Hardware Read-Write Lock Elision

Pascal Felber
University of Neuchâtel
pascal.felber@unine.ch

Shady Issa
INESC-ID / Instituto Superior Técnico, University of Lisbon
shadi.issa@tecnico.ulisboa.pt

Alexander Matveev
MIT
amatveev@csail.mit.edu

Paolo Romano
INESC-ID / Instituto Superior Técnico, University of Lisbon
paolo.romano@tecnico.ulisboa.pt

Abstract

Hardware Lock Elision (HLE) represents a promising technique to enhance parallelism of concurrent applications relying on conventional, lock-based synchronization. The idea at the basis of current HLE approaches is to wrap critical sections into hardware transactions: this allows critical sections to be executed in parallel using a speculative approach, while leveraging on conflict detection capabilities provided by hardware transactions to ensure equivalent semantics to pessimistic lock-based synchronization.

In this paper we present *RW-LE*, the first HLE approach targeting read-write locks. RW-LE introduces an innovative hardware-software co-design that exploits two recent micro-architectural features of POWER8 processors: suspending/resuming transaction execution and rollback-only transactions. RW-LE's original design provides two major benefits with respect to existing HLE techniques: i) eliding the read lock without resorting to the use of hardware transactions, and ii) avoiding to track read memory accesses issued in the write critical section.

We evaluate RW-LE by means of an extensive experimental study based on a variety of benchmarks and real-life, complex applications. Our results demonstrate that RW-LE can provide striking performance gain of up to one order of magnitude with respect to state of the art HLE approaches.

1. Introduction

Over the last few years, hardware supports for transactional memory (TM) have been integrated in several mainstream commercial processors employed in a variety of computing platforms, ranging from commodity systems (Intel's Haswell [33]), to servers (IBM's POWER8 [20]) and super computers (IBM zEC12 [17]).

Processors equipped with hardware transactional memory (HTM) include assembly instructions that provide support for demarcating code blocks, which are guaranteed to be executed as atomic *transactions*. The HTM system is responsible for ensuring semantics equivalent to sequential execution of transactions. In order to maximize parallelism, though, transactions are executed in a speculative fashion, exploiting hardware facilities (typically extensions of the processor's cache coherency protocol) to detect conflicts at run-time and abort/restart transactions that would violate consistency.

The availability of HTM in commercial processors enables a range of novel applications, ranging from transactional programming [1] to thread-level speculation [30] and security [12]. Hardware lock elision (HLE) is probably among the most exciting ones, as well as, arguably, one of the main drivers motivating the commercial adoption of HTM.

HLE allows for enhancing the concurrency of legacy lock-based code in a simple, yet effective way: by executing critical sections as speculative hardware transactions. Several recent studies, e.g., [10, 20], have demonstrated the potentiality of HLE to boost the parallelism of complex legacy applications [8, 29]. However, these studies also highlighted some relevant limitations of HLE approaches, which are inherently rooted to the restricted nature of current HTM implementations. In particular, existing HTM systems can only guarantee the correctness (and hence allow the commit) of transactions that perform a limited number of memory accesses. Transactions that exceed the hardware capacity have to be aborted and re-executed using a pessimistic lock-based synchronization. This can severely hamper performance and limit the current scope of applicability of HLE techniques.

This paper investigates, to the best of our knowledge for the first time in the literature, the design of a hardware elision technique for read-write locks. The resulting

EuroSys '16, April 18-21, 2016, London, United Kingdom

DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901346

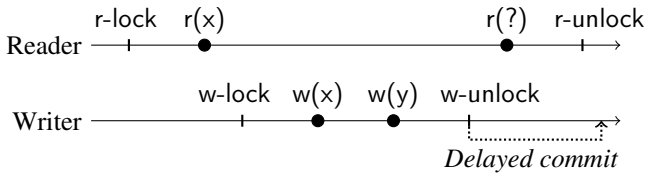

Figure 1: The write back of shared variables updated by a writer must be delayed until after all readers have completed their critical sections to preserve consistency.

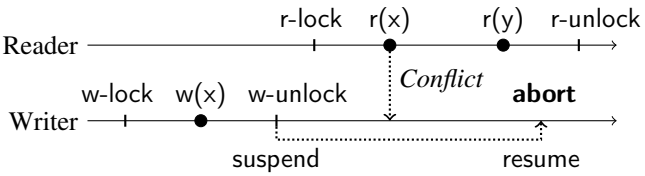

Figure 2: A new reader accessing a shared variable updated by a suspended writer will abort the suspended hardware speculation of the writer upon resume.

technique, which we call RW-LE, introduces an innovative hardware-software co-design that exploits two recent micro-architectural features of POWER8 processors:

1. *Suspend/resume*: the ability to suspend and resume a transaction, allowing, between the suspend and resume calls, for the execution of instructions/memory accesses that escape from the transactional context.
2. *Rollback-only transaction* (ROT): a lightweight form of transaction that has lower overhead than regular transactions but also weaker semantics. In particular ROTs avoid tracing load operations—hence having virtually unlimited capacity for memory accesses in read mode—but ensures the atomicity of the stores issued by a transaction, which appear to be all executed, as a unit, or not executed at all.[1]

RW-LE exploits the suspend/resume mechanism to elide read locks without resorting to the use of hardware transactions. This provides the key benefit of ensuring strong progress guarantees for the read critical sections, which are spared by spurious (and repeated) aborts caused by the underlying HTM implementation. However, unlike classical read-write lock implementations, RW-LE allows writers to execute concurrently with readers. Providing this concurrency is the key challenge in RW-LE and it works as follows.

First, writers execute using either HTM or ROT, which speculatively buffer their actual memory writes until the point of commit. This allows readers to proceed concurrently, because any read of a concurrent speculative write will abort the writer. However, it is unsafe for writers to commit when there are concurrent non-speculative reads. This is illustrated in Figure 1 where two threads, a reader and a writer, concurrently access two shared variables. As the writer executes its critical section fully between two read accesses, it cannot detect the concurrent execution of the non-speculative reader and committing immediately would expose the latter to an inconsistent snapshot, with a mix of old and new values (e.g., if $r(?) \equiv r(y)$ in the figure). To overcome this problem, the key idea in RW-LE is to suspend the hardware speculation of a writer, and then wait for all current readers to complete by using an RCU-like (epoch-based) quiescence mechanism [14, 15, 24]. This suspend-wait sequence has a two-fold effect. First, it drains all current readers that may read a write location of the writer, and therefore, could become inconsistent due to the commit of this writer. Second, any new reader that tries to read a write location of this writer will abort the suspended hardware speculation of the writer, as illustrated in Figure 2. As a result, after the wait is complete, it is safe to commit the writer, so RW-LE simply resumes hardware speculation and commits the writer.

RW-LE employs ROTs for write critical sections as an alternative synchronization scheme to plain hardware transactions. First, it attempts to execute write critical sections concurrently as plain hardware transactions. Next, when a write critical section repeatedly fails to commit, it activates the ROT-based synchronization path. The benefit of using ROTs is the reduced hardware speculation: they do not track memory reads and only track memory writes. This makes ROTs free from hardware capacity limitations due to read memory accesses, which usually compromise 80%-90% of the assembly code for read-dominated workloads. As a result, ROTs can successfully support a much larger class of (write) critical sections, which can still be executed concurrently with readers—waiting for the quiescence of any concurrent reader before committing, just as like for the plain hardware transactions. However, due to their limited isolation guarantees, a ROT cannot run concurrently with other ROTs, i.e., they need to be serialized. In other words, the ROT-based synchronization scheme trades off concurrency among writers to enhance the chance of successfully supporting concurrency between readers and writers.

We conducted an extensive experimental evaluation aimed at assessing the effectiveness of RW-LE with a variety of workloads, generated by a number of benchmarks and real-life, complex applications. We start by using a set of synthetic benchmarks that allow us to generate intensive and diverse usage patterns of the Virtual Memory (VM) system. We exploit these benchmarks to perform a sensitivity analysis on the efficiency of RW-LE when varying several key workload characteristics, such as the length and the ratio of the read-/write critical sections, as well as the likelihood of conflicts among writers and among readers and writers. Next, we consider three complex applications, namely STMBench7 [13], a standard benchmark for TM systems that mimics a cooperative CAD environment, Kyoto Cabinet [11], a commercially used database management library, and a porting of the well known TPC-C [31] benchmark.

[1] The IBM POWER8 specification states that ROT commit may not provide aggregate store semantics (atomic store). However, in practice, current POWER8 chips do.

The experimental results show that RW-LE achieves striking performance gains, up to 10× speedups compared to HLE and lock-based schemes, in a wide range of workloads. These include not only workloads in which HLE fails frequently due capacity exceptions, but also applications that, despite executing short critical sections, generate intensive load for the virtual memory subsystem (e.g., paging)—another potential cause of spurious aborts for HTM. Interestingly, our study shows also that, although being optimized for read-intensive workloads, RW-LE is competitive, and sometimes even outperforms, alternative synchronization schemes in a range of write-dominated workloads.

The remainder of this paper is structured as follows. Section 2 introduces related work. Section 3 presents the RW-LE algorithm. Section 4 reports the results of our experimental study. Finally, Section 5 concludes the paper.

2. Related Work

In a seminal paper, Rajwar and Goodman [27] proposed *hardware lock-elision* (HLE), an automatic way to introduce concurrency into lock-based critical sections, by executing these sections in a *fast path* as hardware transactions. However, hardware transactions are *best-effort* on current Intel Haswell [32] and IBM POWER8 [5] processors, which means that they provide no progress guarantee. A hardware transaction may always fail due to hardware limitation such as L1 cache capacity limitation, an unsupported instruction, or a page protection or scheduler interrupt. In all such cases it may never commit [10, 26]. Therefore, to ensure progress in HLE, a critical section that repeatedly fails to commit in hardware reverts to execute in a *fallback path* that acquires the original serial lock. This fallback path is expensive because it aborts all current fast-path hardware transactions and executes serially.

Afek et al. [2] recently proposed SCM for HLE, a conflict-management scheme that reduces the probability for hardware conflicts. It uses an auxiliary lock to serialize conflicting hardware transactions, so that each one still executes in hardware and can run concurrently with non-conflicting hardware transactions. In addition, Diegues and Romano [9] proposed an adaptive HLE conflict-management scheme that uses learning techniques to identify the best hardware retry configuration in a workload-oblivious manner.

Recent work by Afek et al. [2] and Calciu et al. [6] introduced the *lock removal* or *lazy subscription* lock elision schemes. Lock-removal sacrifices safety guarantees in favor of limited concurrency. The fast path can execute concurrently with the fallback path, but it cannot commit as long as there is a fallback path in progress and can observe inconsistent memory states. These inconsistent states can lead to executing illegal instructions or to memory corruption. Dice et al. [7] showed that complex software-side support is necessary to detect and handle all possible inconsistencies due to lock-removal, slowing down the lock-removal approach to a point that eliminates the advantages of using it in the first place. Therefore, in the same work it is proposed instead to rely on hardware extensions that can provide a fully safe HTM sandboxing capability.

Roy, Hand, and Harris [28] proposed a software-only implementation of the lock-elision approach in which transactions run speculatively and, when they fail or cannot execute, e.g., due to system calls, the application defaults to the original lock. Their system instruments all object accesses, both memory reads and writes, and employs a special kernel-based thread signaling mechanism. This software-based design provides better concurrency than HLE, but requires complex support and introduces severe overheads that make critical sections slow. Afek, Matveev and Shavit [4] proposed PLE, another software-only version of the lock-elision scheme for read-write locks, which uses a fully pessimistic STM. While the PLE algorithm is optimized for read-write locks, it still requires to instrument each memory read and write in software, which slows down critical sections and requires compiler support. Finally, a recent paper by Dice et al. [8] proposes ALE, an adaptive scheme that integrates both hardware and software for efficient lock elision, but it only works for specific access patterns and requires compiler support (as well as manually hand-crafted code modifications to avoid excessive software instrumentation costs). Also, unlike RW-LE, ALE is a general scheme and does not take advantage of the read-write lock semantics to optimize HLE's efficiency.

In addition to the work done on lock elision, several other approaches have been proposed to design synchronization mechanisms for read dominated workloads. The most famous one is the pthread implementation of read-write locks (RWL), which uses two counters to synchronize both readers and writers. RWL has an internal mutex that is used to synchronize the changes to these counters. The values of the counters are used to ensure fairness between readers and writers.

Another well-known synchronization mechanism is the big reader lock (BRLock), which was part of the linux kernel at one point [19]. The key idea behind BRLock is to trade off write throughput for read throughput. A thread acquiring BRLock in read mode will only lock one private mutex, whereas acquiring BRLock in write mode entails locking all private mutexes of running threads.

Recently, Liu et. al [21] introduced passive reader-writer lock (PRWL), a synchronization mechanism that tries to reduce the cost imposed by most reader-writer locks on the writer mode. PRWL leverages a version-based consensus protocol between readers and writers. Writers increment the lock version and wait for readers to signal that they have read the latest version. This approach was designed for total store order systems where an upper bound on the memory staleness can be guaranteed, i.e., on the time until readers see the latest version without a memory barrier.

Read-Copy-Update (RCU) [25] and Read-Log-Update (RLU) [22] represent an alternative paradigm for building read-dominated parallel software. The main idea behind RCU and RLU is to allow both read and write critical sections to

execute concurrently. In RCU, this is achieved by making writers modify a copy of the data structure while readers read the unmodified version. The new copy is only installed once all concurrent readers have finished their critical sections. RLU follows a similar approach but, instead of writers modifying a copy of the data structure, it uses per object logs that enable multiple writers to modify the same data structure. Despite being very efficient for read-dominated workloads, both techniques require tailored code for each application to handle the copying or logging of modifications. In our work, we target the elision of read-write locks without requiring any code modifications, unlike RCU and RLU.

Afek et al. [3] were the first to suggest the use of ROTs to hide writes from concurrent reads. This was an intial step towards RW-LE, which, in addition, incorporates an RCU-like quiescence scheme into the design.

3. The RW-LE Algorithm

In this section we describe our *read-write lock elision* (RW-LE) algorithm. We first introduce the key idea underlying RW-LE before discussing in depth its operating principles, implementation details, and various optimizations.

3.1 Basic Algorithm

The objective of RW-LE is to replace a traditional read-write lock (RWL) by a speculative variant that uses HTM. Yet, RW-LE uses HTM in ways unlike any other lock elision algorithms proposed so far.

The goal of lock elision is to avoid threads to have to "physically" acquire locks (or register as part of a reader list in the context of read-write locks). This is usually implemented by having threads, readers and writers alike, execute their critical sections in the context of hardware transactions. In contrast, RW-LE is optimized for read-dominated workloads and aims at supporting readers with (almost) no overhead. In particular, this means that readers should not even have to execute in the context of a hardware transaction, which would add non-negligible overheads.[2] To achieve this goal, RW-LE puts additional burden on writers as shown in Algorithm 1 and explained next.

To ensure proper synchronization with writers, RW-LE must keep track of which readers execute in a critical section. This is achieved by having every thread maintain a simple counter, or a logical clock, that is incremented in the RWLE_READ_LOCK() and RWLE_READ_UNLOCK() functions when respectively entering and leaving a read-side critical section—hence a clock has an odd value if the reader is in a critical section.[3]

As readers are not transactional, they might read inconsistent data if a concurrent writer modifies that data. Therefore, writers execute speculatively in the context of transactions and do not publish their updates unless it is safe to do so.

Algorithm 1 — RW-LE: basic algorithm (HTM only)

```
1:  Shared variables:
2:      clocks[N] ← {0, 0, ..., 0}          ▷ One counter per thread
3:      wlock ← FREE                         ▷ Spin lock to serialize writers
4:  Local variables:
5:      tid ∈ [0..N]                         ▷ Identifier of current thread
6:  function RWLE_SYNCHRONIZE
7:      c[N] ← clocks                        ▷ Read all clocks
8:      for i ← 0 to N−1 do                  ▷ Wait until all threads...
9:          if c[i] is odd then              ▷ ...that are in critical section...
10:             wait until clocks[N] ≠ c[i]  ▷ ...cross barrier
11: function RWLE_READ_LOCK
12:     clocks[tid] ← clocks[tid]+1          ▷ Enter critical section
13:     MEM_FENCE                            ▷ Make sure writers see reader
14: function RWLE_READ_UNLOCK
15:     clocks[tid] ← clocks[tid]+1          ▷ Exit critical section
16: function RWLE_WRITE_LOCK
17:     repeat                               ▷ Simple spin lock to serialize writers
18:         wait until wlock = FREE          ▷ Test and...
19:     until CAS(wlock, FREE, HTM-LOCKED)   ▷ ...test and set
20:     repeat until TX_BEGIN = SUCCESS      ▷ Start transaction
21: function RWLE_WRITE_UNLOCK
22:     TX_SUSPEND                           ▷ Suspend transaction
23:     wlock ← FREE                         ▷ We can already release lock
24:     RWLE_SYNCHRONIZE                     ▷ Let readers drain
25:     TX_RESUME                            ▷ Resume transaction
26:     TX_COMMIT                            ▷ Write back updates
```

Writers protect their critical sections by calling the RWLE_WRITE_LOCK() and RWLE_WRITE_UNLOCK() functions. Let us initially assume that there is a single writer. When entering the critical section, the writer starts a new hardware transaction. From that point on, and for the whole duration of the critical section, memory writes are speculative and hidden from non-transactional readers.

Consider a concurrent thread with a read-side critical section that overlaps with the writer's transaction and accesses the same shared variable. If the memory access of the reader occurs *after* the writer has updated the variable, then the writer's transaction will immediately abort and restart (or take an alternative fall-back path). If however the read occurs *before* the start of the write transaction, then no conflict will be detected and the reader will be serialized before the writer.

When a writer successfully reaches the end of its critical section, it must commit its transaction to write back its (speculative) updates. Yet, as readers are not transactional, doing so without precaution would break consistency. Indeed, a reader might see a mix of old and new data (prior and after the writer's transaction) whereas the read-side critical section should guarantee a consistent snapshot.

Therefore, before commit, the writer must wait for all readers that *might* have read some of the updated (yet uncommitted) data to have left their read-side critical section. Since we do not keep track of which memory locations have

[2] The begin and commit operations of transactions require tens to a few hundreds of cycles, and overheads become much more significant when they repeatedly abort and restart.

[3] Note that we do not consider here the nesting of critical sections (which can be supported using a simple counter to keep track of the nesting level).

been accessed by readers (no software instrumentations of memory accesses), we rely on lightweight, RCU-like, quiescence mechanism that simply waits for each active reader to end its critical section. This is implemented in the RWLE_SYNCHRONIZE() function by reading the clocks of each thread once and wait for all odd clocks to change value. Note that this quiescence mechanism does not prevent readers to start new critical sections as a read-after-write conflict will be handled as described above by aborting the writer.

An additional challenge is that the quiescence barrier cannot be implemented straightforwardly in the context of the writer's transaction, as the readers would abort the writer when incrementing their clocks. The key idea in our design is to exploit the suspend/resume feature of the POWER8 micro-architecture that allows us to temporarily suspend the active transaction, perform non-transactional operations, and later resume the transaction. Hence the quiescence barrier can execute non-transactionally.

Note that any conflict occurring while a transaction is suspended will trigger an abort upon resume, hence protecting concurrent readers from seeing inconsistent snapshots. Indeed, consider a reader that enters its critical section after the call to RWLE_SYNCHRONIZE(), i.e., it has not been seen by the writer and will execute concurrently with the write-back phase. If the reader accesses any memory location that has been updated by the writer before the write-back phase (which is atomic), then the latter will abort; otherwise the reader will see the new version. Hence consistency is preserved in all cases and the reader never blocks. It is also worth pointing out that we can already release the lock when suspending the transaction: letting another writer execute can at worse trigger an abort of the suspended transaction.

Algorithm 1 presents the pseudo-code of this basic version of RW-LE, where writers are serialized with a simple spin lock. In this version, we do not consider the cause of transaction aborts and instead blindly retry failed transactions. The TX_BEGIN() operation on line 20 returns a status code that indicates success (in which case the transaction can start executing speculatively) or error. If an abort happens during execution of the transaction, then controls jumps back to just after the call to TX_BEGIN() and the status code contains information about the failure cause. For the sake of simplicity, we assume in this paper that the status code can be SUCCESS, ABORT-TRANSIENT, or ABORT-PERSISTENT to respectively indicate if the transaction executes speculatively, or has aborted due a problem that is unlikely (e.g., contention) or likely (e.g., disallowed instruction, capacity) to be encountered again in a subsequent attempt.

3.2 Complete Algorithm with Fallback Paths

After giving the intuition of RW-LE, we now explain how we turn it into a practical algorithm with fallback paths to support transactions that repeatedly abort and to allow for concurrent writers. The complete pseudo-code is shown in Algorithm 2 and details of these extensions are explained in the rest of the section.

Non-Speculative Fallback. HTM has limitations that may prevent speculative execution to succeed, even in absence of contention. The most common scenario of non-contended abort is when a transaction reads or writes too many memory locations and exceeds the tracking limit for transactional storage accesses.[4] Other common causes of abort include execution of disallowed instructions within a transaction, or nesting transactions beyond the maximum level.

We therefore need to have a *fallback* path that can take over when speculative execution of an RW-LE writer repeatedly fails. A typical strategy consists in analyzing the abort reason and, depending on whether the failure cause is persistent or not, switching to a non-speculative path immediately or after a few unsuccessful retries. In our implementation, we delegate the selection of the path to the PATH() function (lines 23 and 28–40). This function retries the same path several times, until reaching a maximum number of attempts or experiencing an ABORT-PERSISTENT failure, in which case it switches to the next fallback path (ultimately defaulting to non-speculative execution).

As RW-LE has been designed to be lightweight and fast, we rely on a lock-based fallback path designed to be simple yet coexist seamlessly with speculative HTM-based transactions running concurrently. In its most basic version, the non-speculative fallback path serializes both readers and writers. The lock is set to FREE if only speculative writers execute and to NS-LOCKED if a non-speculative writer executes (only one at a time). The downside of integrating the fallback path is that readers now have to check the status of the lock and possibly wait if a non-speculative writer holds the lock (lines 12).

Rollback-Only Transactions. The POWER8 micro-architectures provides a lightweight form of transaction, called *rollback-only transaction* (ROT), that has lower overhead than regular transactions but also weaker semantics.

A ROT leverages transactional speculation and rollback mechanisms to ensure atomicity (i.e., the a sequence of instructions is executed, or not, as a unit), but there are fewer barriers (no barrier for transaction begin/end nor integrated cumulative barrier for reads and writes) and ROTs themselves are not serialized. Furthermore, ROT eliminates speculation on reads, i.e., there is no monitoring of storage locations specified by loads for modification by other processors, nor mechanisms between the performing of the loads and the completion of the ROT. As a result, a ROT hardware transaction only tracks memory stores. Nevertheless, updates within the ROTs are still hidden from other threads and follow the same rules for read and write conflicts as normal HTM transactions.

[4] Note that resources for tracking transactional storage accesses may be shared by multiple programs/threads executing concurrently.

Algorithm 2 — RW-LE: complete algorithm with fallback paths

```
    ...                                                  ▷ Same as basic algorithm
11: function RWLE_READ_LOCK
12:     wait until wlock ≠ NS-LOCKED                     ▷ Let writer finish
13:     clocks[tid] ← clocks[tid]+1                      ▷ Enter critical section
14:     if wlock = NS-LOCKED then                        ▷ New writer?
15:         RWLE_READ_UNLOCK                             ▷ Defer to writer...
16:         go to 12                                     ▷ ...and retry acquiring lock
17:     MEM_FENCE                                        ▷ Make sure writers see reader

18: function RWLE_READ_UNLOCK
19:     clocks[tid] ← clocks[tid]+1                      ▷ Exit critical section

20: function RWLE_WRITE_LOCK
21:     aborts ← 0                                       ▷ Keep track of failures
22:     status ← SUCCESS                                 ▷ Assume success for first trial
23:     path ← PATH(status, aborts)                      ▷ Which path?
24:     status ← RWLE_WRITE_LOCK_PATH                    ▷ Execute path
25:     if status ≠ SUCCESS then                         ▷ Success?
26:         aborts ← aborts+1                            ▷ No: register failure...
27:         go to 23                                     ▷ ...and retry

28: function PATH(status, aborts)
29:     if aborts = 0 then                               ▷ First trial?
30:         path ← HTM                                   ▷ Start with HTM path
31:         trials ← MAX-HTM                             ▷ Number of trials left
32:     if status = ABORT-PERSISTENT then                ▷ Worth retrying?
33:         trials ← 0                                   ▷ No: give up with current path
34:     if path = HTM and trials = 0 then                ▷ Done with HTM?
35:         path ← ROT                                   ▷ Switch to ROT path
36:         trials ← MAX-ROT                             ▷ Reset number of trials left
37:     if path = ROT and trials = 0 then                ▷ Done with ROT?
38:         path ← NS                                    ▷ Switch to NS path
39:     trials ← trials−1                                ▷ One less trial
40:     return path                                      ▷ Retry with selected path

41: function RWLE_WRITE_LOCK_HTM
42:     wait until wlock = FREE                          ▷ Let non-HTM writers finish
43:     status ← TX_BEGIN                                ▷ Start transaction
44:     if status = SUCCESS and wlock ≠ FREE then
45:         TX_ABORT                                     ▷ Defer to non-HTM writer
46:     return status                                    ▷ Return status code

47: function RWLE_WRITE_LOCK_ROT
48:     repeat                                           ▷ Simple spin lock to serialize writers
49:         wait until wlock = FREE                      ▷ Test and...
50:     until CAS(wlock, FREE, ROT-LOCKED)               ▷ ...test and set
51:     status ← TX_BEGIN_ROT                            ▷ Start ROT (return status)
52:     if status ≠ SUCCESS then                         ▷ If ROT failed
53:         wlock ← FREE                                 ▷ Handle abort
54:     return status                                    ▷ Return status code

55: function RWLE_WRITE_LOCK_NS
56:     repeat                                           ▷ Acquire global lock
57:         wait until wlock = FREE                      ▷ Test and...
58:     until CAS(wlock, FREE, NS-LOCKED)                ▷ ...test and set
59:     RWLE_SYNCHRONIZE                                 ▷ Let readers drain
60:     return SUCCESS                                   ▷ Always succeeds

61: function RWLE_WRITE_UNLOCK
62:     if wlock = NS-LOCKED then                        ▷ NS path?
63:         wlock ← FREE                                 ▷ Release lock
64:     else if wlock = ROT-LOCKED then                  ▷ ROT path?
65:         RWLE_SYNCHRONIZE                             ▷ Let readers drain
66:         TX_COMMIT                                    ▷ Write back updates
67:         wlock ← FREE                                 ▷ Release lock
68:     else                                             ▷ HTM path
69:         TX_SUSPEND                                   ▷ Suspend transaction
70:         RWLE_SYNCHRONIZE                             ▷ Let readers drain
71:         TX_RESUME                                    ▷ Resume transaction
72:         TX_COMMIT                                    ▷ Write back updates
```

Finally, the POWER ISA transactional memory specification states that the stores that are included in the ROT need not appear to be performed as an aggregate store [16], yet it also indicates that implementations are likely to provide an aggregate store appearance. We have conducted a broad range of experiments on our test machine and our findings confirm that current POWER8 chips do provide aggregate store for ROT commits.

Interestingly, we found that we could use ROT in an innovative way to provide a speculative fallback algorithm that can substitute to the slow lock-based path, while still preserving most of the benefits of the HTM-only base version, namely by allowing concurrent readers. The key insight in our use of ROTs is that, if they only protect the write-side critical sections and we do not allow concurrent writers, their weaker semantics are sufficient to still exploit their speculative properties and isolate writes from readers. Furthermore, since loads are not tracked, the likelihood of capacity aborts decreases and we do not need to use the relatively costly suspend/resume mechanism upon commit (lines 64–67).

To introduce this third code path to our algorithm, we use another state for the global lock, `ROT-LOCKED`, indicating that a writer executes in ROT mode.

Concurrent Writers. The final extension to make RW-LE truly scalable is to allow for concurrent writers in contention-free cases. This is achieved via a simple modification of the RWLE_WRITE_LOCK() function. Instead of serializing writers using the global lock, as in Algorithm 1 (lines 17–19), we try to continue executing speculatively but we check the status of the global lock as first operation of the transaction (Algorithm 2, line 44). By doing so, we can abort if the lock is not free and, at the same time, we also add the lock in the read set of the transaction. This essentially guarantees that another writer that takes the lock in a fallback path will abort immediately the transaction, hence preserving consistency.

Note that, as discussed above, ROT transactions cannot be used for concurrent writers and they are thus serialized (lines 48–50).

3.3 Discussion

We briefly discuss the properties and correctness of our algorithm, as well as some optimizations that are not shown in the pseudo-code but are part of our implementation.

Fairness Properties Unlike most of the read write lock algorithms that prioritize readers over writers, Algorithm 2 gives priority to non-speculative writers (i.e., writers that resort to acquiring the lock in `NS-LOCKED`) over readers. Indeed, a reader may be repeatedly overtaken by a writer at line 12 and at lines 14–16. While this might not be a problem in read-dominated workloads, which are our primary target, it still goes against our objective to optimize performance for the readers.

We have implemented a variant of our algorithm that embeds a version number in the global lock. When entering its critical section, a reader first increments its local clock, and then copies the global lock in a per-thread local lock. If the global lock is busy, the reader waits until the current lock owner releases the lock, before entering its critical section.

Upon acquiring the global lock, a writer increments the version number and only waits for readers that are in their critical section (odd local counter) and have started before the writer (local lock version smaller than the global lock version).[5] Therefore reader cannot be overtaken by writers, yet a writer that enter its critical section gets priority over new readers. Similar liveness properties are provided in most fair implementation of read-write locks (e.g., in Java [18]).

Experiments shown in Section 4 were conducted with the version of RW-LE shown in Algorithm 2. For completeness, we still compare both variants in separate experiment.

Correctness Argument. An RW-LE writer can protect its critical section using (1) HTM, (2) ROT, or (3) a full software write lock. In (1), writers execute concurrently and preserve consistency thanks to the hardware tracking mechanisms of HTM and because they eagerly subscribe the lock acquired by ROTs and non-speculative writers. In (2) and (3), writers simply execute serially. Therefore, the key for correctness of RW-LE is to ensure that read-side critical sections always execute on a consistent memory view (a snapshot). In other words, locations read by a read-side critical section must not be overwritten by a concurrent writer. To see why this is the case, let us analyze the commit process of each possible writer.

1. **HTM.** When it arrives to the commit, the writer first (a) suspends the HTM, and then (b) initiates the RCU-like quiescence loop. In this way, step (a) ensures that no updates become visible to readers at this point, while step (b) waits for current readers to finish (those that started before step (b)). Notice that these readers are the only ones whose snapshot could be invalidated by the commit of the writer. This is the case because any new reader that starts after step (b) and tries to read a memory location updated by the suspended writer will abort the latter. Therefore, after step (b) completes, if the HTM writer successfully resumes and commits, this implies that any other concurrent reader can be safely serialized after the writer—as, so far, no concurrent reader accessed any of the locations updated by the writer. Therefore the writer simply commits.
2. **ROT.** The commit works in a similar way: the writer just executes the RCU-like quiescence call. There is no need for suspend and resume calls in a ROT writer because it speculates only on writes.
3. **Full software write lock.** The writer simply executes as the original read-write lock without any readers. Therefore no readers can be overwritten.

Optimizations. We added several optimizations to RW-LE in order to improve performance. While they are not shown in the pseudo-code, they have been implemented and evaluated in our experiments. We have notably optimized our code as follows.

- The quiescence mechanism in RWLE_SYNCHRONIZE() traverses the array of per-thread clocks twice, first to copy it, then to wait for readers (lines 7 and 8 of Algorithm 1). When calling this function from the non-speculative path (lines 70 and 59 of Algorithm 2) we can optimize it by performing a single traversal of the array since readers are known to be blocked by the global lock.
- By moving the loop at line 12 of Algorithm 2 to just after line 15, we can optimize entry in the read-side critical section for uncontended cases, i.e., if the global lock is free we essentially save one comparison.
- By splitting the global lock in two variables, one acting as ROT lock and the other one as NS lock, we can modify the HTM path to eagerly subscribe to the NS lock (as at line 44 of Algorithm 2 to ensure that a non-speculative transaction aborts competing HTM transactions) and lazily subscribe to the ROT lock in the commit phase. This optimization enables concurrent execution of ROTs and HTM transactions, potentially improving performance when conflicts are infrequent.

4. Evaluation

This section reports the results of an extensive experimental evaluation that aims at quantifying the performance gains achievable by RW-LE with respect to state of the art solutions, based both on hardware lock-elision or pessimistic synchronization schemes, when faced with heterogeneous workloads.

We start by using synthetic benchmarks to generate diverse workloads that stress three key factors that are expected to impact the efficiency of RW-LE, namely:

1. **Critical section length**, which has a strong impact on the likelihood of capacity aborts for HLE-based solutions.
2. **Contention**, i.e., the likelihood of conflicting memory accesses from two critical sections.
3. **Update ratio**, i.e., the percentage of read vs. write locks.

[5] Note that, as for the global lock, the version numbers of local locks only increase.

We compare the performance of RW-LE to the following baselines: (1) a HLE scheme [27], which does not take advantage of the read-write lock semantics, (2) an implementation of the big reader lock (BRLock) that uses compare and swap for mutex acquisition, (3) the read-write lock (RWL) implementation of Linux pthreads' library, and (4) a plain single global lock (SGL). We do not include the passive reader-writer lock as it is designed for total store order architectures, which is not the case of PowerPC that provides weaker guarantees [23]. Furthermore, the source code that has been released by the authors is a kernel patch that is not compatible with our experimental platform.

Next, we evaluate RW-LE using two realistic, complex applications, namely STMBench7 [13] and KyotoCabinet Cache DB [11].

All presented results were obtained by executing on an 80-way IBM Power8 8284-22A processor with 10 physical cores, where each core can execute 8 hardware threads. The OS installed is Fedora 21 with Linux 3.17.3 and the compiler used is GCC 4.9.2 with `-O2` optimization level. The reported results represent the average of 10 runs.

In order to ensure reproducibility of results, the authors will open source both the RW-LE implementation and the benchmarks used in this study. This has not been done yet, in order to ensure the double-blindness of the reviewing process.

4.1 Sensitivity Study

In order to assess the effectiveness of RW-LE in diverse, yet clearly identifiable workload settings, we rely on a synthetic benchmark based on a hashmap synchronized via a single read-write lock (which we elide, when we use RW-LE and HLE). The hashmap is composed of l buckets, each one pointing to a linked list. We choose this benchmark since by varying l and the number of elements initially inserted in the hashmap, we can exert precise control on the workload characteristics and gain deeper understanding of the performance dynamics of the considered synchronization schemes.

We consider four different workload scenarios, differing by the likelihood of inducing HTM capacity exceptions and conflicting memory accesses. We control capacity exceptions by populating the hashmap with either $l{\cdot}200$ or $l{\cdot}50$ items, which lead, respectively, to about 50% (high-capacity) and 2% (low-capacitiy) probability of capacity exception (in absence of concurrency). As for conflict likelihood, we consider a low contention, where l =100,000 buckets, and a high contention scenario, where l =1 bucket.

We include in our sensitivity analysis two additional independent parameters, namely the ratio of write locks (w) requested by the application and the thread count. We consider three different values for the write lock probability, w: 1%, 10%, and 90%. We also performed experiments with 50% write locks but, as results were similar to the 90% case, they are not shown in the plots.

We stress that read-write locks are typically employed in read-dominated workloads. Hence, the inclusion of write

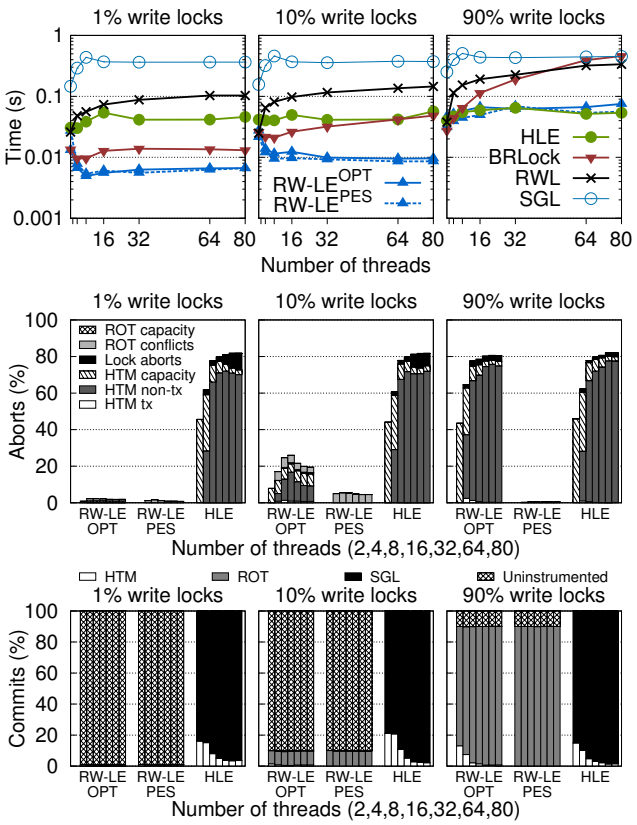

Figure 3: High capacity-high contention scenario: execution time, abort rate, and breakdown of commit types.

dominated workloads (w=90%) allow us to evaluate also worst-case scenarios for the proposed technique, in order to identify its actual limitations. Also, it should be noted that by considering hashmaps of diverse sizes, ranging from just 50 up to 200,000 items in total, combined with heterogeneous read/write ratios, we are not only stressing the HTM conflict and capacity exceptions, but also stressing the VM subsystem with intensive and heterogeneous memory access patterns, in terms of data access locality, page faults, and memory mappings.

Regarding the number of retries for hardware transactions and ROTs, we conducted experiments to explore a wide range of values for different workloads. We found out 5 to be the best value on average and in the majority of workloads, with only small differences in cases where it was not the best.. Also, this number range is the recommended value in other studies that recently evaluated HTM retry policies, e.g., [33]. As for RW-LE, we consider two variants (based on a writer-path policy):

- RW-LE$^{\text{OPT}}$: an optimistic version of RW-LE that attempts first 5 times executing with HTM, and then 5 times with the ROT-based path.
- RW-LE$^{\text{PES}}$: a pessimistic version of RW-LE that serializes writers and retries execution up to 5 times using the ROT-based path.

High capacity-high contention scenario. Figure 3 (top) reports the execution time for the scenario with high likelihood of contention and capacity exceptions, while varying the number of threads. The plots show remarkable gains for both RW-LE variants in the read-dominated workloads (first two columns with w={1%,10%}), with gains that extend up to 5× vs. BRLock (at 80 threads), 10× vs. HLE (at 16 threads) and up to 15× (at 64 threads) vs. the pthreads RWL. The reasons underlying these gains can be found in Figure 3(bottom), which reports the breakdown of the various paths used to commit transactions by the two RW-LE variants and by HLE. With HLE, due to the high probability of incurring capacity exceptions, only a limited fraction of critical sections is successfully elided via hardware transactions, and at high thread counts the non-speculative fallback path is activated more than 90% of the times. Conversely, RW-LE avoids instrumenting read critical sections. Further, even for write critical sections, it avoids activating the non-speculative path by falling back to the ROT-based path—which does not track read memory accesses, hence avoiding capacity exceptions and successfully eliding the write critical sections.

Figure 3 (middle) allow us to gain other interesting insights on the reasons underlying the performance gains of RW-LE vs. HLE, by reporting their abort rates broken down by the following abort causes: aborts of a hardware or rollback-only transaction caused by a conflict with a hardware transaction ("HTM tx" or "ROT conflicts"), by a capacity exception ("HTM/ROT capacity"), due to encountering the global lock busy upon its subscription ("Lock aborts") and due to a conflict with non-transactional code ("HTM non-tx"). With HLE, the latter class of abort causes typically corresponds to a conflict caused by a different thread acquiring the global lock or induced by the VM subsystem (e.g., paging). With RW-LE, it also includes the case of conflicts between a read critical section, which runs uninstrumented, and a writer using HTM or ROT. The data shows clearly that both RW-LE variants achieve a much lower abort rate than HLE. As expected, RW-LE^{PES} eliminates almost all capacity exceptions (avoids tracking memory reads). Conversely, with HLE, the abort rate grows quickly above 80% as the thread count grows. In fact, even despite the read dominated nature of these workloads, HLE falls pray of capacity aborts, which, as the thread count increase, lead to the frequent activation of the non-speculative fallback path and to the extermination of any concurrent hardware transaction.

Moving to the write-intensive workload, we can observe in Figure 3 (bottom) that, even in these unfavorable conditions, RW-LE exhibits a performance similar to HLE at a low thread count. Surprisingly, RW-LE^{PES} consistently outperforms HLE up to 8 threads, with speed-ups of approximately 20%. Thanks to the use ROTs (and to the serialization of writers), this variant of RW-LE exhibits very low abort rates even in this challenging scenario, unlike RW-LE^{OPT} and HLE. Indeed, RW-LE^{OPT} and HLE have similar total abort

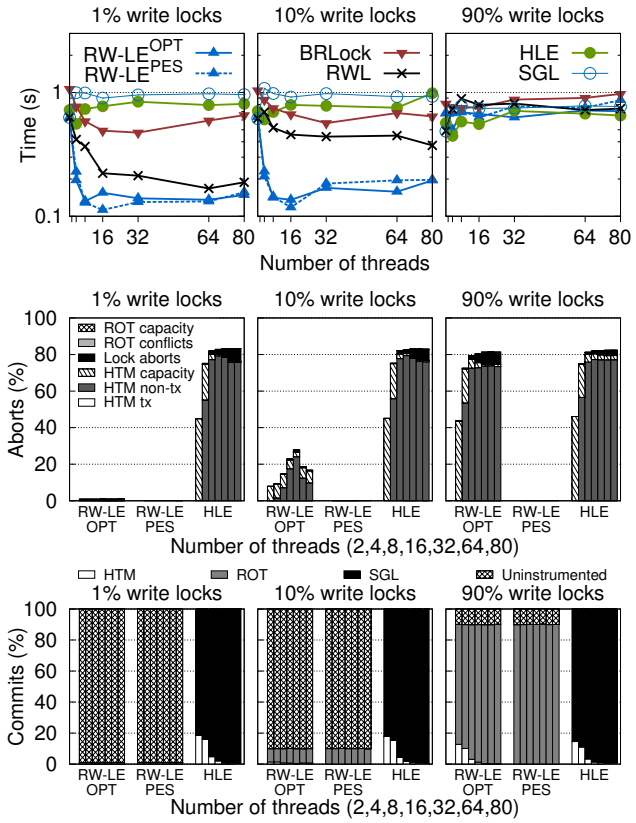

Figure 4: High capacity-low contention scenario: execution time, abort rate, and breakdown of commit types.

rates, as they are both very likely to fail executing a write critical section using a hardware transaction, and fall back, respectively, to ROT and GL (serial global lock). Unlike HLE, though, RW-LE^{OPT} can still execute readers in parallel with writers when it falls backs to ROT. The corresponding gains, however, are reduced in this write dominated workload, and appear to be counterbalanced by the additional costs associated with the quiescence call that RW-LE needs to execute for HTM and ROT based writers.

High capacity-low contention scenario. Let us now analyze a low contention scenario, while preserving a high probability of capacity exceptions. Also in this case, see Figure 4, we observe analogous performance trends: in the read intensive workloads, both RW-LE variants outperform all alternative synchronization schemes with remarkable gains (up to 4× speed-up when w=10%), while in the write dominated scenario, HLE exhibits performance very similar to RW-LE^{OPT}. As expectable, in this low contention workload, the pessimistic variant of RW-LE is slightly penalized with respect to the optimistic one, which can parallelize non-conflicting write critical sections thanks to the HTM-based path.

Low capacity-high contention scenario. Let us now discuss Figure 5, which reports results for scenario with low probability of incurring capacity exceptions and high contention (single linked list populated initially with 50 ele-

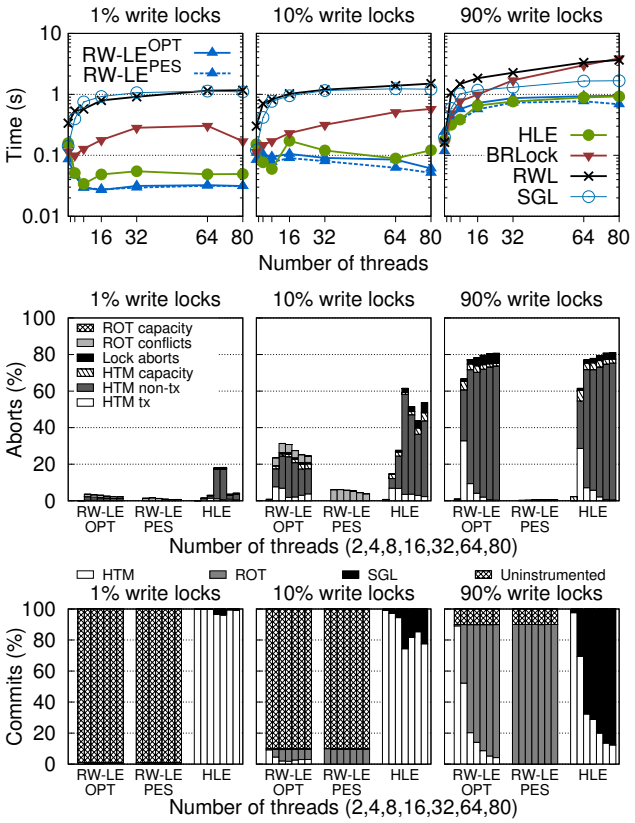

Figure 5: Low capacity-high contention scenario: execution time, abort rate, and breakdown of commit types.

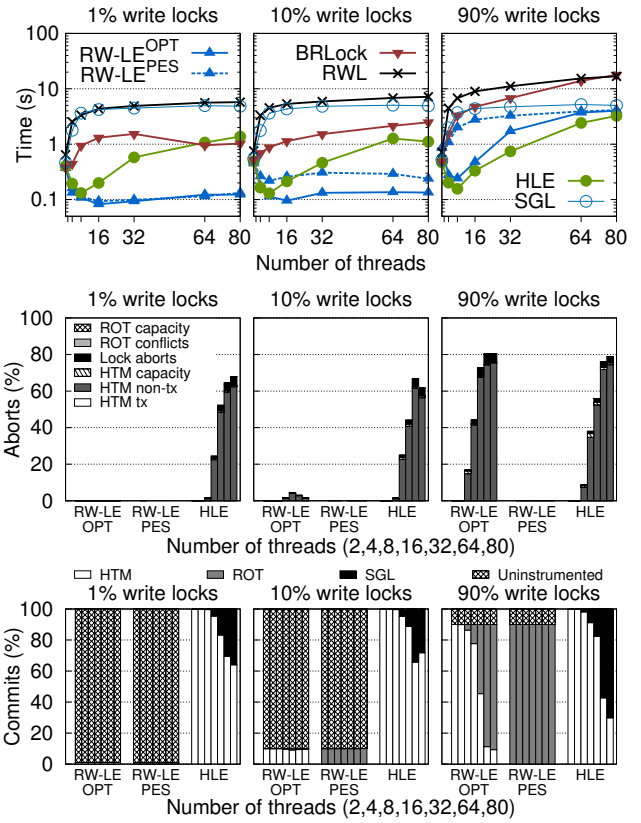

Figure 6: Low capacity-low contention scenario: execution time, abort rate, and breakdown of commit types.

ments). Scenarios with low likelihood of capacity exceptions are supposedly favorable for HLE, which can commit most of the times using the HTM-based path. Nonetheless, in the read-dominated workloads, both RW-LE variants incur much lower abort rates than HLE at high thread counts, achieving significant performance benefits with respect to HLE: average speed-up of approximately 60% for w=1% and 50% for w=10%, when using 16 threads or more.

In this high contention scenario, in fact, HTM transactions are likely to exhaust their retry budget due to conflicts with other hardware transactions. This causes the acquisition of the non-speculative fall-back path, which serializes also the execution of concurrent read critical sections. Conversely, RW-LE can fall-back to ROTs, which are serialized and hence protected from conflicts with concurrent writers. Also, ROTs are much less prone to suffer from capacity exceptions as they do not track reads, and, most importantly, can execute in parallel with readers.

Finally, as expected, the write dominated workload (w=90%) does not scale due to high contention with any of the considered synchronization primitives. Yet, it is noteworthy to mention that both RW-LE variants are competitive with HLE (the best performing of the baselines) and that RW-LE$^{\text{PES}}$ indeed outperforms HLE by around 35% at 80 threads.

Low capacity-low contention scenario. Figure 6 shows results for scenario with low probability of capacity exceptions and low contention. Recall that this scenario uses a hashmap composed of 100,000 linked lists, initially populated with 50 elements each. This setting is the only one where HTM transactions succeed well, but fail due to sparse memory access patterns induced by the large number of buckets. As a result, HTM also suffers from page fault interrupt aborts of the VM subsystem.

Results are shown in Figure 6(middle).As one can see, HLE shows virtually no capacity exceptions, yet a spiking number of non-transactional aborts that are related to page faults. This introduces a severe trashing effect for high thread counts. On the other hand, in read dominated workloads, both RW-LE variants seldom rely on HTM, and hence suffer much less from page faults aborts. This allows them to scale up to 16 threads.At 80 threads, the gains of the RW-LE$^{\text{OPT}}$ variant vs. HLE (which is also in this case the best alternative baseline) extend to an impressive factor of over 10$\times$ in both the w=1% and w=10% cases. On the other hand, as expected, in low contention scenario, RW-LE^{PES} is penalized by the need for serializing writers and pays a significant performance toll to the optimistic variant, with an average slowdown of approximately 2$\times$.

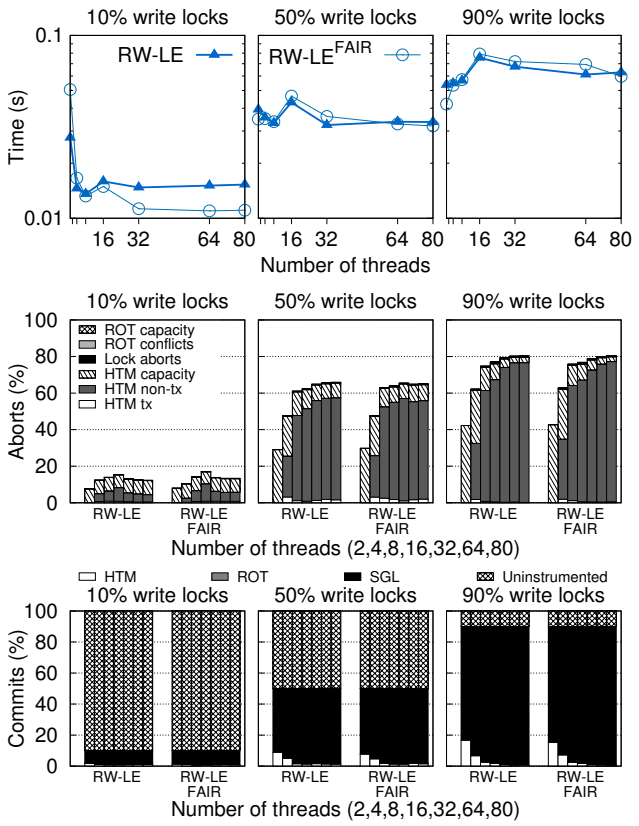

Figure 7: Fairness stress scenario: execution time, abort rate, and breakdown of commit types.

Finally, when analyzing the write dominated workload (w=90%), we find a scenario in which even the best RW-LE variant, RW-LE^{OPT}, incurs a non-negligible performance penalty of approximately 25% compared to HLE. This is not surprising, as 90% of the critical sections in this scenario are executed using HTM by both HLE and RW-LE^{OPT}. As such, they have the same likelihood of incurring an abort due to paging—the predominant abort cause with this workload. RW-LE^{OPT}, however, suffers from additional overheads due eliding, suspending and resuming hardware transactions. The pessimistic variant of RW-LE has poorer performance, as expected, given that it serializes 90% of the critical sections, which can, instead, be successfully parallelized using HTM by HLE and RW-LE^{OPT} (at least at low/moderate thread counts).

Fairness Finally, we ran an experiment to demonstrate the performance of the fair variant of the algorithm (RW-LE^{FAIR}) that we described previously in Section 3.3. In this experiment we use a different configuration of RW-LE where we disable ROTs as a fallback path, in order to stress the non-speculative fallback path that represents the main cause of unfairness. We use the same scenario as above (high probability of capacity exceptions and high contention) to increase likelihood of executing the fallback path. In Figure 7, we can see that the fair variant of the algorithm performs better at high numbers of threads and low update percentages. This is the case when readers may starve because of the non-speculative fallback path. Other than that, the performance of both variants is very similar. As the update percentage increases, the number of readers decreases and thus the effect of readers' starvation becomes less prominent.

Conclusions. Summing up, the above sensitivity study highlighted that:

- RW-LE's performance excels with workloads that are likely to induce capacity exceptions, achieving peak gains of up to 10× compared to the best of the considered baselines.
- The larger the dominance of read critical sections, the larger the gains. This is particularly beneficial for legacy code, where read-write locks are normally employed in read intensive workloads.
- Surprisingly, RW-LE can largely outperform (by up to 10×) all the considered baselines also in scenarios that rarely exhibit capacity exceptions. This is due to two main reasons: by running readers without any instrumentations and hardware speculation, RW-LE is less prone to spurious aborts caused by the HTM and the VM subsystem; and, in high contention workloads, the usage of the serial path significantly alleviates contention on the global lock. It is worth noting here that, consequently, RW-LE will still excel even if the underlying platform does not guarantee aggregate store for ROTs.
- Although RW-LE is optimized for read-intensive workloads, it can still deliver competitive performance in write-dominated workloads. In fact, in 3 of the 4 considered scenarios RW-LE has close, and sometimes even better (up to 25%), performance than the best alternative scheme (HLE), even with workloads that activate write critical section with 90% probability.
- Although RW-LE showed, overall, very robust performance in all the considered scenarios, it does have limitations. In particular, in write dominated workloads where HTM can elide critical sections with a high success rate (e.g., workloads with rare capacity exceptions, limited contention and paging frequency), RW-LE does pay performance tolls to HLE (up to 25% in our study).
- The fair variant of RW-LE can be beneficial in cases where readers starve due to the non-speculative fallback path and it does not introduce significant overheads.

4.2 Complex Applications

In this section, we evaluate RW-LE's performance using three complex applications, which are commonly used to assess the efficiency of speculative lock elision and TM systems.

STMBench7. STMBench7 [13] is a complex benchmark that simulates a cooperative CAD environment, with many heterogeneous transactions over a large and complex graph of objects. Originally designed for TM applications, we adapted it to use a read-write lock interface, by having read-only transactions acquiring a read-write lock in read

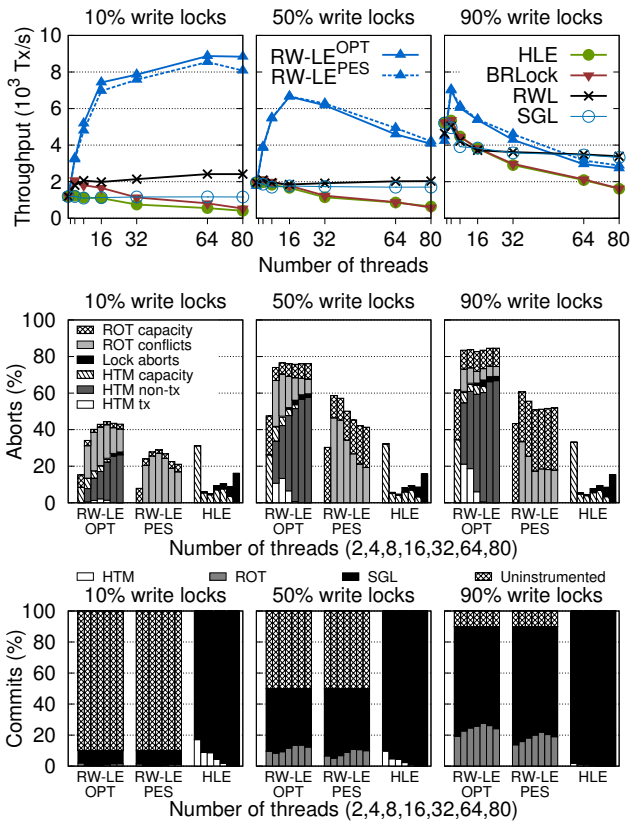

Figure 8: STMBench7 using different percentages of update operations: throughput, abort rate, and breakdown of commit types.

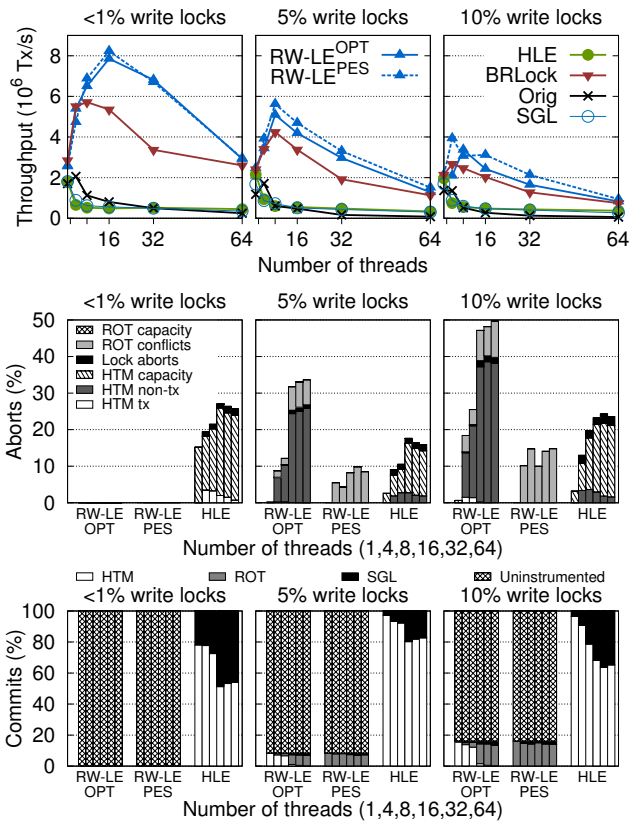

Figure 9: Kyoto Cabinet (wicked benchmark): throughput, abort rate, and breakdown of commit types.

mode, and update transactions in write mode. We evaluate the benchmarks using a standard workload, which we configured to use a medium size database and generate 24 different operations using default mixes (disabling long traversals and maintenance structural modifications).

Figure 8 (top) reports the throughput achieved by both RW-LE's variants and by the considered baselines, as the thread count and the percentage of update operations vary. The plots highlight that, up to about 50% of write operations, both RW-LE variants outperform the best baseline (RWL) with average gains around 2× that extend up to more than 4× in some scenarios, e.g., w=10% and 64 threads. The gains compared to HLE is even larger, extending to around one order of magnitude.

The gains over HLE can be explained by analyzing the abort rate and commit type breakdowns in Figure 8 (middle) and (bottom). STMBench7, as typical of many real-life applications, has relatively large read/write critical sections, which often fail to execute as speculative hardware transactions due to capacity exceptions (this is better seen with a low number of threads, where capacity exceptions are normally the only main abort cause). The need for executing frequently operations in a non-speculative way generates high contention levels on the lock-based fallback path, especially in read-dominated workloads. As a consequence, only very few transactions commit using HTM, which hinders performance severely.

Interestingly, and quite unexpectedly, with STMBench7, both RW-LE variants represent the most competitive synchronization schemes even in a strongly write dominated workload (90% write operations) up to 64 threads. Also in these settings, HLE's performance continue to be crippled by capacity exceptions, which enforces serialization of almost every transaction, yielding worst performance than non-speculative approaches.

Kyoto Cabinet. Next, we consider Kyoto Cabinet [11], a commercial C++ database management library. Here we focus on the in-memory variant KyotoCacheDB. Internally, it breaks the database into slots, where each slot is composed of buckets and each bucket is a search tree. To synchronize database operations, it uses a single global read-write lock, which contains nested per-slot mutexes. When using RW-LE, we elide the external read-write lock, preserving the internal mutex. This is only possible because RW-LE is aware (and takes advantage) of the read write lock semantics. With HLE, we instead elide both the inner and outer lock using HTM.

We use the `wicked` benchmark, which comes with Kyoto Cabinet, to execute a random set of database operations and control the frequency with which the external read-write lock

is acquired either in read or in write mode. Figure 9 reports the results obtained using three workload mixes that generate different rates of acquisitions of the external read-write lock, i.e., 10%, 5%, and $< 1\%$.

Looking at the read-dominated workload, left column of Figure 9, we see that both RW-LE variants scale up to 16 threads before starting to suffer from contention on the inner mutex locks (which we do not elide with RW-LE), while BRLock scales only up to 8 threads. This is due to the fact that, upon write, BRLock must acquire as many mutexes as there are running threads and meanwhile block the readers, which is not the case of RW-LE. By analyzing the abort and commit breakdown plots, we can deduce that RW-LE's gains, in this scenario, are due to its ability to avoid instrumentation for read critical sections—which in this workload are executed with more than 99% probability.

As the rate of acquisition of the read write lock in exclusive mode grows (see the center and right columns of Figure 9), as expected, scalability decreases. In particular, the aborts' breakdown of RW-LE$^{\text{PES}}$ highlights that writers executing with ROTs suffer of non-negligible conflict rates: this suggests that this workload generates high contention between readers and writers, which in turn explains why scalability degrades as the percentage of writers increases. Further, in such a conflict prone scenario, the optimistic variant of ALG suffers of larger abort rates, which induce a small performance penalty compared to RW-LE$^{\text{PES}}$.

Yet, it is interesting to note that, even when faced with such a challenging workload, both RW-LE variants continue to outperform all other alternatives, achieving speed-ups with a peak of approx. $2\times$ with respect to the best alternative scheme.

TPC-C. The last application we consider is TPC-C [31], a well-known benchmark for relational database management systems that generates OLTP workloads, representative of a wholesale supplier. We ported the benchmark to operate on an in-memory data store and, just like with STMBench7, we replaced read-only transactions with a read critical section and update transactions with a write critical section.

Figure 10 shows the results obtained when considering three workloads that have, respectively, 1%, 10%, and 50% of update transactions. Note that, to enhance visualization, we report in this case speedups with respect to an execution using one thread and a single mutex (SGL), and not absolute performances[6].

Also this benchmark confirms that RW-LE's performances excel in read-dominated workloads, while remaining competitive even in write-intensive, and inherently non-scalable workloads. In the read dominated workloads, both RW-LE variants achieve gains that extend up to $6\times$ over BRLock (the best alternative synchronization scheme) and up to $36\times$ over

[6] This choice is due to the fact that, with the write-intensive workload, absolute throughput drops by more than order of magnitude, hindering visualization.

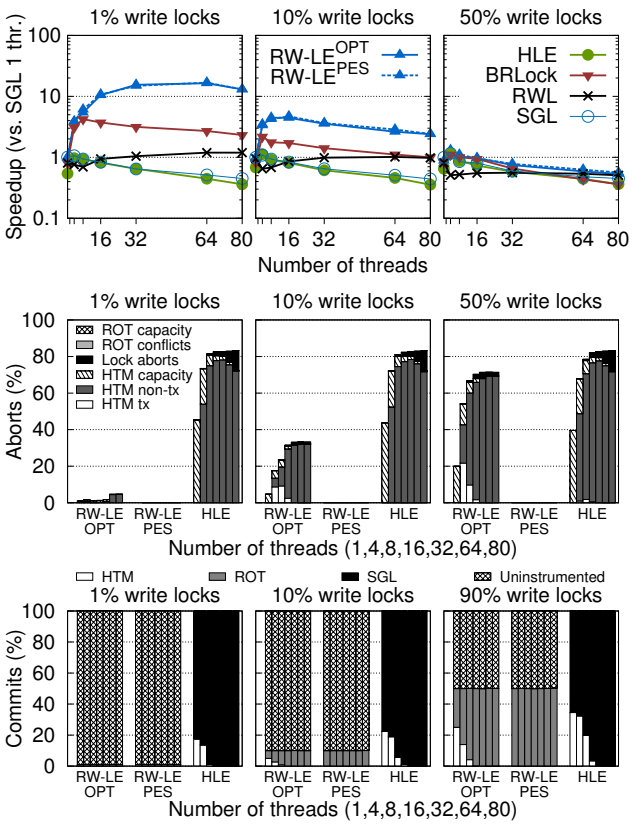

Figure 10: TPC-C: throughput, abort rate, and breakdown of commit types.

HLE (at 80 threads, 1% write probability). Also in this case, RW-LE's benefits in read-intensive workloads derive from its ability to execute readers uninstrumented. This is crucial with this benchmark, in which read critical sections fall prey of capacity exceptions in about 45% of the cases when using HLE.

In the write intensive workload, none of the synchronization mechanisms scale beyond 4 threads. Yet, RW-LE emerges as the (relatively) most efficient solution, with average gains of 25% with respect to the second best alternative (HLE). In this case, ROTs are crucial for RW-LE's efficiency, as they avoid capacity exceptions affecting update transactions and alleviate the risk of conflicts among writers (given that they are serialized).

5. Conclusion

In this paper we presented RW-LE, the first HLE approach targeting read-write locks, which exploits POWER8 HTM suspend/resume and rollback-only hardware transactions, in order to provide two major benefits over HLE: i) read-side critical sections execute without any instrumentation or hardware speculation, and ii) write-side critical sections use rollback-only hardware transactions to avoid tracking memory reads.

By means of an extensive experimental study, we have demonstrated that RW-LE can provide striking performance gains, i.e., up to one order of magnitude speed-ups, with respect to state of the art HLE approaches in a wide range of workloads.

Our work shows that special hardware transactional features, like rollback-only transactions and suspend/resume, can be used in new and unexpected ways to improve hardware lock elision, which deviate from their original intentions. We hope that the introduction of RW-LE will motivate other CPU manufacturers, besides IBM, to integrate such features in their future CPU generations.

Acknowledgements

This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and EXPL/EEI-ESS/0361/2013.

References

[1] ADL-TABATABAI, A.-R., KOZYRAKIS, C., AND SAHA, B. Transactional programming in a multi-core environment. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2007), PPoPP '07, ACM, pp. 272–272.

[2] AFEK, Y., LEVY, A., AND MORRISON, A. Software-improved hardware lock elision. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014* (2014), M. M. Halldórsson and S. Dolev, Eds., ACM, pp. 212–221.

[3] AFEK, Y., MATVEEV, A., AND SHAVIT, N. Reduced hardware lock elision. 6th Workshop on the Theory of Transactional Memory EuroTM WTTM 2014.

[4] AFEK, Y., MATVEEV, A., AND SHAVIT, N. Pessimistic software lock-elision. In *DISC* (2012), M. K. Aguilera, Ed., vol. 7611 of *Lecture Notes in Computer Science*, Springer, pp. 297–311.

[5] CAIN, H. W., MICHAEL, M. M., FREY, B., MAY, C., WILLIAMS, D., AND LE, H. Robust architectural support for transactional memory in the power architecture. *SIGARCH Comput. Archit. News 41*, 3 (June 2013), 225–236.

[6] CALCIU, I., SHPEISMAN, T., POKAM, G., AND HERLIHY, M. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing* (2014).

[7] DICE, D., HARRIS, T. L., KOGAN, A., LEV, Y., AND MOIR, M. Hardware extensions to make lazy subscription safe. *CoRR abs/1407.6968* (2014).

[8] DICE, D., KOGAN, A., LEV, Y., MERRIFIELD, T., AND MOIR, M. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2014), SPAA '14, ACM, pp. 188–197.

[9] DIEGUES, N., AND ROMANO, P. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.* (2014), X. Zhu, G. Casale, and X. Gu, Eds., USENIX Association, pp. 209–219.

[10] DIEGUES, N., ROMANO, P., AND RODRIGUES, L. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 3–14.

[11] FAL LABS. Kyoto cabinet: A straightforward implementation of DBM, 2011. http://fallabs.com/kyotocabinet/.

[12] GUAN, L., LIN, J., LUO, B., JING, J., AND WANG, J. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Security and Privacy (SP), 2015 IEEE Symposium on* (May 2015), pp. 3–19.

[13] GUERRAOUI, R., KAPALKA, M., AND VITEK, J. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 315–324.

[14] HARRIS, T. L. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)* (2001), pp. 300–314.

[15] HART, T. E., MCKENNEY, P. E., BROWN, A. D., AND WALPOLE, J. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput. 67*, 12 (2007), 1270–1285.

[16] IBM. Power ISA transactional memory, Dec. 2012. Version 2.07.

[17] JACOBI, C., SLEGEL, T., AND GREINER, D. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 25–36.

[18] JAVADOCS. Reentrant Read Write Lock. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html, 2016.

[19] JONATHAN CORBET. Big reader locks. http://lwn.net/Articles/378911/, 2010.

[20] LE, H., GUTHRIE, G., WILLIAMS, D., MICHAEL, M., FREY, B., STARKE, W., MAY, C., ODAIRA, R., AND NAKAIKE, T. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development 59*, 1 (Jan 2015), 8:1–8:14.

[21] LIU, R., ZHANG, H., AND CHEN, H. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 219–230.

[22] MATVEEV, A., SHAVIT, N., FELBER, P., AND MARLIER, P. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 168–183.

[23] MCKENNEY, P. E. Memory ordering in modern microprocessors, part i. *Linux Journal 2005*, 136 (2005), 2.

[24] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy-update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Oct. 1998), pp. 509–518.

[25] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, Oct. 1998), pp. 509–518.

[26] NAKAIKE, T., ODAIRA, R., GAUDET, M., MICHAEL, M. M., AND TOMARI, H. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA '15, ACM, pp. 144–157.

[27] RAJWAR, R., AND GOODMAN, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 2001), MICRO 34, IEEE Computer Society, pp. 294–305.

[28] ROY, A., HAND, S., AND HARRIS, T. A runtime system for software lock elision. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 261–274.

[29] SPEAR, M., RUAN, W., LIU, Y., AND VYAS, T. Case study: Using transactions in memcached. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, R. Guerraoui and P. Romano, Eds., vol. 8913 of *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 449–467.

[30] STEFFAN, J. G. *Hardware Support for Thread-level Speculation*. PhD thesis, Pittsburgh, PA, USA, 2003. AAI3159472.

[31] TPC COUNCIL. TPC-C Benchmark. http://www.tpc.org/tpcc, 2011.

[32] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013* (2013), W. Gropp and S. Matsuoka, Eds., ACM, pp. 19:1–19:11.

[33] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 19:1–19:11.