Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory

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

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

ABSTRACT

For many years, the accepted wisdom has been that the key to adoption of best-effort hardware transactions is to guarantee progress by combining them with an all software slow-path, to be taken if the hardware transactions fail repeatedly. However, all known generally applicable hybrid transactional memory solutions suffer from a major drawback: the coordination with the software slow-path introduces an unacceptably high instrumentation overhead into the hardware transactions.

This paper overcomes the problem using a new approach which we call *reduced hardware* (RH) transactions. 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 almost all of the instrumentation from the common hardware fast-path, making it virtually as fast as a pure hardware transaction. Moreover, the "mostly software" slow-path is obstruction-free (no locks), allows execution of long transactions and protected instructions that may typically cause hardware transactions to fail, allows complete concurrency between hardware and software transactions, and uses the shorter hardware transactions only to commit.

Finally, we show how to easily default to a mode allowing an all-software slow-slow mode in case the "mostly software" slow-path fails to commit.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Concurrency, Synchronization, Multiprocessing, Threads

Keywords

Multicore Software, Hybrid Transactional Memory, Obstruction-freedom

SPAA'13, July 23–25, 2013, Montréal, Québec, Canada.

1. INTRODUCTION

IBM and Intel have recently announced hardware support for best-effort hardware transactional memory (HTM) in upcoming processors [16, 17]. 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 [7, 10] 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. This approach, which is the basis of all the generally applicable HyTM proposals, imposes severe overheads on the hardware transaction – the HyTM's frequently executed *fast-path*.

Riegel et al. [13] provide an excellent survey of HyTM algorithms to date, and the various proposals on how to reduce the instrumentation overheads in the hardware fast-path. There are three key proposed approaches, each with its own limitations.

The first is Phased TM [11], in which transactions are executed in phases, each of which is either all hardware or all software. Phased TM 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. Though this is a good approach for some workloads, in general it is not clear how to overcome frequent switches between phases.

The second approach, Hybrid Norec [5], is a hybrid version of the efficient Norec STM [6]. 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 revalidations 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, as with the original Norec STM, scalability is limited because the conflicts cannot be detected at a sufficiently low granularity.

The third approach, by Riegel et al. [13], effectively reduces the instrumentation overhead of hardware transactions in HyTM algorithms based on both the LSA [15] and Norec [6] 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 [4] but are not supported in the best-effort HTMs that IBM and Intel are bringing to the marketplace.

1.1 The Big "If" in Hybrid Transactional Memory

What will the cost of the instrumentation of the hardware transactions in HyTM be in upcoming hardware?

Unfortunately, access to such processors is not available yet. Thus, in an attempt to answer the question, we conducted a number of "emulation" benchmarks on todays Intel processors. We emulated an idealized HTM execution by running virtual transactions that execute the same sets of instructions but modify the shared data structure in a way that does not affect the logic of the concurrently executing transactions.

For example, to implement an update method of a red-black tree as a transaction, the method searches for a node with a key chosen from a given distribution and writes a dummy value to this node. The dummy value is read by concurrent readers, but it is not logically used for decisions during the tree traversals. In this way, the transactions can run correctly and not crash, and still pay the cache coherence traffic that is associated with the writes. To make this emulation more precise, we also introduce an abort ratio similar to that known from past STM-based benchmarks.

What we found in our emulated benchmarks is that in a traditional HyTM implementation, as opposed to the ideal HTM, the overhead in hardware transactions of loading and conditionally branching on the shared STM meta data, is excessively high. This conforms with findings in prior work that stated that the overhead of traditional HyTM implementations is high [5, 13]. As can be seen from the graph of the red-black tree benchmark in Figure 1, with meta data loading, testing and branching, the performance of an HTM goes from 5-6x faster to being only 2x faster than a TL2 STM [8]. In other words, adding the meta data loads and "if" branches to the HTM transactions eliminates much of the benefits of running in hardware.

Obviously, the results of our benchmarks should be taken with a large grain of salt, in particular because processors with HTM support (such as Intel's Haswell) will most likely have caching and speculation paths that differ from those we used, and yet, we believe our emulations have a chance of proving true, at least qualitatively.

The conclusion to be taken from this data – consistent with the arguments made by others (See [5, 13]) – is that HyTM makes sense only if we can remove the meta-data accesses and conditional branches from as much of the HTM code as possible. As we noted above, existing algorithms that provide this on standard architectures, despite their good performance on some benchmarks, suffer from scalability issues or have overall limited applicability. Thus, the question is if one can devise a broadly applicable HyTM algorithm that will have reduced conditional branching and meta data access along the hardware fast-path, and will thus be scalable.

1.2 Reduced Hardware Transactions

This paper presents a new broadly applicable approach: *reduced hardware* (RH) transactions. RH transactions allow an extensive reduction of the instrumentation overhead of the hardware fast-path transactions on all upcoming architectures, without impairing concurrency among hardware and software transactions, and with various other scalability benefits. We present the RH1 reduced hardware HyTM protocol in the body of this paper, and provide the RH2 protocol in the appendix, to be viewed at the committee's discretion.

As we noted earlier, all known HyTMs have the best-effort hardware fast-path default to a *purely software* slow-path if they fail repeatedly due to hardware constraints (These constraints can be the result of transactions that are simply too long, or because they call protected or OS related instructions that are simply not allowed in HTM). In an RH transaction protocol, instead of having the hardware fast-path default to a pure software slow-path, it defaults to a "mixed" path that consists mostly of software but also includes a shorter best-effort hardware transaction during the commit. Though others have proposed STMs that have hardware elements [14, 2], unlike them, the goal here is not to improve the slow-path software's performance. Rather, by introducing this shorter hardware transaction into the software slow-path, we are able to remove most of the meta-data accesses and conditional branches from the common hardware fast-path, making it virtually as fast as pure hardware.

Here, in a nutshell, is how the RH1 HyTM protocol works. (We assume familiarity with global-time based STM algorithms such as TL2 [8] or LSA [15]). The RH1 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, RH1 runs a global-time based STM transaction (such as TL2 [8] or TinySTM [12]) in which each memory location has an associated time-stamp that will be updated when written. The transaction body is executed purely in software, collecting read and write sets, and postponing the actual data writes to the commit phase. The key new element in RH1, is that the commit phase is executed in a single speculative hardware transaction: the read and write locations are validated based on an earlier read of the global clock, and if successful, the actual writes are applied to memory together with an updating of the time-stamps based on a new read of the global clock. Unlike TL2 or TinySTM, there are no locks (only time-stamps), and the transaction is obstruction-free.

Perhaps surprisingly, this change in the slow-path allows us to completely remove all of the testing and branching in the hardware fast-path for both reads and writes. The hardware fast-path transaction needs only to read the global clock (which is updated only rarely by concurrent slow-path transactions that happen to fail) and use it to update the time-stamps of locations it writes. 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. The writing of the new time-

stamps on the fast path makes sure to fail inconsistent slow-path transactions.

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. Moreover, the RH1 slow-path hardware transaction simply validates the time-stamps of each location in the read-set (not the data itself), and writes each location in the write-set. The number of locations it accesses is thus linear in the size of the meta-data accessed, which is typically much smaller than the number of data locations accessed. For example, for the red-black tree, the read-set time-stamp meta-data is 1/4 the size of the locations actually read, and we would thus expect the mixed slow-path to accommodate transactions that are 4x longer than the all-hardware fast-path.

If some slow-path transaction still fails to complete, we show that it is easy to fall back briefly to a slow-slow-path mode, in which concurrent hardware and software both run a more complex protocol that allows software TL2 style transactions. Alternately, once could default first to a mode of running an alternative RH2 protocol which has a shorter hardware transaction on the slow-path rather than a full STM, and manages to avoid instrumenting reads in the fast-path hardware transactions. We note that in our slow-path and slow-slow-path we have not added an implicit privatization mechanism (see for example [1]) which would be necessary in unmanaged environments, and leave this for future work.

In summary, the RH1 protocol allows virtually uninstrumented hardware transactions and mixed hardware-software slow-path transactions that (1) execute the transaction body fully in software (2), significantly extend the length of the transaction, (3) run concurrently with hardware fast-path transactions, and (4) provide obstruction-free progress guarantees. Our emulation results suggest that the RH1 protocol performs as well as pure HTM transactions on a variety of benchmarks including red-black trees, hash-tables, and linked lists, spanning the parallelism and transaction-length range.

2. REDUCED HARDWARE TRANSACTIONS

We begin with an overview of our obstruction-free RH1 hybrid transactional memory protocol.

2.1 RH1 Algorithm Overview

Our algorithm is a variation of the TL2 or LSA-style STM algorithms [8, 15], and we will assume the reader's familiarity with these algorithms. In a similar way to TL2, the shared memory range is divided into logical stripes (partitions), each with an associated metadata entry. The software and hardware transactions communicate by inspecting and updating the metadata entries for the memory locations they read and write. In our hybrid TM every transaction has a pure hardware fast-path implementation, a mostly software slow-path implementation that uses a shorter hardware transaction for its commit protocol, and an all software slow-slow-path in case both of the others fail repeatedly.

Transactions must maintain a consistent snapshot of the locations read during their execution. To this end a global version clock is introduced, used by both fast and slow-path transactions to update local version time-stamps upon writing. Slow-path transactions identify conflicts by reading this shared global version clock on start, and comparing it against the stripe version for every location read. If a location is overwritten after a transaction started, then its timestamp will reflect this causing the transaction to abort, and otherwise the locations read form a consistent snapshot. In TL2 the transaction body is executed collecting a read set and a write set, then validating the time-stamps of all the locations in these sets, and writing the new values with increased time stamps. The TL2 software commit is executed after taking locks on all locations to be updated, but one of the advantages of the scheme here is that we will not need them.

Now, to achieve our goal of making the fast-path hardware transactions execute at hardware speed, we make two observations about a TL2 style Hybrid protocol executed in both hardware and software modes.

The first observation is that if we execute all the commit-time writes of the slow-path in a single hardware transaction, then in order to be consistent the fast-path hardware transaction does not need to do any testing of locations it accesses: it will either see all of them or none of them, since if it sees only part of them then the other transaction must have written concurrently and the hardware transaction will have a cache invalidation and abort.

The second observation is that if we have the hardware transaction update the time-stamps of the locations it writes using the latest value of the global version clock, then it will cause any concurrent software transaction that reads these locations to fail its commit time validation of the timestamps of its read and write sets.

There is one little caveat to this simple approach. The hardware transaction might manage to slip in the middle of the commit and write immediately after a successful validation and before all the updated writes are executed atomically in hardware. Traditionally, as in TL2 or TinySTM, this is prevented by holding locks on the locations to be written. In RH1 we do not wish to use locks since they would have to be updated also in the hardware transaction, introducing an overhead. Instead, the solution is to have the validation and the write-back of the write-set values be part of one hardware transaction. With this change, we are guaranteed that the slow-path is also consistent. (In the appendix we show the RH2 protocol that uses locks, requires only the writes of data to be executed in a single hardware transaction, but introduces the added overhead into the hardware path in order to update the locks.).

2.2 The RH1 Algorithm Details

The global *stripe_version_array* holds the stripe versions (time-stamps). Each thread is associated with a thread local context that includes; *tx_version*, the global version counter value read on transaction start, *read_set*, a buffer of the locations read, and a *write_set*, a buffer of the locations written. All of the versions are 64bit unsigned integers, initialized to 0, and the *read_set* with the *write_set* can be any list implementation.

The global version counter is manipulated by the *GVRead()* and *GVNext()* methods, for reading and "advancing" it, and we use the GV6 [8, 3] implementation that does not modify the global counter on *GVNext()* calls, but only on transactional aborts. This design choice avoids unnecessary aborts

Algorithm 1 RH1 fast-path transaction implementation

```
1: function RH1_FastPath_start(ctx)
2:    HTM_Start()
3:    ctx.next_ver ← GVNext()
4: end function
5:
6: function RH1_FastPath_write(ctx, addr, value)
       ▷ update write location version
7:    s_index ← get_stripe_index(addr)
8:    stripe_version_array[s_index] ← ctx.next_ver
       ▷ write value to memory
9:    store(addr, value)
10: end function
11:
12: function RH1_FastPath_read(ctx, addr)
       ▷ no instrumentation - simply read the location
13:   return load(addr)
14: end function
15:
16: function RH1_FastPath_commit(ctx)
17:   HTM_Commit()
18: end function
```

Algorithm 2 RH1 slow-path transaction implementation

```
1: function RH1_SlowPath_start(ctx)
2:    ctx.tx_version ← GVRead()
3: end function
4:
5: function RH1_SlowPath_write(ctx, addr, value)
       ▷ add to write-set
6:    ctx.write_set ← ctx.write_set ∪ {addr, value}
7: end function
8:
9: function RH1_SlowPath_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:   ctx.read_set ← ctx.read_set ∪ {addr}
       ▷ try to read the memory location
14:   s_index ← get_stripe_index(addr)
15:   ver_before ← stripe_version_array[s_index]
16:   value ← load(addr)
17:   ver_after ← stripe_version_array[s_index]
18:   if ver_before ≤ ctx.tx_version and
      ver_before = ver_after then
19:      return value
20:   else
21:      stm_abort(ctx)
22:   end if
23: end function
24:
25: function RH1_SlowPath_commit(ctx)
       ▷ read-only transactions commit immediately
26:   if ctx.write_set is empty then
27:      return
28:   end if
       ▷ a single hardware transaction that performs read-set
      revalidation and write-back
29:   HTM_Start()
       ▷ read-set revalidation
30:   for addr ∈ ctx.read_set do
31:      s_index ← get_stripe_index(addr)
32:      version ← stripe_version_array[s_index]
33:      if version > ctx.tx_version then
34:         HTM_Abort(ctx)
35:      end if
36:   end for
       ▷ perform the actual writes and update the locations'
      versions
37:   next_ver ← GVNext()
38:   for addr, new_value ∈ ctx.write_set do
39:      s_index ← get_stripe_index(addr)
40:      stripe_version_array[s_index] ← next_ver
41:      store(addr, new_value)
42:   end for
43:   HTM_Commit()
44:   if the HTM failed then
45:      fallback to RH2
46:   end if
47: end function
```

of the hardware transactions that call for *GVNext()* (speculate on the global clock), in order to install it to the write locations.

Algorithm 1 shows the implementation of the RH1 fast-path transaction. The fast-path starts by initiating a hardware transaction (line 2). It performs the reads without any instrumentation (line 13), and the writes with minimal instrumentation that only updates write location's version on every write (lines 6 - 8). On commit, it simply performs the hardware transaction commit instruction (line 17).

Algorithm 2 shows the implementation of the RH1 slow-path. The slow-path starts by reading the global version to its local *tx_version* variable (line 2). During the execution, the writes are deferred to the commit by buffering them to a local write-set (line 6), and scanning this write-set on every read operation (lines 10-11). If the read location is not found in the local write-set, then it is read directly from the memory, followed by a consistency check (lines 14-18). This check verifies that the read location has not been overwritten since the transaction has started, based on the following invariant: If the read location has been already updated from the time the current transaction started, then the location's version must be greater than the transaction's version, *tx_version*. The fast-path and slow-path commits ensure this invariant. Finally, the slow-path commit executes a single hardware transaction that first performs the read-set revalidation, and then the write-back, that includes making the actual memory updates and installing of the next global version to the stripe versions of the write locations (lines 29 - 42).

2.3 RH1 Algorithm Limitations - Fallback to RH2 and the all-software slow-slow-path

The RH1 slow-path commit executes a single hardware transaction that performs the read-set revalidation and the write-back. This hardware transaction may fail for various reasons. In the common-case, the failure reason will be contention, and some kind of contention management mechanism can be applied to handle the transactional retries. In more rare situations, the hardware transaction may fail due to some hardware limitation. Note, that this hardware

transaction accesses a predefined memory range (the metadata range), and it performs only simple memory reads and writes. Therefore, on Intel architectures with RTM [17], the most likely reason for a constant failure of this transaction is a capacity overflow of the hardware reads buffer. In other words, the transaction metadata cannot fit in the L1 cache of the processor. To handle these cases, the algorithm performs fallback to RH2 that we describe in Appendix A and Appendix B.

RH2 reduces the HTM requirements of the slow-path transactions by performing only the commit-time write-back in a single hardware transaction (not including the read-set revalidation). The core idea is to introduce locks to the fast-path and the slow-path, and force the slow-path "expose" its read-set for the duration of the slow-path commit.

Still, one might worry about the progress guarantees of RH2, because the slow-path commit-time hardware transaction that performs the write-back atomically may fail. This would mean that the transaction's write-set cannot be accommodated inside the L1 cache of the processor, which is unlikely for real-world transactions. We show that in any case RH2 can easily fallback to a fully pure software slow-path in which it performs an all software commit and the fast-path transactions inspect the metadata for every read and write, in a similar way to the standard hybrid TMs. The switch to fully software RH2 slow-path aborts the current RH2 fast-path transactions and restarts them in the RH2 *fast-path-slow-read* mode. We call this special mode the *all software slow-slow-path*.

3. PERFORMANCE EVALUATION

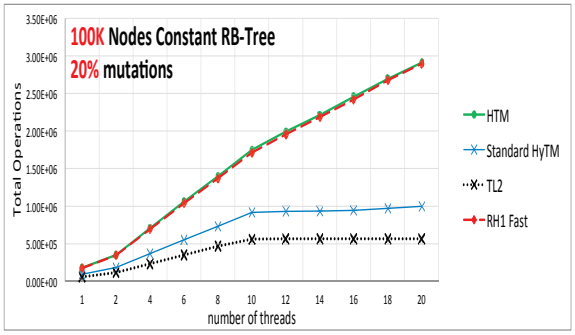

Figure 1: The graphs show the throughput of 100K sized Red-Black Tree for 20% writes. In this test we can see that the standard Hybrid TMs eliminate the benefit that HTMs can achieve, because they instrument the reads and writes of the hardware transactions. In contrast, *RH1* preserves the HTMs benefit by avoiding hardware reads instrumentation.

We evaluate our hybrid TM by constructing a set of special benchmarks that can be executed on current multicore processors, that is, without the (yet unavailable) HTM support. Our results should thus be taken with a grain of salt, and if you will, skeptic minds should treat our quantitative results as being mostly qualitative.

Our idea is to emulate an HTM transaction execution by running its logic and its reads and writes using plain loads and stores. There is no speculation, and the cache performance is obviously not the same as with an HTM mechanism, but we believe that the transaction with plain reads and writes is close to being a lower-bound on the performance of a real HTM system; we would be surprised if an all-hardware HTM, with its added functionality, can perform better.

The problem with executing non-instrumented transactions is that they cannot detect concurrent conflicts and maintain a consistent snapshot of the locations read. As a result, the non-instrumented transactions may crash and get into deadlocks. To avoid this problem, for every benchmark, we constrain the set of possible executions to the ones that will work correctly, and report the performance results for these specific executions. We try to make these executions as realistic as possible by emulating the expected abort ratio for every number of threads.

3.1 Red-Black Tree Emulation Overview

Our red-black tree implementation, the *Constant Red-Black Tree*, must allow only executions that are correct with non-instrumented transactions that simulate the HTM. We populate the RB-Tree with 100K nodes, and execute concurrent operations that do not modify the structure of the tree. Update operations only modify dummy variables inside the tree's nodes, while the lookups traverse the nodes and read these dummy variables, paying the cache-coherence traffic for their fake updates.

More precisely, we expose a read-only and a write operation: *rb-lookup(key)*, and *rb-update(key, value)*. The *rb-lookup(key)* makes the usual tree traversal, looking for the node with the given key, and making 10 dummy shared reads per node visited. The *rb-update(key, value)* also executes the usual tree traversal to find the node with the given key, and then makes fake modifications. It writes a dummy value to the dummy variable in the node it found and its two children; it does not touch the pointers or the key value. To make the modifications mimic tree rotations, the operation makes the same fake modifications to triplets of nodes, going up from the node it found to the root. The number of nodes climbed up the tree is selected at random, so that getting to the upper levels and the root will happen with diminishing probability, as in a real tree implementation.

We estimate the expected abort ratio for a given execution by first executing with the usual TL2 STM implementation. Then, we force the same abort ratio for the hybrid execution by aborting HTM transactions when they arrive at the commit. Obviously the STM abort ratio is only an estimate of the HTM abort ratio. Real HTM may add more aborts because of the internal hardware implementation limitations, or may reduce the number of aborts because of the reduced transaction execution window (hardware transactions execute faster); making them less vulnerable to conflict. Therefore, the STM abort ratio is probably somewhere in the middle.

3.2 Red-Black Tree Emulation Execution

The benchmark first creates a 100K node red-black tree, and then spawns the threads that execute the *rb-lookup(key)* and *rb-update(key, value)* operations as transactions. We vary the number of threads and the write ratio (the percentage of update transactions).

We execute the benchmarks on Intel 20-way Xeon E7-4870

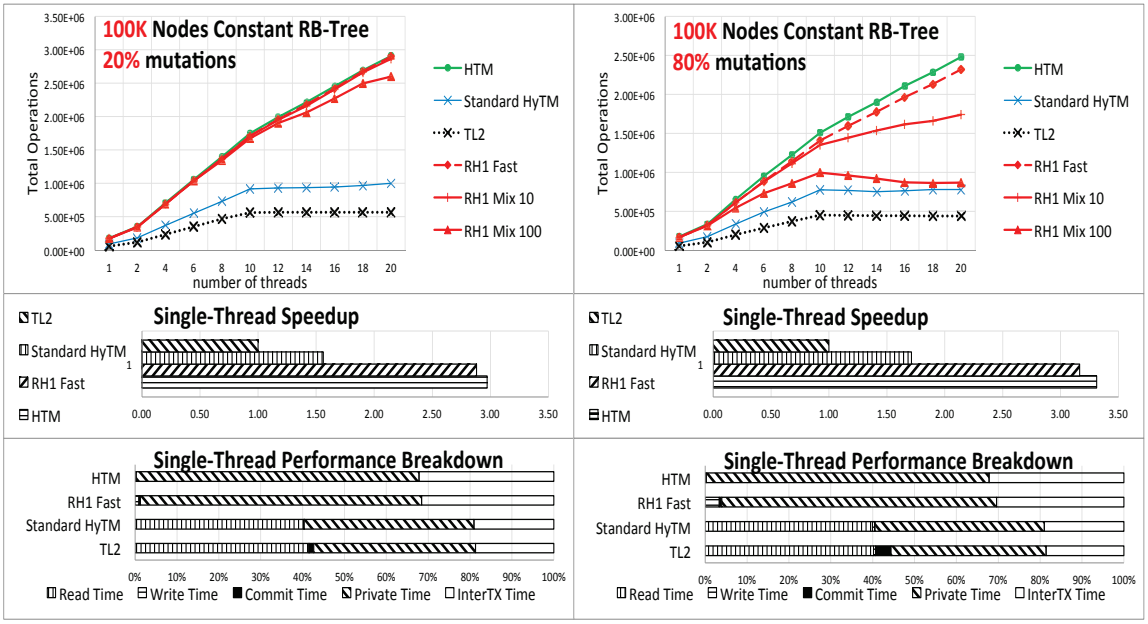

Figure 2: The top graphs show the throughput of 100K sized Red-Black Tree for varying number of write s; 20% and 80%. The middle and the bottom graphs show the single-thread speedup and performance breakdown.

chip with 10 2.40GHz cores, each multiplexing 2 hardware threads (HyperThreading). Each core has a private write-back L1 and L2 caches and the L3 cache is shared.

The algorithms we benchmark are:

HTM *Hardware Transactional Memory without any instrumentation:* all of the transactions are executed without instrumenting the reads and the writes. This represents the best performance that HTM can achieve.

Standard HyTM *The Standard Hybrid Transactional Memory*: This represents the best performance that can be achieved by current state-of-the-art hybrid TMs [13]. To make the hybrid as fast as possible, we execute only the hardware mode implementation, by executing and retrying transactions only in hardware, without any software fallback. We implement the hardware mode transaction with instrumented read and write operations, and make the commit immediate without any work. The hardware transaction reads and writes are minimally instrumented; each read and write accesses the STM metadata and creates a fake "if" condition check on its contents. The "if" condition does not change the execution logic; its only purpose is to show the resulting instrumentation overheads that occur for the standard hybrid TMs.

RH1 Mixed *Reduced Hardware Transactions 1:* Our new hybrid TM with hardware commit in the slow-path and uninstrumented hardware reads. This implementation uses both the all hardware fast-path and the mixed hardware-software slow-path.

RH1 Fast This is the RH1 fast-path only. All of the aborts are retried in hardware mode.

TL2 This is the usual TL2 STM implementation [8], that uses a GV6 global clock.

The standard hybrid TM algorithms instrument the read and write operations of the hardware transaction. In contrast, our new hybrid TM executes the reads with no instrumentation and the writes with an additional write. Therefore, our first benchmark goal is to measure the cost of adding instrumentation to the hardware operations. Figure 1 shows the penalties introduced by instrumenting the reads of the hardware transactions. Since, we are only interested in the hardware instrumentation overhead, this test is not using the RH1 slow-path mode, and retries the hardware transactions in fast-path mode only. The *TL2* and *HTM* graphs show the results for STM and HTM executions respectively. We can see that HTM performs 5-6x better than STM, and by adding instrumentation to the hardware reads in *Standard HyTM*, a dramatic performance penalty is introduced that makes HTM only 2x better than STM. In contrast, *RH1 Fast* with the non-instrumented hardware reads, executes approximately at the same speed as HTM, and preserves the 5x speedup of the HTM.

Figure 2 shows the performance of our *RH1 Mixed* that first tries the fast-path, and on abort, retries the transaction in the slow-path. *RH1 Fast*, *RH1 Mixed 10*, and *RH1 Mixed 100* mean that 0%, 10%, and 100% of the aborted transactions are retried in the slow-path mode respectively. We compare the different variants of the *RH1 Mixed* to the best case *Standard HyTM* that uses only a hardware mode for its aborted transactions. For 20% writes, the *RH1 Mixed* slow-path mode penalty is not significant, because the abort ratio is low (approximately 5%). But for the 80% writes case, where the abort ratio is high (approximately 40%), the software fallback introduces a significant penalty. De-

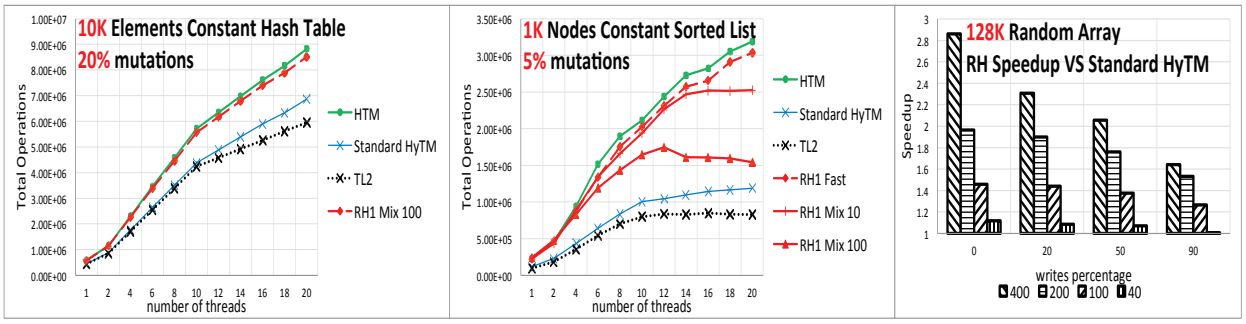

Figure 3: The results for the hash-table, sorted-list, and random-array benchmarks (from left to right).

spite this, *RH1 Mixed 100* performs slightly better than the same *Standard HyTM* for the mix of 80% writes. Recall, that *Standard HyTM* uses only the hardware mode for its execution and retries, but still is slightly slower than *RH1 Mixed 100*.

In order to understand the factors that affect the performance, we measured the single-thread speedups and the single-thread performance breakdowns of the different algorithms involved in Figure 2. The single-thread speedup is normalized to the *TL2* performance. They show the relative time used for the transactional read, write and commit operations, with the time used for the transaction's private code execution (local computations inside the transaction), and the time used for the inter-transactional code (code not inside a transaction). We can see that there is a correlation between the single-thread speedup and the algorithm's overall performance. Also, the single-thread breakdown shows that the read time is the dominating reason for the slowdown of the *Standard HyTM* relative to *RH1*.

3.3 Hash Table Emulation

We implemented a *Constant Hash Table* benchmark using an approach similar to the one we used in the *Constant Red-Black Tree*. The benchmark inserts 1000K distinct elements into the hash table. Then, the benchmark spawns the threads that execute the *hash_query(key)* and the *hash_update(key, val)* operations, where the number of updates is defined by the writes ratio parameter. The *hash_update* makes a query for the given key, and when the node with the key is found, it updates to the dummy variables inside this node, without touching the structure (pointers) of the hash table.

In Figure 3, the left graph shows the hash table results for 20% writes. In contrast to the red-black tree, the hash table transactions are much shorter and introduce less STM overhead relative to the non-transactional code. As a result, for the hash table, *HTM* improves the *TL2* STM performance by approximately 40%, where in the red-black tree it provides a 5x factor improvement. Additionally, the abort ratio is very small (approximately 3%) due to the highly distributed nature of hash table access. Still, the throughput of the *Standard HyTM* remains as low as that of the STM, while the *RH1 Mixed 100* preserves the HTM's advantage over STM.

3.4 Sorted List Emulation

The *Constant Sorted List* benchmark creates a 1K sorted list of distinct elements, and spawns the threads that execute the *list_search(key)* and the *list_update(key, val)* operations. The *list_update* searches for the node with the given key by a linear scan, and then, makes updates to the dummy variables inside this node, without touching the structure of the list.

In Figure 3, the middle graph shows the sorted list results for a mix that includes 5% writes. This benchmark represents a heavy-contended case for the STM. The transactions are long, introducing a significant STM overhead, and are prone to aborts because the *list_search(key)* operation makes a linear scan that implies in a shared list prefix by all currently executing transactions. The abort ratio is approximately 50% for 20 threads. We can see that the *HTM* is 4x faster than the *TL2* STM. As in the previous benchmarks, the *Standard HyTM* eliminates the HTM benefit and improves on the TL2 STM by only 50%, while the *RH1 Fast* preserves the HTM speedup. The introduction of the software mode aborts in *RH1 Mixed 10* and *RH1 Mixed 100* degrades the hybrid performance for high number of threads.

3.5 Random Array Emulation - Measuring the Effect of the Reads/Writes Ratio

The RH1 fast-path executes instrumented writes with non-instrumented reads. A common rule is that in real-world applications with transactions, the ratio of reads to writes is approximately 4 to 1 (20% writes). Still, since in the RH1 fast-path writes are not free, it is interesting to see the effect of increasing their number inside a transaction.

The *Random Array* is a shared array with 128K entries. Transactions simply access random array locations to read and write, without any special additional logic. This setup allows us to control the transaction length and the number of reads and writes inside a transaction. All of the executions have 20 threads.

In Figure 3, the right hand graph shows the speedup that *RH1 Fast* gains over *Standard HyTM* for different transaction lengths (400, 200, 100 and 40 shared accesses) and different write percentages inside a transaction (0%, 20%, 50% and 90% of writes). We can see that for long transactions the speedup decreases as the fraction of writes increases. For short transactions, the speedup change is less significant, because the overall effect of the small transactions on the benchmark is much less than that of the long ones. The interesting result is that even with mixes of 90%

writes, *RH1* with sufficiently long transactions provides a good speedup of 1.3-1.7x relative to the *Standard HyTM*. The reason is the different cache-coherence behavior of the two algorithms. RH1 does not read metadata on hardware reads, and only writes metadata on hardware writes. In contrast, *Standard HyTM* reads and writes the metadata on hardware reads and writes respectively. This introduces significantly more cache traffic between concurrently executing transactions, resulting in a performance degradation.

4. ACKNOWLEDGEMENTS

This work was supported in part by NSF grant CCF-1217921, ISF grant 1386/11, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

5. REFERENCES

[1] D. Dice A. Matveev and N. Shavit. Implicit privatization using private transactions. In *Transact 2010*, Paris, France, 2010.

[2] Hagit Attiya and Eshcar Hillel. A single-version stm that is multi-versioned permissive. *Theory Comput. Syst.*, 51(4):425–446, 2012.

[3] Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In *SIROCCO*, pages 131–140, 2008.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] D. Dice and N. Shavit. Tlrw: Return of the read-write lock. In *Transact 2009*, Raleigh, North Carolina, USA, 2009.

[10] 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.

[11] Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. In *In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/ TRANSACT2007PhTM.pdf*, 2007.

[12] C. Fetzer P. Felber and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[13] 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.

[14] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. Hardware acceleration of software transactional memory. Technical report, DEPT. OF COMPUTER SCIENCE, UNIV. OF ROCHESTER, 2006.

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

[16] 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.

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

APPENDIX

A. RH1 FALLBACK TO RH2

In this section we present the RH1 code modifications that implement the fallback to RH2.

RH1 uses a global *is_RH2_fallback* counter variable to perform the switch to the RH2 mode. The RH1 slow-path atomically increments this global counter before executing the fallback RH2 slow-path commit code, and decrements it on fallback finish. As a result, the *is_RH2_fallback* counter indicates the number of currently executing RH2 slow-path transactions, and the RH1 fast-path transactions can use this global counter to decide when to execute the RH2 fast-path transactions. Upon the first *is_RH2_fallback* increment, all currently executing RH1 fast-path transactions must abort and restart in RH2 fast-path mode. For this purpose, the RH1 fast-path monitors this global counter for

Algorithm 3 RH1 fast-path and slow-path modifications for switching to RH2

```
1: function RH1_FastPath_start(ctx)
2:   if is_RH2_fallback > 0 then
3:     RH2_FastPath_start(ctx)
4:     return
5:   end if
6:   HTM_Start()
       ▷ RH1 fast-path monitors the is_RH2_fallback global
   counter to be 0 for the duration of the hardware transaction
7:   if is_RH2_fallback > 0 then      ▷ speculative load of the
   global value
8:     HTM_Abort(ctx)
9:   end if
10: end function
11:
12: function RH1_SlowPath_commit(ctx)
       ▷ read-only transactions commit immediately
13:  if ctx.write_set is empty then
14:    return
15:  end if
       ▷ a single hardware transaction that performs read-set
   revalidation and write-back
16:  HTM_Start()
       ▷ read-set revalidation
17:  for addr ∈ ctx.read_set do
18:    s_index ← get_stripe_index(addr)
19:    version ← stripe_version_array[s_index]
20:    if version > ctx.tx_version then
21:      HTM_Abort(ctx)
22:    end if
23:  end for
       ▷ perform the actual writes and update the locations'
   versions
24:  next_ver ← GVNext()
25:  for addr, new_value ∈ ctx.write_set do
26:    s_index ← get_stripe_index(addr)
27:    stripe_version_array[s_index] ← next_ver
28:    store(addr, new_value)
29:  end for
30:  HTM_Commit()
31:  if the HTM failed then
32:    fetch_and_add(is_RH2_fallback)
33:    RH2_SlowPath_commit(ctx)
34:    fetch_and_dec(is_RH2_fallback)
35:  end if
36: end function
```

the duration of the transaction by speculatively reading this global counter and verifying its value is zero, immediately after the hardware transaction starts. In addition, before the hardware transaction starts, the RH1 fast-path checks this global counter to be greater than 0, and if so, then it executes the RH2 fast-path, else it runs the RH1 fast-path. Algorithm 3 presents the RH1 fast-path and slow-path modifications that support the switching to the RH2 algorithm.

B. RH2 ALGORITHM OVERVIEW

In this section we give a brief overview of the *RH2* hybrid protocol. Our main RH1 protocol falls back to RH2 upon persistent failure of the RH1 slow-path commit-time hardware transaction. RH2 reduces the HTM requirements of the slow-path transactions by performing only the commit-time write-back in a single hardware transaction (not including the read-set revalidation). The core idea is to introduce locks to the fast-path and the slow-path, and force the slow-path "expose" its read-set for the duration of its commit.

Still, one might worry about the progress guarantees of RH2, because the slow-path commit-time hardware transaction that performs the write-back may fail. This would mean that the transaction's write-set cannot be accommodated inside the L1 cache of the processor, yet as we show even in this case, RH2 can easily fallback to a fully pure software slow-path that performs the whole commit in the software, and the fast-path transactions inspect the metadata for every read and write, in a similar way to the standard hybrid TMs. The switch to full software RH2 slow-path aborts the current RH2 fast-path transactions and restarts them in the RH2 *fast-path-slow-read* mode. We call this special mode the *all software slow-slow-path*.

The main difference between RH1 and RH2 is the fact that RH2 uses locks for synchronization between the fast-path and the slow-path. The RH2 slow-path commit locks the write-set, revalidates the read-set, and then executes a small hardware transaction that performs the write-back. The RH2 fast-path writes inspect these locks, while the reads execute without any instrumentation. Now, since the RH2 slow-path is not executing the read-set revalidation inside a hardware transaction, a problematic scenario may occur between the fast-path and the slow-path as follows: a slow-path transaction arrives at the commit, locks its write-set and revalidates its read-set. Now, before the new values are actually written to the memory, a fast-path transaction starts, reads a location that is currently locked, and decides to overwrite a location inside the read-set of this slow-path transaction. Then, the fast-path transaction commits successfully, and the slow-path finalizes the commit using an atomic memory write-back. In this scenario, one of the transactions must abort, yet both commit successfully.

The problem is that the un-instrumented fast-path transaction reads cannot see that a location is currently being locked by a concurrent slow-path transaction. To overcome this race, during the slow-path commit, the transaction makes its read-set visible to the writes of the fast-path transaction. In this way, fast-path transactions cannot write to a read-set of a concurrently committing slow-path transaction.

The read-set visibility is implemented by adding a read mask for every memory stripe. The bits of the read mask are associated with threads: the transaction of thread K makes its read-set visible by setting the K-th bit of every read location's read mask. To set the K-th bit on and off, we use a non-blocking fetch-and-add synchronization primitive. In our implementation, we use a 64bit read mask to represent 64 active threads, and a fetch-and-add atomic primitive to turn the read mask's bits on and off. For larger thread numbers, additional read masks are required. See [9] for a discussion of the scalability of the mask array approach.

A fast-path hardware transaction collects the write-set, and on commit, detects if there is a concurrent slow-path transaction executing. If so, it inspects the read masks of the locations in the write-set before committing. It sums up the total of all mask bits and aborts the transaction if this sum is greater than zero, that is, one of the mask's bits was made non-zero by some concurrent slow-path transaction.

Usually, making an STM's reads visible results in poor

performance, since every STM read is augmented with a write to shared memory. In our implementation the read visibility works differently, because it is applied only during the commit phase of the software write transactions. Any other transactions, hardware or software read-only, are not performing this visibility procedure, and do not pay any additional overhead for their reads. Additionally, we use an efficient fetch-and-add synchronization primitive to update locations' read masks, instead of using a compare-and-swap (CAS) operation that can fail multiple times before turning on the required bit. As a result, our software transactions with a commit-time visible read-set have nearly the same performance as that of state-of-the-art STMs.

C. RH2 ALGORITHM DETAILS

Here we present the implementation details of RH2 hybrid protocol. Algorithm 4 and Algorithm 5 show the RH2 fast-path and slow-path. Algorithm 6 presents the fast-path-slow-read mode implementation for the pure software slow-path execution, and Algorithm 7 presents slow-path additional helper functions, that implement the locking and visibility mechanisms.

In a similar way to RH1, the memory range is divided into logical stripes (partitions), each with a stripe version and a stripe read mask. Additionally, a global version counter is used to coordinate the transactions, and each thread is associated with a thread local context that includes; *tx_version*, the global version read on transaction start, *read_set*, a buffer of the locations read, and a *write_set*, a buffer of the locations written. All of the versions are 64bit unsigned integers, initialized to 0, and the *read_set* and *write_set*s can be any list implementation.

The global version counter is manipulated by the *GVRead()* and *GVNext()* methods, for reading and "advancing" it, which can be implemented in different ways. We use the GV6 [3, 8] implementation that does not modify the global counter on *GVNext()* calls, but only on transactional aborts. This design choice avoids unnecessary aborts of the hardware transactions that call *GVNext()* (speculate on the global clock) in order to install it to the write locations.

The RH2 slow-path commit protocol is based on two basic mechanisms: (1) locking a location, and (2) making the location visible. The location's stripe version lowest order bit is reserved for locking. Transactions lock a location by setting the stripe version to the thread's lock value *ctx.thread_id * 2 + 1*: that turns on the lowest order bit and encodes the thread id into the lock. The visibility of a location is represented by its read mask. Every bit of the read mask is associated with some active thread, so a 64bit read mask can hold information for 64 threads. In our implementation the threads are assigned ids from 0 to 63, and these ids are used to "index" the read masks (more threads require more read masks per stripe). A thread with id K will turn on the K-th bit in the location's read mask to indicate that it's reading it, and will reset this bit to remove this indication. We use the fetch_and_add() synchronization primitive to turn on and off bits in read masks, instead of using a CAS operation that can fail multiple times before actually succeeding (implementation in Algorithm 7).

Algorithm 5 shows the implementation of the RH2 slow-path. The slow-path starts by reading the global version to its local *tx_version* variable (line 2). During the execution, the writes are deferred to the commit by buffering them to a local write-set (line 6), and scanning this write-set on every read operation (lines 10-11). If the read location is not found in the local write-set, then it is read directly from the memory, followed by a consistency check (lines 14-18). This check verifies that the read location has not been overwritten since the transaction has started, based on the following invariant: If the read location has been already updated from the time the current transaction started, then the location's version must be greater than the transaction's version, *tx_version*. The fast-path and slow-path commits ensure this invariant.

Upon RH2 slow-path commit, the write-set locations are locked and the read-set is made visible (lines 29 - 30). Locking is done by setting the location's stripe version to the thread's lock value *ctx.thread_id * 2 + 1*. This value turns on the lowest order bit, the one reserved for locking, and encodes the thread that locked the location. Location visibility is done by turning on the thread-associated bit in the location's read mask (shown in Algorithm 7). Then, the next global version is generated (line 44), and the read locations are revalidated (line 31), ensuring they have not been overwritten from the transaction's start. After a successful revalidation, the new values are written-back to the memory by using a hardware transaction (line 32 - 43). On a successful write-back, the write locations are unlocked, by updating their versions to the new next global version, and the read locations' visibility is removed, by turning off the thread-associated bit in every read location's read mask.

Now, if the RH2 slow-path commit-time small hardware transaction fails due to contention reasons, then it is retried again. Else, aborts all of the current fast-path transactions and restarts them in the fast-path-slow-read mode, and performs the slow-path write-back in pure software (lines 36 - 42). RH2 implements this switch through a global integer *is_all_software_slow_path* variable, that counts the number of slow-paths that currently execute the commit-time write-back in pure software. Current fast-path transactions monitor this global variable to be 0 during their execution (by speculatively loading it), and on its modification (by the slow-path) automatically abort. On fast-path start, the transactions check this global variable, and if its not zero, they switch to the fast-path-slow-read mode.

Algorithm 4 shows the implementation of the RH2 fast-path hardware transaction. The fast-path performs speculative reads and writes, where the writes are augmented with logging the addresses written (line 13) and the reads proceed as is, without any instrumentation. These reads cannot be inconsistent, because, as we said, the slow-path transactions perform the actual memory writes atomically.

Finally, the fast-path commit verifies that the read masks of the write locations are all 0 (lines 25 - 33), before initiating the HTM commit instruction. Additionally, the write locations are speculatively locked (lines 34 - 45), by verifying that they are not locked by others, and by writing the special thread lock-mask value to each one of them. Then the HTM commit instruction is executed, and upon success, the write locations are updated and locked atomically. Finally, it gets the next global version, and installs it to the write location (lines 48 - 52).

Algorithm 4 RH2 fast-path transaction implementation

```
1: function RH2_FastPath_start(ctx)
2:   if is_all_software_slow_path > 0 then
3:     RH2_FastPath_SR_start(ctx)
4:     return
5:   end if
6:   HTM_Start()
       ▷ Fast-Path monitors the is_all_software_slow_path global counter to be 0 for the duration of the hardware transaction
7:   if is_all_software_slow_path > 0 then  ▷ speculative load of the global value
8:     HTM_Abort(ctx)
9:   end if
10: end function
11:
12: function RH2_FastPath_write(ctx, addr, value)
       ▷ log the write
13:  ctx.write_set ← ctx.write_set ∪ {addr}
       ▷ write value to memory
14:  store(addr, value)
15: end function
16:
17: function RH2_FastPath_read(ctx, addr)
       ▷ no instrumentation - simply read the location
18:  return load(addr)
19: end function
20:
21: function RH2_FastPath_commit(ctx)
       ▷ read-only transactions commit immediately
22:  if ctx.write_set is empty then
23:    return
24:  end if
       ▷ verify the write-set locations are not read by concurrent software transactions
25:  total_mask ← 0
26:  for addr ∈ ctx.write_set do
27:    s_index ← get_stripe_index(addr)
28:    mask_arr ← stripe_read_mask_array
       ▷ || - bitwise OR operation
29:    total_mask ← total_mask || mask_arr[s_index]
30:  end for
31:  if total_mask ≠ 0 then
32:    HTM_Abort()  ▷ there is a concurrent software reader
33:  end if
       ▷ put locks on the write-set locations.
34:  lock_mask ← (ctx.thread_id * 2) + 1
35:  for addr ∈ ctx.write_set do
36:    s_index ← get_stripe_index(addr)
37:    cur_ver ← stripe_version_array[s_index]
38:    if is_locked_by_me(ctx, cur_ver) then
39:      continue
40:    end if
41:    if is_locked(cur_ver) then
42:      HTM_Abort()
43:    end if
44:    stripe_version_array[s_index] ← lock_mask
45:  end for
46:  HTM_Commit()
47:  if HTM commit successful then
       ▷ now the write-set locations are updated and locked - unlock the write-set locations by updating their versions to the next one.
48:    next_version ← GVNext()
49:    for addr ∈ ctx.write_set do
50:      s_index ← get_stripe_index(addr)
51:      stripe_version_array[s_index] ← next_version
52:    end for
53:    return
54:  end if
55: end function
```

Algorithm 5 RH2 slow-path transaction implementation

```
1: function RH2_SlowPath_start(ctx)
2:   ctx.tx_version ← GVRead()
3: end function
4:
5: function RH2_SlowPath_write(ctx, addr, value)
       ▷ add to write-set
6:   ctx.write_set ← ctx.write_set ∪ {addr, value}
7: end function
8:
9: function RH2_SlowPath_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:  ctx.read_set ← ctx.read_set ∪ {addr}
       ▷ try to read the memory location
14:  s_index ← get_stripe_index(addr)
15:  ver_before ← stripe_version_array[s_index]
16:  value ← load(addr)
17:  ver_after ← stripe_version_array[s_index]
18:  if ver_before ≤ ctx.tx_version and ver_before = ver_after then
19:    return value
20:  else
21:    stm_abort(ctx)
22:  end if
23: end function
24:
25: function RH2_SlowPath_commit(ctx)
       ▷ read-only transactions commit immediately
26:  if ctx.write_set is empty then
27:    return
28:  end if
       ▷ set locking and visibility
29:  lock_write_set(ctx)
30:  make_visible_read_set(ctx)
       ▷ commit validation
31:  revalidate_read_set(ctx)
       ▷ perform the writes atomically
32:  while True do
33:    HTM_Start()
34:    write the write-set values to memory
35:    HTM_Commit()
36:    if the HTM transaction failed due to contention then
37:      continue  ▷ retry HTM transaction
38:    else
39:      fetch_and_add(is_all_software_slow_path)
40:      write-back the write-set using regular store instructions.
41:      fetch_and_dec(is_all_software_slow_path)
42:    end if
43:  end while
       ▷ reset locking and visibility
44:  next_version ← GVNext()
45:  release_locks(addr, next_version)
46:  reset_visible_read_set(ctx)
47: end function
```

Algorithm 6 RH2 fast-path-slow-read transaction implementation

```
1: function RH2_FastPath_SR_start(ctx)
2:     ctx.tx_version ← GVRead(ctx, global_version)
3:     HTM_Start()
4: end function
5:
6: function RH2_FastPath_SR_write(ctx, addr, value)
       ▷ log the write
7:     ctx.write_set ← ctx.write_set ∪ {addr}
       ▷ write value to memory
8:     store(addr, value)
9: end function
10:
11: function RH2_FastPath_SR_read(ctx, addr)
       ▷ try to read the memory location
12:    s_index ← get_stripe_index(addr)
13:    version ← stripe_version_array[s_index]
14:    value ← load(addr)
15:    if ¬ is_locked(version) and version ≤ ctx.tx_version
   then
16:        return value
17:    else
18:        HTM_abort(ctx)
19:    end if
20: end function
21:
22: function RH2_FastPath_SR_commit(ctx)
       ▷ read-only transactions commit immediately
23:    if ctx.write_set is empty then
24:        HTM_Commit()
25:    end if
       ▷ put locks on the write-set locations.
26:    lock_mask ← (ctx.thread_id * 2) + 1
27:    for addr ∈ ctx.write_set do
28:        s_index ← get_stripe_index(addr)
29:        cur_ver ← stripe_version_array[s_index]
30:        if is_locked_by_me(ctx, cur_ver) then
31:            continue
32:        end if
33:        if is_locked(cur_ver) then
34:            HTM_Abort()
35:        end if
36:        stripe_version_array[s_index] ← lock_mask
37:    end for
38:    HTM_Commit()
39:    if HTM commit successful then
           ▷ now the write-set locations are updated and locked -
       unlock the write-set locations by updating their versions to
       the new one.
40:        next_version ← GVNext()
41:        for addr ∈ ctx.write_set do
42:            s_index ← get_stripe_index(addr)
43:            stripe_version_array[s_index] = next_version
44:        end for
45:        return
46:    end if
47: end function
```

Algorithm 7 RH2 slow-path transactions: additional functions

```
1: function is_locked(stripe_version)
2:     return (stripe_version & 1) = 1   ▷ checks if the low order
   bit is 1
3: end function
4:
5: function is_locked_by_me(ctx, stripe_version)
6:     lock_mask ← (ctx.thread_id * 2) + 1
7:     return version = lock_mask
8: end function
9:
10: function lock_write_set(ctx)
11:    lock_mask ← (ctx.thread_id * 2) + 1
12:    for addr ∈ ctx.write_set do
13:        s_index ← get_stripe_index(addr)
14:        ver ← stripe_version_array[s_index]
15:        if is_locked_by_me(ctx, ver) then
16:            continue                    ▷ to next - already locked
17:        end if
18:        if is_locked(ver) then
19:            stm_abort(ctx)              ▷ someone else locked
20:        end if
21:        if ver ≠ CAS(stripe_version_array[s_index], ver, lock_mask)
   then
22:            stm_abort(ctx)              ▷ someone else locked
23:        end if
24:    end for
25: end function
26:
27: function revalidate_read_set(ctx)
28:    for addr ∈ ctx.read_set do
29:        s_index ← get_stripe_index(addr)
30:        version ← stripe_version_array[s_index]
31:        if is_locked_by_me(ctx, version) then
32:            continue
33:        end if
34:        if is_locked(ctx, version) then
35:            stm_abort(ctx)
36:        end if
37:        if version > ctx.tx_version then
38:            stm_abort(ctx)
39:        end if
40:    end for
41: end function
42:
43: function release_locks(addr, new_version)
44:    for addr ∈ ctx.write_set do
45:        s_index ← get_stripe_index(addr)
46:        stripe_version_array[s_index] ← new_version
47:    end for
48: end function
49:
50: function make_visible_read_set(ctx)
51:    for addr ∈ ctx.read_set do
52:        s_index ← get_stripe_index(addr)
53:        mask_arr ← stripe_read_mask_array
54:        id ← ctx.thread_id
55:        if (mask_arr[s_index] & 2^id) = 0 then
           ▷ turn ON the id-th bit of the read mask
56:            fetch_and_add(mask_arr[s_index], 2^id)
57:        end if
58:    end for
59: end function
60:
61: function reset_visible_read_set(ctx)
62:    for addr ∈ ctx.read_set do
63:        s_index ← get_stripe_index(addr)
64:        mask_arr ← stripe_read_mask_array
65:        id ← ctx.thread_id
66:        if (mask_arr[s_index] & 2^id) ≠ 0 then
           ▷ turn OFF the id-th bit of the read mask
67:            fetch_and_add(mask_arr[s_index], (−2^id))
68:        end if
69:    end for
70: end function
```