NOREC_FP_READ(ctx, addr)
        ▷ no instrumentation - simply read the location
10:     return load(addr)
11: end function
12:
13: function RH_NOREC_FP_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
```

Here, in a nutshell, is how the our hybrid 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. The transaction body is executed purely in software. It collects read-set and 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

Algorithm 2 RH NOREC slow-path transaction implementation

```
1: function RH_NOREC_SP_START(ctx)
2:     ctx.tx_version ← global_clock
3: end function
4:
5: function RH_NOREC_SP_WRITE(ctx, addr, value)
       ▷ add to write-set
6:     ctx.write_set ← ctx.write_set ∪ {addr, value}
7: end function
8:
9: function RH_NOREC_SP_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:        if ¬revalidate_read_set_value_based(ctx) then
17:            stm_abort(ctx)
18:        end if
19:    end if
20:    return cur_value
21: end function
22:
23: function RH_NOREC_SP_COMMIT(ctx)
       ▷ read-set revalidation
24:    label: start_revalidate
25:    local_global_clock ← global_clock
26:    if ¬revalidate_read_set_value_based(ctx) then
27:        stm_abort(ctx)
28:    end if
29:    HTM_Start()
       ▷ verifies read-set revalidation is still valid
30:    if local_global_clock ≠ global_clock then
31:        htm_abort(ctx)
32:    end if
       ▷ write the values and update the global clock
33:    for addr, new_value ∈ ctx.write_set do
34:        store(addr, new_value)
35:    end for
36:    global_clock ← global_clock + 1
37:    HTM_Commit()
       ▷ we can safely restart the commit revalidation,
   if the failure reason is not hardware capacity limitation
38:    if the HTM failed NOT due to capacity then
39:        goto start_revalidate
40:    else
41:        fallback to slow-slow mode
42:    end if
43: end function
```

key new element in RH NORec, is that the commit-time write-back of the new values, executes in 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 check verifies that the read-set revalidation is still valid, and then the small hardware transaction performs the writes and global clock update. Unlike the original Hybrid NORec, there are no locks, and the transaction is obstruction-free.

This change in the slow-path allows us to implement the hardware fast-path transactions without reading the NORec global clock on every fast-path transaction start. Instead, we only required to update the global clock on every fast-path transaction commit. As a result, the RH NORec avoids many of the original Hybrid NORec's false aborts that limited its scalability. Intuitively, this 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.

Algorithm 1 and 2 show the pseudo-code implementation for RH NORec hardware fast-path and software slow-path respectively.

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 this cases are usually rare, and if they happen we fallback to the slow-slow mode, where concurrent hardware and software transactions run the original Hybrid NORec.

3 Performance Evaluation

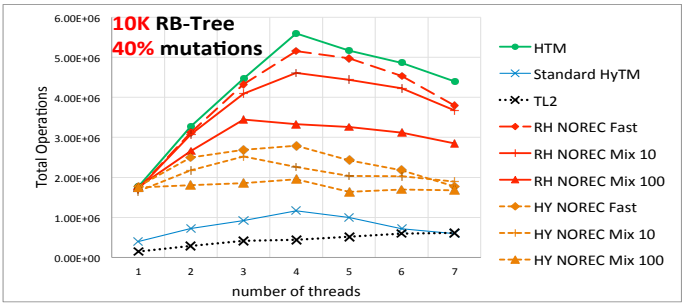

Figure 1: The graphs show the throughput of 10K sized Red-Black Tree for 40% mutations.

We execute the benchmarks on Intel 8-way Haswell chip with 2 cores, each multiplexing 2 hardware threads (HyperThreading). For our testing we use a red-black tree benchmark. The algorithms we benchmark are:

HTM *Hardware Transactional Memory:* all of the transactions are executed with Intel Haswell RTM mechanism without any additions.

Standard HyTM *The Standard Hybrid Transactional Memory*: This represents the best performance that can be achieved by current state-of-the-art hybrid TL2-Style TMs [5]. To make this hybrid as fast as possible, we execute only the hardware fast-path transactions, by executing and retrying all of the transactions only in the hardware fast-path mode (there is no fallback to the software slow-path).

HY NOREC *Original Hybrid Norec:* The hardware fast-path reads the global clock on start, and the software slow-path executes STM Norec. There are three variations: Mix 10, Mix 100 and Fast. For the first 10% of the fast-path aborts retry in the slow-path and for the second it is 100%. The Fast version retries all of the aborts in the fast-path.

RH NOREC *Reduced Hardware Norec:* This is our new hybrid TM. The hardware fast-path only updates the global clock on hardware commit, and the mixed software slow-path executes the transaction body in pure software and the transaction commit writes in a small hardware transaction. In a similar way to HyNORec there are three variations: Mix 10, Mix 100 and Fast.

TL2 This is the usual TL2 STM implementation [4], that uses a GV6 global clock.

Figure 1 shows the results for the different algorithms on a red-black tree of size 10K nodes and 40% mutation operations. First, we can see that the Standard HyTM performs close to TL2 STM, eliminating all the HTM performance benefits. This due to the fact that Standard HyTMs instrument every fast-path read and write with STM's metadata inspection. Hybrid Norec delivers better performance than the Standard HyTMs, but still it incurs a high degradation relative to the HTM performance. Even Hybrid Norec Fast, that retries all of the aborted transaction in the fast-path, and never uses the slow-path mode, performs much worser than the HTM. In contrast, the RH-Norec Fast and Mix 10 are able to preserve close to HTM performance, while Mix 100 degrades because of excessive use of the slow-path. But still, RH-Norec Mix 100 is better than the Hybrid Norec Fast. The main reason for RH-Norec advantage over Hybrid Norec, is that RH-Norec hardware fast-path transactions only update the global clock on hardware commit, while the Hybrid Norec fast-path transactions read this global clock on hardware start and update it on hardware commit. Therefore, RH-Norec eliminates the Hybrid Norec's false aborts related to global clock updates.

4 RH NORec - Optimization for HTM that supports non-speculative operations

Algorithm 3 RH NORec optimization - Fast-Path transaction

```
1: function RH_NOREC_OPT_FP_START(ctx)
2:     HTM_Start()
3: end function
4:
5: function RH_NOREC_OPT_FP_WRITE(addr, value)
       ▷ no instrumentation - simply write the new value to the memory
6:     store(addr, value)
7: end function
8:
9: function RH_NOREC_OPT_FP_READ(ctx, addr)
       ▷ no instrumentation - simply read the location
10:    return load(addr)
11: end function
12:
13: function RH_NOREC_OPT_FP_COMMIT(ctx)
14:    seq_clock ← seq_clock + 1
15:    HTM_Commit()
16: end function
```

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.

IBM new Power 8 ISA transactional memory specification defines a hardware transactional memory system with a *suspend-resume* operations. They allow to suspend a hardware transaction, so that a non-transactional code can execute, and then resume the transaction execution. RH NORec algorithm based on this feature has a different slow-path commit implementation. The new slow-path commit starts a hardware transaction that writes to every write location its current value, suspends itself, and then does the read-set value-based revalidation (non-speculatively). On revalidation success, it resumes the hardware transaction, writes the new values to the write locations and commits the hardware speculation. If the hardware fails to commit, it restarts the slow-path commit procedure. The whole slow-path transaction is restarted only when the read-set value-based revalidation fails.

Algorithm 3 and Algorithm 4 show the implementation of *RH NORec* fast-path and slow-path transactions for a processor that allows non-speculative operations inside a hardware transaction.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

Algorithm 4 RH NORec optimization - Slow-Path transaction

```
1: function RH_NOREC_OPT_SP_START(ctx)
2:     ctx.local_seq_clock ← seq_clock
3: end function
4:
5: function RH_NOREC_OPT_SP_WRITE(addr, value)
       ▷ add to write-set
6:     ctx.write_set ← ctx.write_set ∪ {addr, value}
7: end function
8:
9: function RH_NOREC_OPT_SP_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
       ▷ log the read
13:    value ← load(addr)
14:    ctx.read_set ← ctx.read_set ∪ {addr, value}
15:    if ctx.local_seq_clock ≠ seq_clock then
16:        ctx.local_seq_clock ← seq_clock
17:        if ¬ revalidate_read_set(ctx) then
18:            TxAbort(ctx)
19:        end if
20:    end if
21:    return value
22: end function
23:
24: function RH_NOREC_OPT_SP_COMMIT(ctx)
       ▷ read-only transactions commit immediately
25:    if ctx.write_set is empty then
26:        return
27:    end if
       ▷ STEP 1: put the write-set locations into speculation (to be monitored by HTM)
28:    HTM_Start
29:    for addr ∈ ctx.write_set do
30:        cur_value ← load(addr)
31:        store(addr, cur_value)
32:    end for
33:    HTM_Suspend
       ▷ STEP 2: read-set revalidation
34:    if ctx.local_seq_clock ≠ seq_clock then
35:        ctx.local_seq_clock ← seq_clock
36:        if ¬ revalidate_read_set(ctx) then
37:            TxAbort(ctx)
38:        end if
39:    end if
40:    HTM_Resume
41:    for addr, new_value ∈ ctx.write_set do
42:        store(addr, new_value)
43:    end for
44:    seq_clock ← seq_clock + 1
45:    HTM_Commit
46:    if HTM failed then
47:        goto STEP 1
48:    end if
49: end function
50:
51: function REVALIDATE_READ_SET(ctx)
52:    for addr, old_value ∈ ctx.read_set do
53:        cur_value ← load(addr)
54:        if cur_value ≠ old_value then
55:            return false
56:        end if
57:    end for
58:    return true
59: end function
```


[5] 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.

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

[7] 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.

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