Reduced Hardware Lock Elision

Yehuda Afek
Tel-Aviv University
afek@post.tau.ac.il

Alexander Matveev
MIT
matveeva@post.tau.ac.il

Nir Shavit
MIT
shanir@csail.mit.edu

Abstract

Hardware lock elision (HLE) concurrently executes lock critical sections as hardware transactions, but fallbacks to the original sequential lock fallback path when some hardware transaction fails. Recent software-assisted *lock-removal* based schemes provide a better concurrency by sacrificing safety (opacity). Hardware transactions can execute at the same time with the lock fallback path as long as they do not try to commit. This limited concurrency is beneficial, but enables hardware transactions to see inconsistent memory states, that may lead to illegal instructions, corrupted memory, and other unsafe behaviors.

We propose a novel *reduced hardware lock elision* algorithm (RH-LE). It provides a safe (and opaque) concurrency between hardware transactions and the lock fallback path. The core idea behind the RH-LE approach is to execute the lock fallback path as a *rollback-only hardware transaction*, that tracks only the memory writes of the transaction. This special hardware transaction is already introduced in the IBM Power8 ISA specification with the intention to be used only for non-shared writes. We propose to use the rollback-only hardware transaction for all of the writes, shared or non-shared, and in this way to hide all of the write modifications of the lock fallback path till its successful commit. The hiding process preserves opacity of concurrent hardware transactions, and allows read-only hardware transactions to commit even when there is a concurrent fallback execution.

Currently, IBM Power8 processors are unavailable for public use, and therefore we implement a software-based simulation of the RH-LE algorithm. It shows that the new lock elision scheme has the potential to be almost 2 times faster than other lock elision techniques.

1. Introduction

In a ground breaking paper, Rajwar and Goodman [9] proposed the idea of *hardware lock-elision* (HLE). Their scheme uses hardware transactions to execute lock-based critical sections concurrently, and ensures progress by reverting back to the original sequential execution when some hardware transaction fails. Recent Intel Haswell processor introduces on-chip implementation of hardware transactional memory (HTM) that also includes the HLE technique, and promises to deliver performance improvements for lock-based applications that are able to commit most of their transactions in hardware. The main challenge in achieving this goal is to overcome the many limitations that hardware poses on HTM transactions, like cache capacity limitations, hardware or software interrupts, unsupported or protected instructions, and more, that may render the HTM mechanism inefficient.

A recent work by Afek et al. [13] and Calciu et al. [6] proposes software-improved lock elision schemes, based on the ideas of *lock-removal* [10], that provide a better concurrency by sacrificing safety (or opacity). The original HLE scheme subscribes hardware transactions to the global lock when they start. The subscription includes a speculative read of the global lock and a verification that it is free. In effect, when HLE fallbacks and grabs the global lock, this triggers an abort of all current hardware transactions and ensures a safe execution of the fallback. In *lock-removal* schemes, hardware transactions subscribe to the global lock when they commit. As a result, hardware transactions execute concurrently with the lock fallback path, but cannot successfully commit. This additional concurrency in beneficial, but comes on expense of safe execution. Now, hardware transactions may see inconsistent memory states and generate illegal instructions, corrupt memory regions, jump to random memory locations and so on. For example, a hardware transaction may mistakenly write to the global lock, and successfully pass the global lock check on its HTM commit (where it should abort). Protecting hardware transactions from those unsafe side-effects is still an open research problem.

We propose a novel *reduced hardware lock elision* algorithm (RH-LE). It provides an opaque concurrency between hardware transactions and the lock fallback path. The core idea behind the RH-LE approach is to execute the lock fallback path as a *rollback-only hardware transaction*, that tracks only the memory writes of the transaction. This special hardware transaction is already introduced in the IBM Power8 ISA specification with the intention to be used only for non-shared writes. We propose to use the rollback-only hardware transaction for all of the writes, shared or non-shared, and in this way to hide all of the write modifications of the lock fallback path till its successful commit. The hiding process preserves opacity of concurrent hardware transactions, and allows read-only hardware transactions to commit even when there is a concurrent fallback execution.

Successful execution of the lock fallback path as rollback-only hardware transaction depends on the amount of writes. If the writes cannot fit into the hardware capacity, then the lock fallback path grabs the global lock, and aborts all concurrent hardware transactions, in a similar way to the all-abort HLE fallback. The point of RH-LE design is to exploit the fact that in many applications many operations are mostly composed of reads, and small amount of writes. It is rare to see operations that perform substantial large modifications to a shared data-structure, and the idea is to apply the expensive all-abort HLE fallback only when we have such rare large modifications.

We implement a software-hardware simulation of RH-LE algorithm based on Haswell HTM mechanism and automatic segmentation [3]. The automatic segmentation splits the lock fallback path to multiple segments, each executing as a short hardware transaction, and hides the write modifications of the segments, by passing the write information "internally" from one hardware transaction to the next. The segmentation avoids tracking an increasingly large read-set in the HTM, while the writes passing preserves whole write-set tracking during segments execution. During the execution, each segment performs in-place direct memory reads and writes, while maintaining a private undo-redo-log of the write modifications. The

purpose of the undo-redo-log is to enable to undo the writes at the split point of a transaction, and then re-does them when a new short transaction is started. To combine the two HTM transactional segments, the RH-LE simulation (1) undoes the write modifications of the first segment by using the undo-log data, (2) commits the HTM segment, (3) starts a new HTM segment, and (4) uses the redo-log to redo the write modifications back. In this way, the RH-LE simulation transfers the write modifications from one segment to the next, in a way that hides them from other concurrent operations within the HTM environment. The actual in-place writes of the segments are thus deferred to the final segment, and the reads of the segments execute without any instrumentation and pose minimal overhead for the simulation.

The automatic segmentation splits the lock fallback path into the longest possibly short segments. With higher abort rates the segment length is shortened and vice versa. In this way, the RH-LE simulation minimizes the HTM read speculation effects, and allows the segments to execute successfully as long as they use HTM supported instructions and have a write-set size that fits into HTM capacity limits. If a segment have become very small, and still constantly fails to commit in HTM, then it represents a case for which the corresponding rollback-only hardware transactions would also fail, and therefore the segment fallbacks to non-speculative execution. It acquires the global lock, which aborts all current hardware transactions, and resumes the execution from the point of the failed segment.

Empirical testing on state-of-the-art 8-way Intel Haswell chip with 4 cores, shows that RH-LE has the potential to deliver almost 2 times faster performance than other lock elision techniques. It is important to note that we try to approximate the performance of real rollback-only hardware transaction. One should take the simulation with a grain of salt, in particular because currently IBM Power8 is unavailable for public access, and it is hard to predict all possible hardware interactions and optimizations that may effect rollback-only hardware transactions performance. Still, we believe that our simulation correctly estimates and provides a good lower bound on the performance of our RH-LE scheme, at least qualitatively, and shows that there is a benefit in using rollback-only hardware transactions for tracking shared and non-shared writes. In particular, the rollback-only hardware transaction has the write-set in HTM tracking for the whole execution of the lock fallback path. In the simulation, this is true as long as the segments execute in HTM, but temporary becomes false in the short split gap between the commit of the current segment HTM and the start of the next one. The length of this temporary gap is the time it takes to restore the write-set back when the next segment starts, and usually it is very short or zero for read-only HTMs. In addition, the simulation introduces additional overheads due to the automatic segmentation, and uses standard HTM transactions to implement the segment executions. In effect, the segments speculate also on the reads, while the automatic segmentation works to minimize the effects of this speculation. All this factors only introduce additional overheads that would not be present in a real rollback-only hardware transaction, and therefore our results should represent a lower bound on the possible performance that can be achieved.

2. Other Related Work

Roy, Hand, and Harris [1] proposed an all software implementation of HLE, in which transactions are executed speculatively in software, and when they fail, or if they cannot be executed due to system calls, the system defaults to the original lock. In order to synchronize correctly and get privatization, their system uses Safe(..) instrumentation for objects and a special signaling mechanism between the threads that is implemented inside the kernel. In short, speculative lock-elision is complex and requires OS patches or hardware support because one has to deal with the possible failure of the speculative calls.

Afek et al. [2] propose a more effective all-software version of HLE for read-write locks. Their technique is based on a fully pessimistic STM, in which every software transaction executes only once and never aborts.

A different approach to ensure HTM progress is *hybrid transactional memory* (HyTM). The idea of HyTM is to allow concurrent execution of hardware and software transactions. In this way, hardware transactions that fail can fallback to execute as software transactions, and avoid the hardware limitations. A HyTM approach has been studied extensively in the literature [4, 7, 8, 11], and its key challenge is to design efficient coordination protocol for the concurrent hardware and software transactions.

3. RH-LE Simulation

Algorithm 1 RH-LE : fast-path

```
 1: function FAST_PATH_START(ctx)
 2:     ctx.is_writer ← 0
 3:     HTM_Start()
 4:     if HTM fails then
            ▷ Outside HTM: handle abort
 5:         Fallback to the lock-fallback-path
 6:     end if
            ▷ Inside HTM
            ▷ Abort if there is non-speculative execution
 7:     if global_lock ≠ 0 then
 8:         HTM_Abort()
 9:     end if
10: end function
11:
12: function FAST_PATH_READ(ctx, addr)
            ▷ do nothing - uninstrumented
13:     return load(addr)
14: end function
15:
16: function FAST_PATH_WRITE(ctx, addr, new_val)
            ▷ update writer indication for the commit
17:     ctx.is_writer ← 1
18:     store(addr, new_val)
19: end function
20:
21: function FAST_PATH_COMMIT(ctx, addr, value)
22:     if ctx.is_writer = 1 then
            ▷ Fallback is in progress: only read-only HTMs can commit.
23:         if fallback_lock ≠ 0 then
24:             HTM_Abort()
25:         end if
26:     end if
27:     HTM_Commit()
28: end function
```

We implement a software-hardware simulation of RH-LE algorithm based on Haswell HTM mechanism and automatic segmentation [3]. Each segment executes as a short hardware transaction, and hides the write modifications of the segments, by passing the write information from one hardware transaction to the next. The goal of HTM and segmentation combination is to avoid tracking an increasingly large read-set, while preserving the writes tracking in the HTM.

The RH-LE algorithm first tries to execute transactions in the pure hardware *fast-path*, and if some transaction fails, then it fallbacks to the *lock-fallback-path* that executes as a series of segments. If a segment fails to commit in HTM, then it fallbacks to non-speculative execution, by acquiring the global lock and resuming the execution from the point of the failed segment.

Algorithm 2 RH-LE : lock-fallback-path

```
 1: function FIRST_SEGMENT_START(ctx)
 2:     LockAcquire(fallback_lock)
 3:     ctx.is_non_speculative ← 0
 4:     Segment_Start(ctx)
 5: end function
 6:
 7: function SEGMENT_READ(ctx, addr)
        ▷ do nothing - uninstrumented
 8:     return load(addr)
 9: end function
10:
11: function SEGMENT_WRITE(ctx, addr, new_val)
        ▷ log the current value and write a new one
12:     cur_val ← load(addr)
13:     add (addr, cur_val, new_val) to ctx.write_set
14:     store(addr, new_val)
15: end function
16:
17: function SEGMENT_SPLIT(ctx, addr, value)
        ▷ do nothing for non-speculative run
18:     if ctx.is_non_speculative = 1 then
19:         return
20:     end if
        ▷ Undo write modifications
21:     for addr, cur_val ∈ ctx.write_set do
22:         store(addr, cur_val)
23:     end for
24:     HTM_Commit()
25:     Segment_Start(ctx)
        ▷ Redo write modifications
26:     for addr, new_val ∈ ctx.write_set do
27:         store(addr, new_val)
28:     end for
29: end function
30:
31: function SEGMENT_START(ctx)
32:     HTM_Start()
33:     if HTM fails then
            ▷ Outside HTM: handle abort
34:         ... some segment retry policy ...
35:         if the policy decides not to retry then
                ▷ Start non-speculative execution: aborts everyone else
36:             ctx.is_non_speculative ← 1
37:             LockAcquire(global_lock)
38:             return
39:         end if
40:     end if
        ▷ Inside HTM
41: end function
42:
43: function FINAL_SEGMENT_COMMIT(ctx, addr, value)
        ▷ non-speculative run only releases the locks.
44:     if ctx.is_non_speculative then
45:         LockRelease(global_lock)
46:         ctx.is_non_speculative ← 0
47:         LockRelease(fallback_lock)
48:         return
49:     else
50:         HTM_Commit()
51:         LockRelease(fallback_lock)
52:     end if
53: end function
```

Algorithm 1 shows the implementation of the fast-path. The fast-path executes a pure hardware transaction. On start, it initiates a hardware transaction and verifies that the global lock is free (=0), in order to check that there is no concurrent non-speculative execution. The fast-path dynamically detects if it executes a write transaction, because it is possible to always commit read-only hardware transactions and improve the concurrency of lock elision scheme. The *is_writer* local variable is initialized to 0 on start, and is set to 1 by the first write operation. Then, the commit uses this information and aborts only write transactions that try to commit concurrently with the fallback execution.

Algorithm 2 shows the implementation of the lock fallback path, that executes as a series of segments. On start, the lock fallback path acquires the *fallback_lock*, and then initiates the segmentation process. Every segment starts a hardware transaction, and performs direct memory reads and writes while logging the values of the write locations. The split procedure "transfers" the speculative memory modifications of one segment to the next, without exposing them to other transactions. Immediately before the segment commits, the split procedure restores the write locations to their current (or original) values, commits the current HTM segment, starts the next HTM segment, and restores the write locations to their new values. Now, the new segment can execute with the memory modifications of the previous segments. The final commit only needs to execute the HTM commit instruction and release the fallback lock. If a segment fails to commit in HTM, even when it becomes very short, due to excessive speculative writes or unsupported instructions, then it fallbacks to non-speculative execution. In this case, it acquires the global lock (which aborts all current transactions), and resumes the execution from the point of the failed segment. On commit, it releases the global lock and the fallback lock.

3.1 Automatic Segmentation

Each segment length dynamically adapts to the contention level, which is deduced from the HTM abort ratios. With higher abort rate the segment length is shorten and vice versa.

The automatic segmentation scheme of the RH-LE algorithm is based on [3]. The algorithm controls the segment lengths on the level of basic blocks (a sequence of instructions till next jump instruction). It injects a call to split checkpoint function at the end of every basic block in the code, and this allows to count the number of basic blocks that the program has gone through. When it reaches a predefined threshold, it executes the segment split operation, that commits the current hardware transaction and starts a new one.

The algorithm finds out how many basic blocks every segment can execute by first trying to execute a large number of basic blocks. If this fails, then it tries a smaller number, and so on. When it reaches the number for which the segment succeeds, it saves this number and use it for the next run of this segment. If the segment succeeds "too much" without any HTM aborts, then it increases its saved length. Currently we implement a naive simple protocol, that simply decrements the segment length by 1, every time it gets 5 consecutive capacity aborts, and increments its length by 1 every time it gets 5 consecutive successful commits (without any aborts). Improving the automatic segmentation technique is an intriguing topic for future work.

4. RH-LE Performance

We evaluated the performance of the RH-LE software-hardware simulation on an 8-way Intel Haswell chip with 4 cores, each multiplexing 2 hardware threads (HyperThreading). We present results for a red-black tree data-structure, and benchmark various lock elision schemes. As a reference, we also provide the results

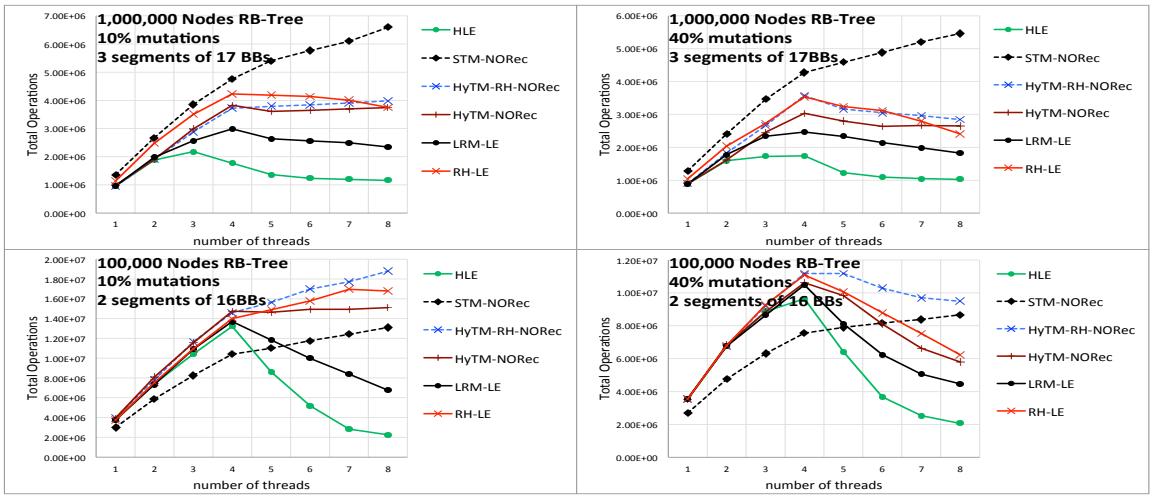

Figure 1. Throughput results for the red-black benchmarks.

for HybridTMs and STMs. It is important to note that HybridTMs and STMs are not lock elision techniques, and cannot be used for the purpose of lock elision as is.

STM-NORec *The original NORec STM:* Transactions execute as an all-software NORec STM [5]. It defers the writes to the commit-phase and uses a global clock for coordination.

HyTM-NORec *The original Hybrid Norec:* The algorithm of Dalessandro et. al [4]. The hardware fast-path subscribes to the global clock on start (reads it and verifies that it is free) and increments it on the commit. The software slow-path executes the all-software NOrec STM. In this hybrid, hardware transactions abort when the NORec STM locks the global clock on its commit-phase, immediately before the write-back.

HyTM-RH-NORec *Reduced hardware NORec [8]:* The hardware fast-path only updates the global clock at the end of the hardware transaction. The fallback is a "mixed" hardware-software slow-path that executes the transaction body in pure software, and the transaction commit write-back by using a small hardware transaction. If the "mixed" slow-path fails, then it fallbacks to the original hybrid NORec approach. In this scheme, hardware transactions abort only on real conflicts, as long the "mixed" path succeeds. When the "mixed" path fails, they abort in the same way as Hybrid NORec.

HLE *Hardware lock elision:* Transactions try to execute the operations as pure hardware transactions using the Intel Haswell RTM mechanism [12]. If some transactions fails, then it grabs a global lock, which aborts all current hardware transactions, and executes the operation in a mutually exclusive manner.

LRM-LE *Lock removal based lock elision [6, 13]:* Transactions try to execute in HTM, and when fail, grab the global lock without aborting the currently executing hardware transactions. To make this work, the hardware transactions subscribe to the global lock (speculatively read it and verify it is free) before the HTM commit, instead of doing this immediately after the HTM start. The main problem with this approach is safety. Hardware transactions may execute illegal instructions, corrupt memory and more. For example, a hardware transaction may mistakenly write to the global lock, and successfully pass the global lock check on its HTM commit (where it should abort).

RH-LE *Reduced hardware lock elision:* The RH-LE algorithm simulation as described in Section 3.

Fallback policy. In our testings we found out that it is worth retrying failed hardware transactions before making the software fallbacks. This is true for all of the schemes that use hardware transactions. As a result, our fallback policy for all of these schemes, retries hardware transaction 10 times before making the software fallback.

Execution. The benchmarks allow us to control the size of the data-structure, 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. In addition, for every graph we indicate the average number of splits (segments) per transaction, and the average length of a split (segment) in basic blocks (BBs). Recall that a basic block is a sequence of instructions till the next branch.

Results. Figure 1 top graphs show the results for a large RB-Tree with 1,000,000 nodes, the bottom graphs for a smaller tree with 100,000 nodes. The left graphs show low mutation ratios of 10%, and right high ratios of 40%.

First, we can see that all of the HTM-based techniques suffer significant penalties in the 4-8 threads range. This is due to the HyperThreading mechaism initiation after 4 threads, that generates excessive HTM capacity aborts. Next, we can see that the HLE scheme delivers the worst scalability. The LRM-LE sacrifices opacity (safety), and allows concurrency between the HTM transactions and the lock fallback path. As a result, it shows better scalability, and can be as twice faster than the HLE scheme. Looking at the RH-LE, we can see that on average it can be 2 times faster than the LRM-LE for low mutation ratios, and 1.5 times faster for high mutation ratios. This is a result of the RH-LE improved concurrency due to opacity, that allows HTM read-only transactions to commit even when there is a concurrent lock fallback execution. In LRM-LE none of the HTM transactions can commit with a concurrent lock fallback.

As a reference, we show the performance of the STM NORec and the Hybrid TM schemes (NORec and RH). We can see that the STM is better than the hardware-based schemes for the large RB-Tree. This is because the large tree generates many capacity aborts. Also, we can see that HybridTMs are similar to RH-LE, and the RH scheme is sometimes better.

References

[1] T. Harris A. Roy, S. Hand. A runtime system for software lock elision. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 261–274, New York, NY, USA, 2009. ACM.

[2] Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic software lock-elision. In Marcos K. Aguilera, editor, *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2012.

[3] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[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] Calciu Irina, Shpeisman Tatiana, Pokam Gilles, and Herlihy Maurice. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop*, 2014.

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

[8] Alexander Matveev and Nir Shavit. Reduced hardware norec: An opaque obstruction-free and privatizing hytm. In *Transact 2014 Workshop*, 2014.

[9] R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.

[10] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *10th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2003.

[11] 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.

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

[13] Afek Yehuda, Levy Amir, and Morrison Adam. Software-improved hardware lock elision. In *PODC 2014*, Paris, France, 2014. ACM Press.