



Speculative Read Write Locks

Shady Issa

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

Paolo Romano

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

Tiago Lopes

INESC-ID / Instituto Superior Técnico,
University of Lisbon, Portugal
tiago.s.lopes@tecnico.ulisboa.pt

ABSTRACT

Hardware Transactional Memory (HTM) has recently entered the realm of mainstream computing thanks to its integration in processors commercialized by major industrial manufacturers. HTM provides highly-efficient, hardware-assisted synchronization mechanisms for concurrent programs. Unfortunately, though, existing HTM implementations also suffer from severe limitations that are inherently related to their best-effort, hardware-based design.

This work introduces SpRWL (Speculative Read Write Lock), a HTM-based implementation of read-write locks that provides a key benefit: allowing readers to execute outside the scope of hardware transactions, thus, effectively sparing them from any HTM-related limitation. SpRWL is the first HTM-based read-write lock implementation to support the concurrent execution of uninstrumented readers, while assuming a standard transaction demarcation API that is universally supported by any HTM implementation. Via an extensive experimental study, we show that SpRWL can achieve striking performance gains (up to 16×) with respect to state of the art read-write lock implementations based not only on pessimistic/lock-based schemes, but also on HTM-based techniques that exploit specific hardware mechanisms currently supported solely by a restricted number of architectures.

CCS CONCEPTS

• **Software and its engineering** → **Multithreading; Mutual exclusion; Concurrency control; Process synchronization;**

ACM Reference Format:

Shady Issa, Paolo Romano, and Tiago Lopes. 2018. Speculative Read Write Locks. In *19th International Middleware Conference (Middleware '18)*, December 10–14, 2018, Rennes, France. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3274808.3274825>

1 INTRODUCTION

Parallelism has become a pervasive characteristic of today's computer architectures and current trends suggest a steady growth of the number of available logical cores in the foreseeable future. In this technological landscape, a crucial problem is how to build scalable synchronization primitives that enable programmers to take full advantage of the potential of modern parallel architectures.

The introduction of Hardware Transactional Memory (HTM) by mainstream CPU manufacturers like Intel and IBM [32] responds

precisely to this urge. HTM provides a highly-efficient, hardware-assisted implementation of the atomic transaction abstraction, long used in the context of database systems, and now exposed to programmers/compiler via a dedicated extension of the processor instruction set. HTM requires programmers to wrap code blocks in transactions that will be executed concurrently, in a speculative fashion, while transparently ensuring semantics equivalent to sequential execution by detecting conflicts among concurrent transactions in hardware and aborting/restarting them, if necessary. HTM can be employed both as a transaction-centric synchronization mechanism explicitly leveraged and controlled by HTM-aware applications, as well as to enhance the parallelism of legacy applications based on pessimistic, lock-based synchronization primitives — a technique that goes under the name of *lock elision*.

A number of recent works have shown that HTM can reduce significantly the synchronization overheads incurred by parallel applications [14, 18, 37] in various application domains. Unfortunately, though, several studies have also highlighted that existing HTM implementations suffer from several limitations, stemming from the inherently restricted nature of the hardware mechanisms that they employ. Commercially available HTM systems, in fact, rely on (relatively) non-intrusive extensions of pre-existing cache coherency protocols, which can only ensure isolation and atomicity provided that the metadata associated with active transactions (in particular, their read-set and write-set) fits in the processor's cache.

Such a design approach limits the applicability of HTM in a number of ways: not only it explicitly restricts the maximum amount of different cache lines that can be accessed by transactions, but also makes hardware transactions unable to withstand events that lead to scratching the processor's cache, which includes, notably, system calls, context switches and interrupt requests (including periodic timer interrupts raised for OS scheduling purposes). Overall, these restrictions make current HTM systems unfit to serve as a general-purpose synchronization mechanism, significantly limiting the scope of their applicability.

This work aims at tackling precisely this issue, i.e., broadening the scope of applicability of HTM, by introducing SpRWL (Speculative Read Write Lock), a novel HTM-based synchronization primitive that provides a key benefit: allowing read-only critical sections to execute outside the scope of any hardware transaction, thus, effectively sparing them from the limitations that affect existing HTM implementations. This is beneficial for applications that encompass long read-only operations, such as range queries and long traversals. SpRWL's name stems from the fact that it exposes the familiar interface of a classic read-write lock (RWLock) and can, therefore, be seen as a specialized HTM-based technique for eliding this type of locks in legacy applications. Nevertheless, SpRWL can also be straightforwardly employed in applications that assume a transactional API by mapping the beginning of a read-only or

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

Middleware '18, December 10–14, 2018, Rennes, France

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5702-9/18/12...\$15.00

<https://doi.org/10.1145/3274808.3274825>

an update transaction to a request for acquiring a read or write lock, respectively. Throughout the paper, for the sake of brevity, we will refer to threads that request to acquire the lock in read/write mode (or, equivalently, to execute a read-only/update transaction) as readers/writers, respectively.

SpRWL combines two key novel techniques aimed, respectively, at ensuring safety and at maximizing efficiency.

SpRWL preserves safety of readers, which run outside the scope of hardware transactions, by using a simple, yet surprisingly effective, technique: it requires writers, which execute using HTM, to check, at commit time, whether there is any active reader, and allows them to commit only if none is found, forcing them to abort otherwise. The correctness of this approach hinges on two key properties of HTM:

- HTM externalizes the memory writes produced by a writer only if it successfully commits, making them visible to both transactional and non-transactional code atomically. This ensures that readers never observe uncommitted writes of concurrent writers.
- HTM detects conflicts between transactional and non-transactional code in an eager fashion, triggering the immediate abort of the former — a property that is known as *strong isolation* in the literature [4]. This property avoids data races, which might otherwise occur if a reader started, after its state was checked (and found inactive) by a concurrent update transaction. If the writer were to be allowed to commit in presence of a concurrent reader, the latter could observe an inconsistent snapshot, e.g., by returning different values upon two subsequent reads of the same memory position. SpRWL prevents such a scenario by leveraging the strong isolation property of HTM, which ensures that if a reader alters its own state after it has been checked by a concurrent writer, then the latter will be immediately aborted.

As we will show, the technique described above can yield up to 8× throughput gains over plain HTM in workloads with long readers. However, these throughput gains are achieved at the cost of an increased latency of writers, which can suffer from frequent aborts (and theoretically from starvation) in read-dominated workloads.

SpRWL addresses these shortcomings by complementing the above base algorithm with two ad-hoc scheduling schemes, which we refer to as reader synchronization and writer synchronization.

The reader synchronization scheme requires that readers, before starting, wait for the completion of active concurrent writers, if any. This technique not only reduces the likelihood for a writer to abort due to the existence of a concurrent reader. It also guarantees an important fairness property for writers, by ensuring that if a writer is activated before a reader, then the former cannot be aborted by the latter. The reader synchronization scheme of SpRWL is further enhanced by allowing readers to shortcut their initial waiting phase in case they find some other reader already waiting: in such a case, the last activated reader joins the one already waiting and starts as soon as the latter does. This brings two advantages: reducing the average duration of the readers' waiting phase — and, hence, their latency — and striving to minimize the duration of time windows during which some reader is active by aligning their start time — which increases the chance for a writer to commit successfully.

The writer synchronization scheme aims, instead, at optimizing the scheduling decision on when to activate a writer by pursuing

a twofold goal: on one hand, postponing the activation of writers in order to reduce the chances that they have to abort due to a concurrent reader; on the other hand, activating writers as early as possible, to maximize concurrency with already active readers. This is achieved by estimating, at run-time, the average duration of readers and writers, and accordingly delaying the activation of a writer in order to maximize the chances that it requests to commit shortly after the last concurrent reader has completed.

We evaluated SpRWL via an extensive experimental study conducted using the HTM implementations available on Intel's Broadwell [37] and IBM's POWER8 [26] CPUs. We consider both synthetic micro-benchmarks, aimed at assessing the sensitivity of the proposed solution to a broad spectrum of workloads, as well as a standard benchmark (TPC-C [15]) representative of complex/realistic applications. The results of our study show that SpRWL can achieve striking performance gains (up to 16×) with respect to state of the art RWLock implementations based not only on pessimistic/lock-based schemes, but also on HTM-based techniques that, unlike SpRWL, exploit specific hardware mechanisms currently supported solely by IBM processors [17].

The remainder of this paper is structured as follows. Section 2 reviews related work. The algorithm underlying SpRWL is presented in Section 3, along with a discussion on its correctness and on a set of relevant optimizations. Section 4 presents the results of the experimental evaluation of SpRWL. Finally, Section 5 concludes this work and discusses possible avenues for future research.

2 RELATED WORK

SpRWL is a novel HTM-based synchronization mechanism targeting read-dominated workloads, hence, it is related to two main areas of the literature: one aiming to enhance the efficiency of HTM systems, and one investigating the implementation of efficient synchronization mechanisms for read-intensive workloads.

Tuner [12] and Seer [13] are two instances of solutions that aim to enhance HTM's efficiency: Tuner uses online reinforcement learning techniques to optimize the retry policy of HTM transactions with the goal of reducing the frequency of activation of HTM's pessimistic fallback path; Seer relies on probabilistic models to infer the conflict patterns between transactions and accordingly schedule their execution to reduce contention. Both these techniques are orthogonal to the problem tackled in this paper, and might, in fact, be integrated within SpRWL. Calciu et al. [6] suggested lazy subscription to the fallback lock from within a HTM transaction to allow more parallelism by reducing the window during which a transaction can be aborted due to concurrent activation of the fallback path. Unfortunately, though, Dice et al. [10] identified a number of subtle issues that render lazy subscription unsafe.

RWLocks are probably one of the most common synchronization primitives targeted at workloads in which a large fraction of critical sections accesses shared data in read-only mode [9]. The idea behind RWLock is to support concurrent readers, while allowing the execution of only one critical section that modifies shared data at a time. Several works addressed the problem of devising efficient implementations of the RWLock abstraction. The most famous are probably the pthread's library implementation [1], which uses two counters protected by an internal mutex to track readers and

writers, and the classical MCS RWLock algorithms [31] that use a linked list instead of a counter to avoid spinning on global variables. Another RWLock implementation that was used previously in the Linux kernel is the Big Reader Lock (BRLock) [8]. With BRLock, a reader only acquires a per thread mutex, whereas writers acquire first a global mutex and then all the per thread mutexes. Passive Reader Writer Lock (PRWL) [28] is a recent RWLock solution that reduces the cost usually imposed by RWLock algorithms on writers. It uses a version based consensus where writers increment the lock version and wait for readers to signal they have read the latest version. *Phase-fair* RWLocks (PFRWLs) [5] are based on the idea of which fairly alternating the execution of readers and writers in distinct phases, in order to ensure a bounded wait time for writers. This phase-based execution model is similar in spirit to the one pursued by the scheduling policy adopted by SpRWL, which prevents readers from starting in presence of concurrent active writers. However, SpRWL's scheduling policy is designed to take advantage of the underlying HTM system (unlike PFRWLs, which adopt a pessimistic design), e.g., by allowing writers to run concurrently with readers, if the latter ones are expected to finish first.

Overall, when compared with SpRWL, the various algorithms that implement RWLock differ in two aspects: (i) they do not allow concurrency among writers, which instead SpRWL enables, thanks to HTM; and (ii) they do not allow concurrency between readers and writers — which SpRWL strives to maximize via scheduling techniques. The scheduling technique used by SpRWL is also related to the literature on back-off mechanisms for transactional systems, e.g., [19–21, 24, 27, 34]. Indeed, SpRWL's scheduling technique can be seen as a specialized back-off mechanism, which is designed to increase the chances for HTM-based writers to commit in absence of concurrent readers, while striving to overlap their execution.

Read-Copy-Update (RCU) [29] is an alternative synchronization mechanism for read-dominated workloads that allows concurrency between readers and writers without the need of HTM support. With RCU, writers must create a copy of the shared data they intend to update and apply their modifications to this copy. Readers that existed prior to the writer continue to access the old, unmodified data. Only when all readers activated before the writer started have completed is the writer allowed to apply the updates it produced, by atomically replacing the current data version with its updated copy. Similarly to SpRWL and unlike RWLocks, with RCU, readers do not need to acquire any mutex, but are only required to advertise when they start and end a critical section using a memory fence. Although RCU can provide significant performance gains, it comes with a high cost in terms of usability. To support RCU, programs must explicitly be designed to operate correctly over copies of shared data. This can be not trivial, for instance, if applications store pointers to a shared data region, which is discarded when a writer applies its updated version: in this case all references to the discarded memory region have also to be correctly updated. This limits the applicability of RCU and is arguably one of the key reasons a few number of RCU-based data structures have been proposed [3, 7, 30]. On the contrary, SpRWL does not require any changes neither to legacy RWLock-based programs, nor to the software development methodology.

Finally, Hardware Read-Write Lock Elision (RW-LE) [17] is probably the most related work to SpRWL. RW-LE was the first work

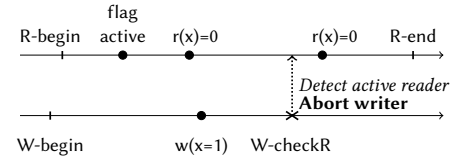


Figure 1: A writer aborting itself before it commits because there is an active reader protects the reader from witnessing inconsistent snapshots.

to propose a HTM-based lock elision technique specialized for RWLocks. Analogously to SpRWL, RW-LE executes read-only critical sections outside the scope of transactions. However, unlike SpRWL, RW-LE relies on two specific features currently available solely in POWER8's HTM implementation. Because of RW-LE's reliance on these non-standard HTM features, its scope of applicability is much more constrained with respect to SpRWL, which, on the contrary, can be employed on any HTM implementation — as it only assumes a plain API for transaction demarcation that simply allows for beginning, committing and aborting transactions. Analogous considerations apply to POWER8-TM [23], which can be seen as an extension of RW-LE that exploits POWER8's ROTs and S-R to expand the capacity limitations of update transactions. SpRWL is not only more generic than these solutions, but, as we will show in Section 4, when employed on POWER8, it can actually achieve significant performance gains in a broad range of workloads.

3 ALGORITHM

As mentioned, SpRWL exposes a plain read-write lock interface. As such, SpRWL can be used as a drop-in replacement for conventional¹ (e.g., pthread's) read-write locks in applications that already use this synchronization primitive. However, it is straightforward to employ SpRWL in TM-based applications, by mapping the begin and commit of read-only and update transactions to lock and unlock requests to a RWLock implemented using SpRWL.

For the sake of clarity, we present SpRWL in an incremental fashion. We start by presenting, in Section 3.1, a simple, base algorithm that embodies one of the key ideas at the basis of SpRWL: enabling safe concurrency between uninstrumented readers and HTM-backed writers. We extend this base algorithm in Section 3.2, by introducing two scheduling techniques that aim both at enhancing performance and ensuring fairness. We conclude by discussing the correctness of the proposed solution (Section 3.3) and presenting a set of relevant optimizations (Section 3.4).

3.1 Base Algorithm

The pseudo-code of SpRWL base algorithm is reported in Algorithm 1. As already mentioned, writers execute speculatively, using HTM: a write lock acquisition request triggers the activation of a HTM transaction and the corresponding unlock request triggers the commit of its associated hardware transaction. Readers, conversely, are

¹Being based on HTM, though, SpRWL still incurs some limitations that may prevent its use with certain RWLock implementations. For instance, SpRWL can not be used to elide a RWLock whose implementation allows for sharing the transactional context among multiple threads, since HTM does not support such a pattern.

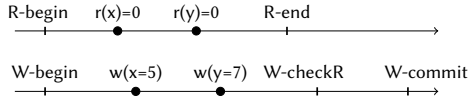


Figure 2: A reader committing before an active update transaction writes to memory or checks for readers allows the writer to successfully commit.

executed uninstrumented, i.e., without recurring to HTM, and are therefore spared from HTM's inherent limitations.

In order to ensure the safety of readers, in presence of concurrent writers executing in HTM, SpRWL uses the following mechanism. Before a reader, with thread id *tid*, is granted access to the read critical section, it first advertises its existence to concurrent writers in the *tid*-th entry of the *state* shared array. The update of the shared array is followed by a memory fence, which, as we will discuss, is key for correctness, as it ensures that the state of readers is globally visible before they enter the read critical section. Upon releasing the read lock, the reader's state is accordingly reset after issuing a memory fence to prevent reads in the critical section from being reordered after the state resetting. Note, that on TSO architectures, e.g. Intel x86, this last fence is not needed, since the hardware guarantees that loads cannot be reordered after a subsequent store.

Writers, in their turn, check for the existence of concurrent active readers, by inspecting the *state* array, upon requesting to release the write lock, i.e., before attempting to commit the corresponding HTM transaction. Only if no reader is found active, the HTM transaction can be committed; else, the writer is forcibly aborted and restarted (see Fig. 1).

This mechanism ensures that no writer can commit and materialize any changes to memory if there is any concurrent, active readers. This, in turn, guarantees that readers execute on isolated snapshots of memory, despite they can run concurrently with HTM-backed writers, as illustrated in Fig. 2, as well as with other readers.

In the above description, we have, for simplicity, omitted discussing the management of the fall-back execution path, which, we recall, is required in HTM systems to ensure termination of transactions that cannot be successfully executed in hardware (for example, when exceeding capacity limitation or executing I/O operations). As in typical HTM systems, SpRWL uses a *single_global_lock* (SGL) as fall-back synchronization method [33] in case a transaction cannot complete successfully in HTM. After some predetermined number of attempts, the transaction is executed pessimistically, after having acquired the SGL. SGL is also *subscribed* right after a hardware transaction begins, i.e., the lock's state is read and the transaction is aborted if the lock is not found free. This guarantees that if a thread activates the fall-back path and acquires the SGL, any concurrent hardware transaction is immediately aborted.

In order to ensure the correct interplay between uninstrumented readers and writers active using the SGL, readers check the SGL after flagging their own state to active, and are allowed to proceed only if the SGL is found free (see line 29). The writers that execute in the fall-back path, in their turn, have to wait for the completion of any active reader after acquiring the SGL and before executing the write critical section (see line 45). Overall, this mechanism ensures

safety by precluding any concurrency between uninstrumented readers and writers executing in the SGL.

As we will show, despite its simplicity, this base algorithm is surprisingly effective in boosting system's throughput in workloads dominated by long readers that do not fit HTM's capacity. Indeed, if one attempted to use plain HTM to elide a read critical section that does not meet the hardware capacity limitations, the reader would eventually exhaust its budget of retries using HTM and acquire the SGL fallback. This would prevent any concurrency with other readers and/or writers. Conversely, with SpRWL, readers that exceed HTM's capacity can still execute concurrently not only with other readers, but also with other writers executing in HTM, as exemplified by Fig. 2.

However, since writers are only allowed to commit using HTM in absence of concurrent readers, in read intensive workloads, this base algorithm exposes writers to the risk of starvation. More precisely, this approach can expose writers to the risk of exhausting their budget of retries in HTM, leading to frequent activations of the pessimistic fall-back path that can hinder not only the latency of writers, but also the global degree of concurrency in the system.

3.2 Scheduling Techniques

To address the above discussed shortcomings, SpRWL integrates two scheduling techniques, which we refer to as reader and writer synchronization schemes. The former imposes delays on the readers' side, in case they detect active writers, whereas the latter imposes delays to writers, if they detect active readers. The two synchronization schemes operate in a synergy, ultimately aimed to enhance SpRWL's efficiency, but they do pursue different goals.

Specifically, the reader synchronization scheme pursues a twofold goal: (i) providing fairness guarantees for the writers, by ensuring that newly readers cannot cause the abort of already active writers, and (ii) reducing the time window during which writers can find active readers. The writer synchronization scheme, conversely, stalls writers to prevent them from uselessly consuming their budget of attempts using HTM, while striving to achieve maximum concurrency with active readers.

3.2.1 Reader Synchronization. The pseudo-code of the reader synchronization scheme is reported in Alg. 2. Note that the pseudo-code illustrates only the differences with respect to the base algorithm, omitting the parts in common. This variant uses two additional shared arrays, also having one entry per each thread in the system: *clock_w*, which stores the expected end time of any currently active writer, and *waiting_for*, which is used by readers to advertise the identity of any writer they are currently waiting for.

In more detail, SpRWL samples the execution time of critical sections on a single thread² — so as to reduce measurement overhead — and computes an exponential moving average — which can be efficiently computed in an on-line fashion and allows for quickly reflecting changes in the workload characteristics³. This approach

²In order to identify different critical sections, the current prototype of SpRWL requires programmers to specify a unique identifier to the APIs used to enter/exit a critical section. This task could, however, be delegated to the compiler and made fully automatic and transparent for programmers, using, e.g., the stack trace as unique identifier of different critical sections.

³In order to estimate the expected end time of critical sections in a lightweight, yet accurate, fashion, SpRWL relies on the hardware timestamp counter, which in modern

Algorithm 1 — Basic algorithm (thread *tid*)

```

1: Global variables:
2:   state[N]  $\leftarrow$  { $\perp$ ,  $\perp$ , ...,  $\perp$ }  $\triangleright$  One state per thread
3:   gl  $\triangleright$  Global lock for HTM fallback
4: function SPRWLbasic_READ_LOCK
5:   state[tid]  $\leftarrow$  #READER  $\triangleright$  Flag active reader...
6:   MEM_FENCE  $\triangleright$  ...and make sure writers see it
7:   READER_GL_SYNC()
8: function SPRWLbasic_READ_UNLOCK
9:   MEM_FENCE  $\triangleright$  Ensure critical section reads are executed...
10:  state[tid]  $\leftarrow$   $\perp$   $\triangleright$  ...before readers are flagged as inactive.
11: function SPRWLbasic_WRITE_LOCK
12:  attempts  $\leftarrow$  0
13:  BEGIN_HTM_TX()  $\triangleright$  Start transaction
14: function SPRWLbasic_WRITE_UNLOCK
15:  if tid is executing in a HTM transaction then
16:    CHECK_FOR_READERS()
17:    TX_COMMIT  $\triangleright$  Commit the HTM transaction
18:  else
19:    RELEASE_GL()
20: function CHECK_FOR_READERS
21:  for i  $\leftarrow$  0 to N-1 do  $\triangleright$  Abort if any other thread...
22:    if i  $\neq$  tid and state[i] is #READER then  $\triangleright$  ...is an active reader
23:      TX_ABORT()
24: function WAIT_FOR_READERS
25:  for i  $\leftarrow$  0 to N-1 do  $\triangleright$  Wait for completion...
26:    if i  $\neq$  tid then
27:      wait until state[i]  $\neq$  #READER  $\triangleright$  ...of active readers
28: function READER_GL_SYNC
29:  if isLocked(gl) then
30:    state[tid]  $\leftarrow$   $\perp$   $\triangleright$  Defer to gl writer...
31:    wait until isFree(gl)  $\triangleright$  ...and wait until lock is free
32:  go to 5
33: function BEGIN_HTM_TX
34:  repeat until !isLocked(gl)  $\triangleright$  Wait until lock is free
35:    attempts ++
36:    status  $\leftarrow$  TX_BEGIN()  $\triangleright$  Begin HTM transaction
37:    if status == SUCCESS then  $\triangleright$  Normal exec. path
38:      if isLocked(gl) then  $\triangleright$  Add lock to read-set and...
39:        TX_ABORT()  $\triangleright$  ...abort Tx if lock is busy
40:    else  $\triangleright$  Branch executed upon abort of a hw tx.
41:      ABORT_HANDLER()
42: function ABORT_HANDLER
43:  if attempts > MAX_RETRIES then  $\triangleright$  Is budget over?
44:    ACQUIRE_GL()  $\triangleright$  yes...activate fallback
45:    WAIT_FOR_READERS()
46:  else
47:    BEGIN_HTM_TX()  $\triangleright$  no...retry

```

assumes that the sampling thread eventually executes all the critical sections that can ever be acquired by *any* thread. It also assumes that the measurements gathered by the sampling thread are representative for every other thread. These assumptions hold for all the applications that we have tested in our experimental study, but may not be true for applications where threads play specialized/heterogeneous roles. These scenarios could be accommodated by sampling the duration of critical sections at multiple threads, and periodically merging the statistics gathered at each thread, as, e.g., in [11, 13]. To simplify the presentation, we omit describing

CPUs provides a low-overhead, cycle-accurate time source. Further, in order to cope with programs having heterogeneous critical sections, SprWL gathers independent statistics for different critical sections.

explicitly these mechanisms in the pseudo-code and encapsulate them in the *estimateEndTime()* primitive.

The reader synchronization mechanism introduces two main changes to the base algorithm presented in Section 3.1.

First, when starting a critical section, writers advertise their existence and expected end time in the *state* and *clock_w* arrays.

Second, before entering a read critical section, readers check whether they have to first execute a wait phase (READERS_WAIT() function). In more detail, a reader inspects the *state* and *waiting_for* arrays' entries of the other threads in the system and starts a waiting phase in case (i) it finds any active writer, or (ii) any reader already waiting for an active writer ⁴.

If there are no readers already waiting, the newly arrived reader waits for the writer that is expected to complete last. It is easy to see that this ensures that newly arrived readers do not prevent already active writers from committing using HTM. If, instead, a newly arrived reader *r* detects the existence of another reader *r'* waiting for some writer *w*, *r* joins *r'* in the wait for *w*. As we will show experimentally, this policy has a relevant beneficial impact on performance at high thread counts, since it tends to synchronize the starting time of readers.

If readers are likely to begin simultaneously, the time window in which there is any active reader in the system is globally reduced, increasing the chances for writers to be able to execute using HTM. Further, it tends to reduce the average reader latency, by allowing them to start sooner.

3.2.2 Writer Synchronization. The writer synchronization scheme, whose pseudo-code is reported in Alg. 3, aims at sheltering writers from the risk of incurring repeated aborts due to already active readers, while striving to jointly maximize the concurrency achievable by writers and readers.

In a nutshell, these goals are pursued by delaying the start time of a writer that executes in HTM, in case it encounters any active reader, by the shortest time possible that still allows the writer to commit successfully. This is achieved by timing the start of the writer so that the execution of its write critical section completes “shortly after” the last reader completes. This way, not only writers are guaranteed (or, at least, are more likely) not to have to abort due to a concurrent reader; they also maximize the period of time during which their execution overlaps with that of concurrent readers.

More in detail, a writer first attempts, optimistically, to execute in HTM immediately, i.e., avoiding the writer synchronization phase. Note that the pseudo-code in Alg. 3 only reports the parts that differ with respect to Alg. 2 and, as such, omits specifying the pseudo-code for entering a write critical section, which is not modified by the writer synchronization scheme.

If a writer, executing in a HTM transaction, incurs an abort due to an active reader, it determines the maximum end time of any active reader. To this end, with the writer synchronization scheme, also readers advertise their expected end time, right after having

⁴It is worth noting here that a reader could be spared from waiting for already active HTM-based writers that are expected to complete after it does — as these writers would not be hampered by the reader. In SprWL we choose not to implement directly this mechanism (which would require writers to estimate and advertise their expected duration), as we achieve the same goal (i.e., avoid blocking a reader in presence of already active HTM-based writers that are likely to finish earlier than it does) via another optimization, i.e., attempting to execute readers using HTM first. We discuss this optimization in Section 3.4.

Algorithm 2 – Reader synchronization (thread tid)

```

1: Global variables:
2: ... ▷ As in Alg. 1
3:  $clock\_w[N] \leftarrow \{\perp, \perp, \dots, \perp\}$  ▷ Writer exp. end time
4:  $waiting\_for[N] \leftarrow \{\perp, \perp, \dots, \perp\}$  ▷ Reader is waiting for
5: function  $SpRWL^{wSync\_WRITE\_LOCK}$ 
6:    $state[tid] \leftarrow \#WRITER$ 
7:    $clock\_w[tid] \leftarrow estimateEndTime();$ 
8:    $SpRWL^{basic\_WRITE\_LOCK}()$  ▷ Execute basic alg.
9: function  $SpRWL^{wSync\_WRITE\_UNLOCK}$ 
10:   $SpRWL^{basic\_WRITE\_UNLOCK}()$ 
11:   $state[tid] \leftarrow \perp$ 
12: function  $SpRWL^{wSync\_READ\_LOCK}$ 
13:   $READERS\_WAIT()$  ▷ Sync with writers
14:   $SpRWL^{basic\_READ\_LOCK}()$  ▷ Execute basic alg.
15: function  $READERS\_WAIT()$ 
16:   $wait \leftarrow -1$ 
17:   $max\_wait \leftarrow 0$ 
18:  for  $i \leftarrow 0$  to  $N-1$  do ▷ Wait for longest ...
19:    if  $state[i] = \#WRITER \wedge clock\_w[i] > max\_wait$  then ▷ ...active writer ...
20:       $max\_wait \leftarrow clock\_w[i]$ 
21:       $wait \leftarrow i$ 
22:    else if  $waiting\_for[i] \neq \perp$  then ▷ ...or join..
23:       $wait \leftarrow waiting\_for[i]$  ▷ ...a waiting reader.
24:      BREAK
25:  if  $wait \neq -1$  then ▷ Wait until last writer finishes.
26:     $waiting\_for[tid] \leftarrow wait$  ▷ Advertise wait phase.
27:    wait until  $state[wait] \neq \#WRITER$ 
28:     $waiting\_for[tid] \leftarrow \perp$ 

```

completed the reader synchronization and before starting execution (see function $SpRWL^{wSync_READ_LOCK}()$) using a dedicated global array ($clock_r$). In order to overlap its execution with that of already active readers, a writer adjusts its waiting phase so that it is expected to complete δ cycles after the last active reader. δ is a parameter, whose tuning allows to trade-off the degree of concurrency between writers and readers (which can be increased by setting δ close to 0) for the likelihood that writers can commit successfully (which can be increased by setting δ close to the expected duration of the writer). In SpRWL, we use half the expected duration of the writers as default value for δ , which we have observed to provide the best over-all performance based on preliminary experimentations.

Note that, to preserve fairness, writers maintain their writer flag active while waiting for a reader: this guarantees that writers have a chance to commit successfully, as new coming readers will wait for them thanks to the reader synchronization scheme.

3.3 Correctness and Fairness

This section elaborates on the correctness and fairness guarantees ensured by SpRWL. As in typical HTM systems, the correctness (more precisely safety) criterion ensured by SpRWL is opacity [22].

The correctness of readers is ensured by two mechanisms: (i) the check performed by writers using HTM before committing, and the wait performed by writers executing in the SGL path before they begin. Recall that HTM guarantees that memory writes issued in a HTM transaction are hidden from concurrent threads during transaction's execution and will appear atomically only upon a successful commit. Therefore, for a HTM writer to break the consistency of a reader, it must commit throughout the course of the

Algorithm 3 – Writer synchronization (thread tid)

```

1: Shared variables:
2: ... ▷ As in Alg. 2
3:  $clock\_r[N] \leftarrow \{\perp, \perp, \dots, \perp\}$  ▷ per thread
4: function  $SpRWL^{wSync\_READ\_LOCK}$ 
5:   $READERS\_WAIT()$  ▷ Sync with active writers
6:   $clock\_r[tid] \leftarrow estimateEndTime()$  ▷ Advertise end time
7:   $SpRWL^{basic\_READ\_LOCK}()$  ▷ Execute basic alg.
8: function  $ABORT\_HANDLER^{wSync}$ 
9:  if  $attempts > MAX\_RETRIES$  then ▷ is budget over?
10:     $ACQUIRE\_GL()$  ▷ activate fallback
11:     $WAIT\_FOR\_READERS()$  ▷ as in Alg. 1
12:  else ▷ Can still retry in HTM
13:    if  $abort\_cause$  is  $reader\_abort$  then ▷ Sync with active readers
14:       $WRITER\_WAIT()$ 
15:       $BEGIN\_HTM\_TX()$ 
16: function  $WRITER\_WAIT()$ 
17:   $wait \leftarrow 0$ 
18:  for  $i \leftarrow 0$  to  $N-1$  do ▷ Store the end time...
19:    if  $state[i] = \#READER$  then ▷ ...of the last reader...
20:      if  $clock\_r[i] > wait$  then ▷ ...to finish...
21:         $wait \leftarrow clock\_r[i]$  ▷ ...in wait.
22:        ▷ Delay the writer to end  $\delta$  cycles after the last reader.
23:   $wait -= estimateDuration() - \delta$ 
24:  wait until  $RDTS() \geq wait$ 

```

reader's execution. However, this is impossible, since readers advertise their presence using a memory fence before starting, and clear their status only after issuing another fence, which guarantees that any writer using HTM will detect the reader's presence before committing and, thus, abort.

For analogous reasons, writers that activate the fallback path are guaranteed to detect and wait for any active reader before starting executing (line 45, Alg. 1). Further, readers starting after a writer has acquired the SGL are guaranteed to detect the writer's presence — acquiring a lock entails a full memory fence — and are forced to wait for its completion (line 28, Alg. 1). Overall, these two synchronization mechanisms prevent any concurrency between an uninstrumented reader and writers executing in the SGL.

Let us now discuss the liveness and fairness properties of SpRWL. We note that the synchronization scheme employed by SpRWL to avoid concurrency between uninstrumented readers and writers executing in the SGL cannot lead to deadlocks. In fact, in order for a writer to wait for a reader (line 27, Alg. 1), the reader must be advertising its state as active. However, before starting waiting for any writer (line 31, Alg. 1), readers first reset their state flag (line 30, Alg. 1). This precludes the possibility of mutual waits/deadlocks between writers and readers.

Further, since writers wait for readers only after having acquired the SGL, and they wait for a given reader at most once, SpRWL guarantees that a writer that activates the fallback path cannot wait indefinitely (line 45, Alg. 1), even in presence of a constant stream of readers. We note that SpRWL algorithm may, theoretically, cause readers to wait indefinitely in presence of a constant stream of writers executing in the SGL. This issue can be avoided by implementing the SGL via a versioned lock that is incremented each time writers acquire it. The first time a reader finds the SGL busy, it registers and advertises in a per-thread variable the current lock's version number and only waits for writers (lines 30-31, Alg. 1)

if the lock's version number is not larger than the one the reader first observed. Writers, in their turn, acquire the SGL and increase the lock's version, but start executing only if there are no readers waiting with a smaller version number. We omitted this optimization since in read-dominated workloads, i.e., the ones targeted by read-write locks and SpRWL, the probability for a reader to starve due to a continuous stream of writers is expected to be quite low.

Finally, as already mentioned, the base version of SpRWL algorithm can cause writers that execute in HTM to be overrun by newly arrived readers. This leads to a violation of fairness for HTM-backed writers, which can lead to frequent and unnecessary activations of the fallback path. This issue is tackled by the reader synchronization scheme, which guarantees that if a reader r starts after a writer w , r will only start after w completes executing.

3.4 Optimizations

We start by describing how to optimize the memory utilization of SpRWL and then introduce a set of relevant performance-oriented optimizations that we integrated in SpRWL, but omitted while presenting its pseudo-code, to simplify presentation.

The presented pseudo-code assumes that each thread maintains four variables: *state*, *clock_w*, *clock_r* and *waiting_for*. A possible optimization (not implemented in our current prototype), aimed to minimize memory consumption, would consist in encoding these 4 variables using a single 64 bit word, which we call *clock*. *state* can be encoded by setting *clock* to 0 when the thread is neither a reader, nor a writer. Else, *clock*'s MSB can be used to distinguish whether the thread is in a read or a write critical section. The subsequent k MSBs of *clock* can encode the thread id stored in *waiting_for*. Considering that $k < 10$ bits would suffice to support of up to 1024 threads, this scheme would allow to use more than 50 bits to encode the value of *clock_w/clock_r* (corresponding to several days assuming timestamp counters with nanosecond granularity).

The first performance optimization consists in letting readers, just like writers, attempt to first execute speculatively using HTM, and to use the uninstrumented execution mode as a fall-back patch that is only activated if the transaction experiences an abort due to a capacity exception or if it exhausts its budget of retries. This optimization allows SpRWL to be competitive with a plain HTM-based approach, even in workloads where readers fit in HTM, while introducing negligible overheads in workloads with long readers. Furthermore, this optimization allows readers that finish before already active, HTM-based, writers to execute concurrently with these writers, instead of waiting unnecessarily for them (refer to Section 3.2.1). In this case, being shorter than HTM-based writers, readers are also likely to fit in HTM. In fact, as already highlighted by previous works in the HTM domain, e.g., [25], after experiencing an abort, transactions tend to re-execute significantly faster than in their first execution as they are more likely to incur higher cache hit rates. We note that, in principle, one could use the online statistics gathered by SpRWL to predict a priori whether certain readers are likely to incur capacity exceptions when using HTM, and run them directly using the uninstrumented execution path. In SpRWL, though, we have opted for adopting the previously described approach, because of its simplicity and robustness.

A second optimization consists in employing the Scalable NonZero Indicators (SNZI) [16] to allow writers to detect the existence of concurrent readers, at commit time. The previously presented pseudo-code, in fact, forces the writers to incur a cost linear in the number of threads in the system, given that it requires writers to scan the entire *state* array. Since the reading of the *state* array takes place from within the scope of a HTM transaction, this results in an increase of the memory footprint of the encompassing hardware transaction and in a reduction of the cache capacity effectively available for writers. Also, the longer it takes for a writer to scan the *state* array, the more it is exposed to the risk of aborting due to a conflict with a concurrent reader that changes its own entry. The two issues above can be avoided by having readers advertise their existence via SNZI, since SNZI can execute queries in constant time, by reading a single counter. This comes, though, at the cost of an increase in the overhead incurred by readers to advertise the start/end of their execution, as updates to SNZI have, roughly, logarithmic complexity in the number of threads.

Finally, a relevant optimization consists in avoiding that, while executing the reader synchronization scheme, readers spin on the entry of the *state* array associated with the writer they are waiting (Alg. 2, line 27). This polling mechanism generates unnecessary load for the memory subsystem, imposing non-negligible overhead. This overhead can be strongly reduced by having the reader wait until the expected duration of the writer by using the hardware timestamp counter, which can be read via CPU registers.

4 EVALUATION

In this section we evaluate SpRWL against a number of RLock implementations that use either speculative or pessimistic techniques. The experimental study is conducted on two HTM-enabled processors (by Intel and IBM) characterized by different capacity limitations, and encompasses a large set of synthetic and complex benchmarks/real-life applications.

More specifically, we consider the following baseline solutions: (i) pthread's RLock (RWL), (ii) Big Reader Lock [8] (BRLock), (iii) plain transactional lock elision (TLE) and (iv) Hardware Read-Write Lock Elision (RW-LE) [17], which, unlike SpRWL, relies on specific features of IBM POWER8 processor (see Section 2) and, as such, cannot be tested on Intel platforms.

For all HTM based solutions, including SpRWL, we used a retry policy that attempts a transaction 10 times in hardware before activating the fallback path, except upon capacity aborts, in which case the fallback path is immediately activated. The only exception is RW-LE, where we used a budget of attempts equal to 5 for update transactions executing using ROTs — the same policy used by the authors of RW-LE in their evaluation.

In the following, when reporting the abort data, we distinguish four abort causes: aborts due to *conflicts* and *capacity*; *explicit* aborts, which arise when transactions explicitly request to abort, either because the fall-back lock is found to be acquired or because the application's logic demands so; *reader*, namely aborts incurred in SpRWL by writers, when they encounter some active reader during their commit phase. When reporting commit data, we will distinguish the following alternatives: *HTM*, *ROT*, *GL* and *Unins*, which correspond to critical sections executed (successfully) using HTM,

ROT, the pessimistic fallback path, and the readers' uninstrumented execution path presented in Section 3.1, respectively.

The evaluation is performed using two HTM-equipped CPUs, namely Intel Broadwell and IBM POWER8. For Intel, we used a dual socket Intel Xeon E5-2648L v4 processors with 28 cores and up to 56 hardware threads running Ubuntu 16.04 with Linux 4.4. For IBM, we used a POWER8 8284-22A processor that has 10 physical cores, with 8 hardware threads each running Fedora 24 with Linux 4.7.4. It should be noted that, due to their architectural differences, these Intel's and IBM's processors are faced with very different capacity limitations. Specifically, POWER8 processors have a 8KB capacity for both memory reads and writes, whereas the capacity of Broadwell is 22KB for writes and 4MB for reads [32].

The source code [2] was compiled with -O2 flag using GCC 5.4.0 and 6.2.1 for the Intel and IBM platforms, respectively. Thread pinning was used to pin a thread per core at the beginning of each run for all the baselines, and threads were distributed evenly across the available CPUs. All the results reported in the following are obtained as the average of at least 5 runs.

First, we conduct a sensitivity analysis aimed to stress different aspects of SpRWL's design and optimizations, such as the relevance of the reader and writer synchronization schemes, and using SNZI vs per thread flags to track the status of active readers. To this end, we rely on a micro-benchmark, based on a concurrent hashmap, which allows us to control in a precise way the workload's characteristics.

Finally, we test SpRWL using a port of TPC-C [36] for in-memory databases [15, 35].

4.1 Sensitivity Analysis

In this section we describe the results of a sensitivity analysis based on a micro-benchmark consisting of a hashmap protected using a single global lock, that offers three operations: lookup, insert and delete. The three operations access items uniformly at random and the ratio of lookups to inserts and deletes controls the percentage of update transactions performed. We protect each insert and delete operation with a write lock and pre-populate the hashmap so that update operations fit the capacity limitations of the underlying HTM implementation. To this end, we use an hashtable with 5000 buckets and populate it with 8 million and 3 million items for the Broadwell and POWER8 processor, respectively.

We control the size of readers by performing either 1 or 10 lookups within a single read critical section. In the latter case, lookups exceed the HTM capacity — thus, allowing to quantify the gains that SpRWL can achieve by supporting uninstrumented readers. In the former case, instead, lookups can be successfully executed in HTM, which nullifies the benefits of SpRWL with respect to plain HTM-based lock-elision — thus, allowing us to evaluate SpRWL's overhead in an unfavourable workload.

In this section, when running SpRWL, we enable all the scheduling techniques and optimizations presented in Section 3, with the exception of the SNZI-based readers' tracking mechanism, which will be investigated in detail in Section 4.1.2.

In Figure 3, we show the results for executing readers that perform ten lookup operations within a single read critical section. On the left we report data for Broadwell and on the right for POWER8.

From top to bottom, for each architecture, we report the throughput, abort rate, breakdown of commit modes, as well as the readers' and writers' latency. From left to right, we increase the percentage of update operations/critical sections from 10% to 50% then 90%.

As mentioned, in this workload, readers do not normally fit in HTM transactions, as confirmed by the aborts breakdown which shows almost 50% capacity abort rate with a single thread. The high percentage of capacity aborts due to large readers leads to the frequent activation of the fallback path, as reflected in the commit breakdown plot. Consequently, TLE achieves modest scalability in both the 10% and 50% update workloads.

Thanks to its ability to execute readers without instrumentation (see commit breakdown plot), instead, SpRWL exhibits a much better scalability level, achieving, in the 10% updates workloads, speedups versus TLE up to $16\times/8\times$ on Broadwell/POWER8, respectively. When moving to workloads with higher percentage of update operations — which fit HTM's capacity — we notice that SpRWL still outperforms TLE by up to $5\times/2\times$ on Broadwell/POWER8 in the 50% workload. Even in a 90% workloads, SpRWL is still capable of yielding $2\times$ higher throughput than TLE on Broadwell and marginally higher throughput on POWER8. The ability of SpRWL to scale more on Broadwell as compared with POWER8 is related to the fact that our Broadwell machine supports up to 28 threads without hyper-threading; conversely, with more than 10 threads, on POWER8, multiple (up to 8) hardware threads start sharing the same physical cores, which reduces their effective capacity and their ability to execute write critical sections in HTM.

When compared with pessimistic solutions, i.e., BRLock and RWLock, similar trends are observed: SpRWL achieves up to $4\times/2.5\times$ throughput gains compared to the best performing of the two (BRLock) on Broadwell/POWER8 in the 10% update workloads. In workloads with higher percentage of update operations, we can see that SpRWL achieves higher gains, due to its ability to execute writers concurrently using HTM.

RW-LE, which also executes readers without instrumentation on top POWER8, is able to achieve similar performance to SpRWL up to $8/4$ threads in the 10/50% updates workloads. However, it does not scale beyond that point. This behavior can be explained by the commits breakdown, which shows that RW-LE executes all update transactions as ROTs from that point on. Although ROTs allow concurrent readers, unlike the pessimistic lock used by SpRWL, before ROTs can be committed writers need to execute a quiescence phase to wait for the completion of any active reader. In a workload with long readers, this leads writers to incur significant overheads, as highlighted by the writers' latency plot. By executing writers in the SGL, SpRWL avoids this issue, reducing the average writers' latency by more than 2 orders of magnitude lower latency at 32 threads — the point where SpRWL achieves the highest throughput gains with respect to RW-LE — at the cost of a (relatively) much smaller increase of the readers' latency (approximately $3\times$).

Figure 4 shows the throughput and breakdown of aborts and commits when readers perform a single lookup operation in the read lock, thus fitting in HTM transactions. As expected, since also update operations can be successfully executed using hardware transactions, TLE is the overall best performing variant, achieving the highest throughput across all workloads and architectures. Indeed, by looking at the commits breakdown, we can notice the

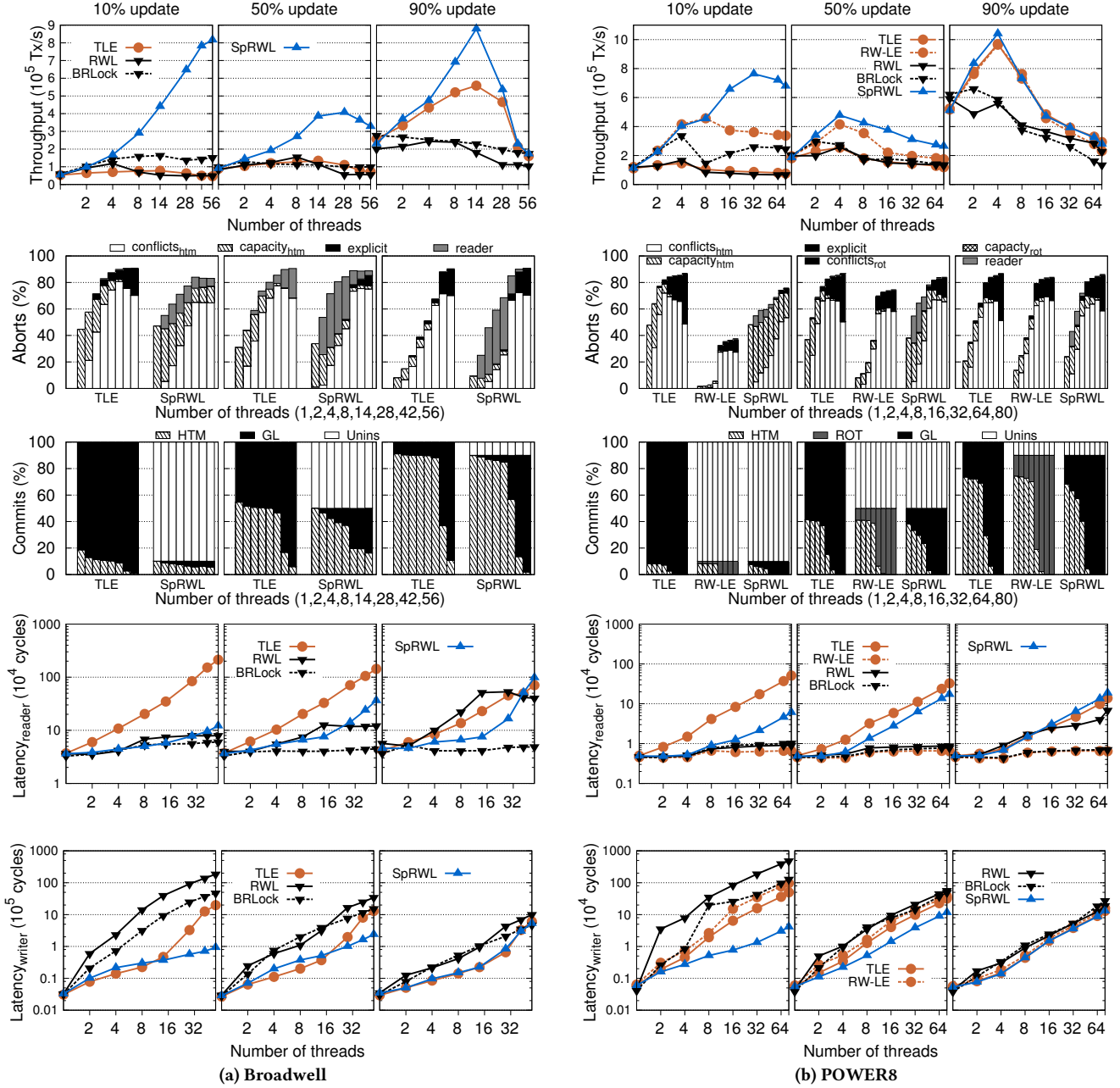


Figure 3: Hashmap: readers execute 10 lookups, writers execute 1 insert/delete. Broadwell (left) and POWER8 (right).

ability of TLE to commit almost all transactions using HTM, except for very high thread counts (due to the coexistence of multiple hardware threads on the same physical core).

Nevertheless, even in this unfavorable workload, SpRWL achieves performance comparable with TLE. In fact, we have experimentally verified (data not reported due to space constraints) that when faced with workloads composed exclusively by read-only critical sections that fit in HTM, TLE and the uninstrumented execution mode of SpRWL achieve very similar throughput levels. TLE does attain highest peak throughputs up to 30% higher

than SpRWL (on Intel 50% update), as it avoids the overheads of the additional, software-based, synchronization mechanisms that SpRWL imposes on the writer's side (e.g., checking for concurrent readers and possibly incurring spurious aborts). However, the average throughput across all thread counts of the two solutions is, on average, 36% higher for SpRWL. At high thread counts and read-dominated workloads, in fact, readers using TLE start incurring capacity exceptions, which increases the frequency of acquisition of the pessimistic fall-back path and cripples performance, a problem that SpRWL avoids by allowing readers to execute concurrently and

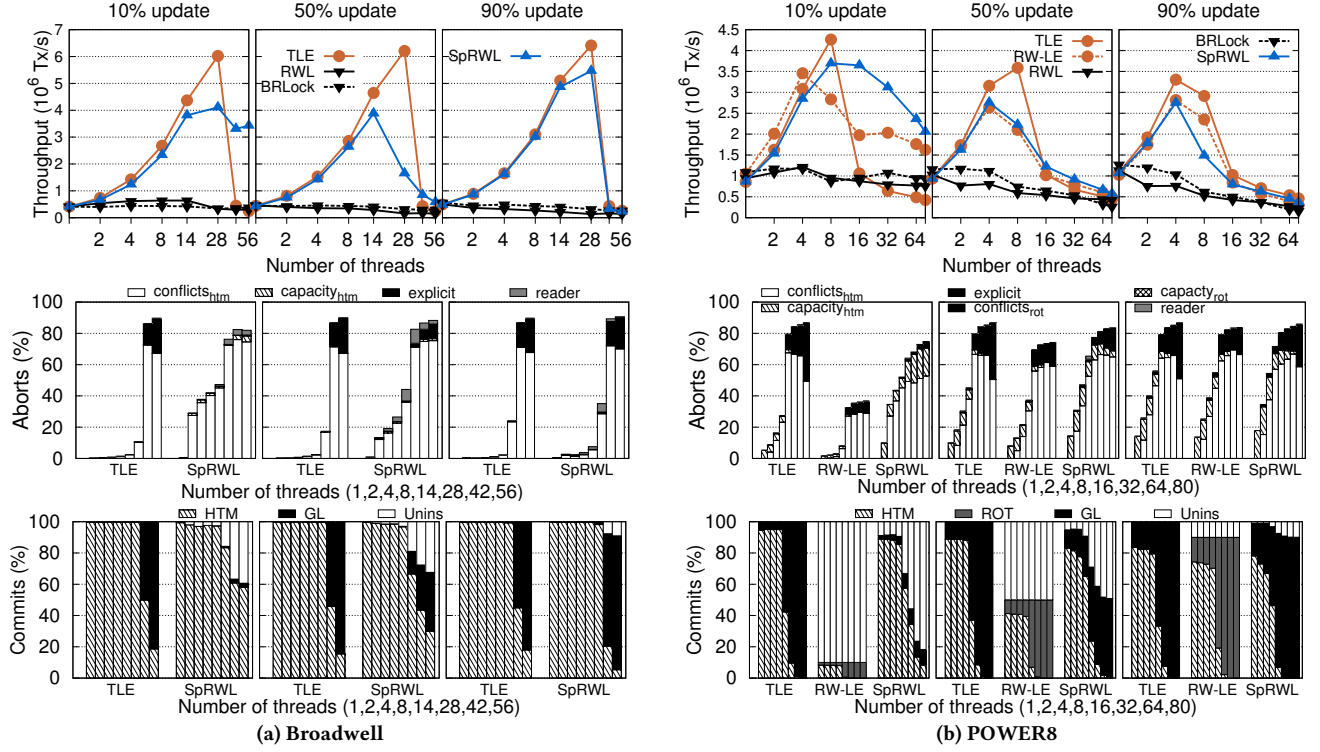


Figure 4: Hashmap: readers execute 1 lookup, writers execute 1 insert/delete. Broadwell (left) and POWER8 (right).

uninstrumented. Further, at low thread counts, SpRWL can commit a percentage of HTM transactions similar to the one achieved by TLE, avoiding most of the overheads it incurs to support the safe execution of uninstrumented readers. This is possible thanks to SpRWL’s policy, presented in Section 3.4, which first attempts, in an optimistic fashion, to execute readers using HTM.

RW-LE, which is the other hardware-based system that executes readers without instrumentation, follows trends similar to SpRWL. However, it pays a higher penalty in terms of performance when compared with either SpRWL or TLE. Even though RW-LE always commits update critical sections either as HTM or ROTs, the fact that ROTs are serialized and both HTM and ROT have to wait for active readers (that can fit in HTM) limit its scalability.

When comparing with the pessimistic variants, RWLock and BRLock, we can clearly see their inability to scale. This can be mainly related to the high cost they impose relative to the small size of the critical sections they protect and the fact that HTM allows concurrent writers.

4.1.1 Impact of Scheduling. In this section we aim at assessing the impact on performance of the different scheduling techniques employed by SpRWL. To this end we developed several variants of SpRWL, which we obtained by selectively disabling different scheduling mechanisms:

- NoSched: the base version of SpRWL, which does not employ any scheduling technique, described in Section 3.1.
- RSync: the version of SpRWL presented in Section 3.2.1, in which readers wait for the active writer that is predicted to complete last,

if no readers are already waiting for a writer, and that otherwise makes newly arrived readers join already waiting ones.

- RWait: A variant of Rsync, in which readers simply wait for the active writer that is predicted to complete last, but do not join other awaiting readers;

For this evaluation we used the hashmap micro-benchmark, with the same configuration and initial population as in the previous study. The workload constitutes 10% write critical section, which execute an individual update operation each. Each lookup operation, instead, executes 10 lookups per critical section. Due to space restrictions, we only report the results for the Broadwell architecture, but results on POWER8 exhibit similar trends.

The results obtained (Fig.5) show that even the base version of SpRWL shows significant throughput gains with respect to HTM. However, at high thread counts, the amount of concurrent read transactions increases, reducing the time window during which updates can commit. As this happens, update transactions fall-back more frequently, hindering performance.

The RWait policy brings some noticeable gains, both in terms of throughput and writer latency, at high thread counts. This is due to the fact that update transactions, executed in HTM, are no longer overrun by newly arriving readers, which now wait for already active updates to commit before starting.

RSync shows improvements compared to RWait since, as mentioned in 3.2, it allows read transactions to begin sooner, reducing reader latency, and to synchronize their start time, narrowing the time window during which update transactions are forced to abort

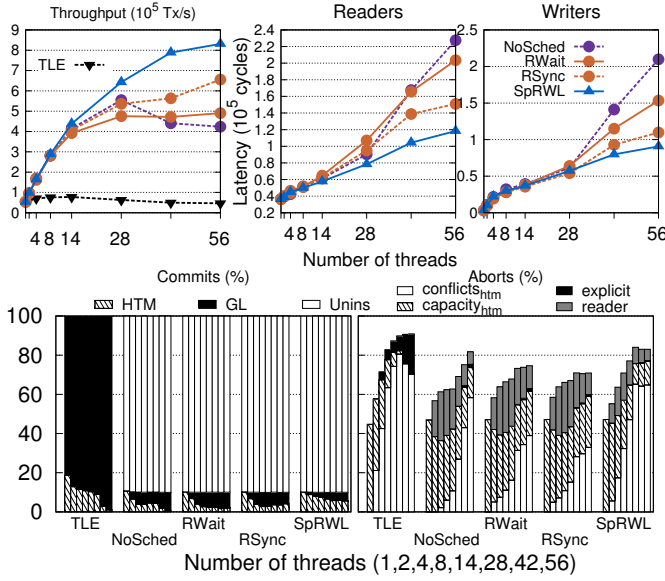


Figure 5: SpRWL variants: readers execute 10 lookups, writers execute 1 insert/delete. 10% updates on Broadwell.

(due to the existence of some active reader). This leads to a reduction of the abort rates and a decrease of the writer latency compared to RWait, yielding, ultimately, up to 30% throughput gains.

Finally, the comparison of RSync and SpRWL highlights additional gains of the writer synchronization scheme, which, especially on Intel, reduces significantly the aborts incurred by writers due to concurrent readers, enhancing their likelihood of committing successfully in HTM and yielding a further 30% improvement of the peak throughput. Overall, this study provides quantitative evidence on the synergistic contribution and actual relevance of the scheduling techniques that SpRWL employs.

4.1.2 Reader Tracking Scheme. We now evaluate the effects of employing the SNZI-based reader tracking mechanism, described in Section 3.4. The use of SNZI allows writers to determine more efficiently (a single memory lookup) whether there is any concurrent active reader, sparing them from inspecting the *state* array, whose size grows linearly with the number of threads in the system. This comes, though, at the cost of an increased overhead for readers, which grows logarithmically with the number of threads in the system.

In order to evaluate this trade-off, we vary the size of the readers by increasing the number of lookup operations they perform and consider a workload with 50% updates. Figure 6 reports the results obtained using 80 threads on POWER8 (where the impact of using SNZI is more relevant, since POWER8 has a more restricted capacity than Broadwell). The experimental data shows that the use of SNZI can yield significant throughput gains, up to $\sim 6\times$, with long readers, but also that it imposes large overheads, up to $6\times$, with short readers. This is a consequence of the larger overhead that SNZI imposes to readers, which can be amortized by long readers, but plays a dominant role with short readers.

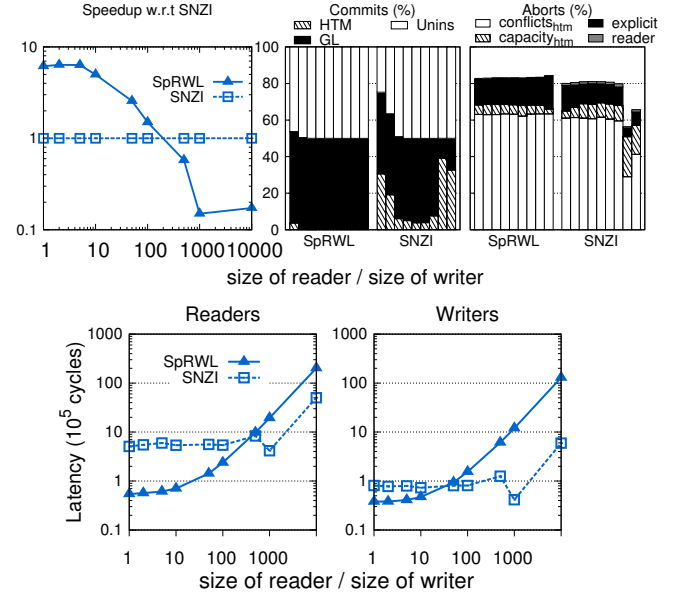


Figure 6: Reader tracking scheme: Hashmap, 50% updates, while varying the size of readers at 80 threads on POWER8.

It is interesting to note that, for large readers, not only is the writer latency reduced (as expected, since writers incur less overhead to verify the existence of readers), but also the reader latency is smaller, despite reader incur higher costs with SNZI. This is explicable considering that the reader synchronization forces readers to wait for active writers. Thus, the reduction of the writer latency enabled by SNZI benefits, indirectly, also the readers' latency, by reducing the average time readers spend waiting for writers.

Overall, this experimental data suggests that the effectiveness of the SNZI-based reader tracking scheme is strongly workload dependent, and that there exists a strong correlation with the readers' size.

4.2 TPC-C

TPC-C is an OLTP benchmark that emulates a warehouse supplier application. TPC-C uses five different types of transactions, with very diverse profiles, including long read-only transactions, long and contention-prone vs. short and almost contention-free update transactions. We used a C++ port of the TPC-C designed to operate on in-memory data structures [36], which we adapted to execute read-only/update transactions as read/write critical sections of a single RWLock. We use this benchmark with the following transactional mix (which abides by the original TPC-C specifications and encompass the whole set of transaction profiles): Stock-level, 31%, Delivery, 4%, Order Status, 4%, Payment, 43%, and New Order, 18%. Throughout the experiment, we set the number of warehouses equal to the maximum number of threads used on each platform.

Although this workload generates only 35% read-only transactions, Figure 7 shows that SpRWL is capable of achieving remarkable throughput gains — up to $4\times/2\times$ on Broadwell and POWER8, respectively, compared to the second best baseline (TLE on Broadwell and RW-LE on POWER8). This can be explained by looking

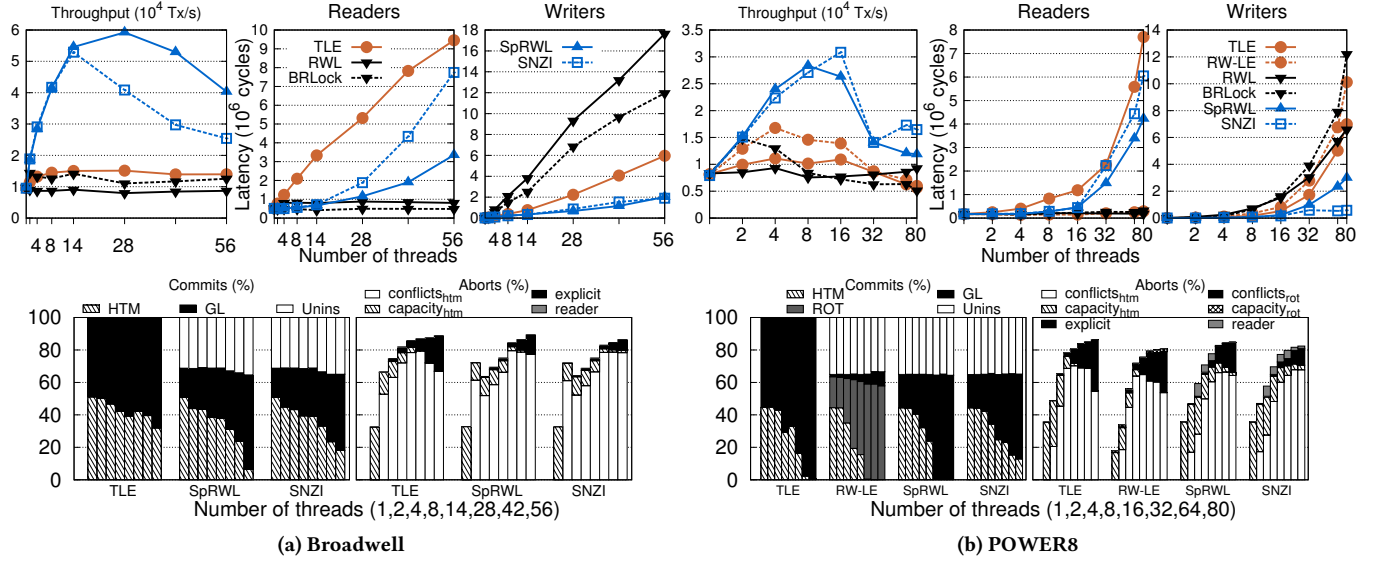


Figure 7: TPC-C. Mix comprising the following transaction profiles: Stock-level, 31%, Delivery, 4%, Order Status, 4%, Payment, 43%, and New Order, 18%. Broadwell (left) and POWER8 (right).

at the commits plots where SpRWL is capable of committing $\sim 70\%$ of update transactions in hardware up to 28/8 threads on Broadwell/POWER8, which allows more parallelism compared to pessimistic techniques. Compared to HTM-based techniques, we can see that although TLE commits similar percentage of transactions in hardware, its inability to elide long read critical sections prevent it from scaling. As for RW-LE, despite the fact that it manages to commit all update transactions either as HTM or ROTs while executing uninstrumented readers, it incurs large writer latencies: also in this case, this is due to RW-LE's quiescence phase, which forces writers to wait for any active reader.

In this setting, both SpRWL variants achieve reader latencies on par with the most competitive baselines up to 28/16 threads on Broadwell/POWER8, while reducing writer latency by $\sim 4\times/2\times$, respectively. Again, on POWER8, we can notice that SNZI can play a role in optimizing SpRWL's performance by scaling up to 16 threads instead of 8. This can be explained by considering that SNZI reduces the footprint of writers' transactions, thus increasing the likelihood that these can commit using HTM (see commits breakdown plot). Conversely, on Broadwell, the performance of the SNZI variant deteriorates starting from 14 threads. We argue that this is due to the fact that, on the Intel's processor, the costs incurred by readers due to the use of SNZI is much more amplified than on POWER8 and outweighs by far the gains deriving from the increased ability of writers to commit using HTM.

5 CONCLUSIONS AND FUTURE WORK

This work introduced SpRWL, a HTM-based implementation of RWLock that allows readers to execute outside the scope of hardware transactions, thus, sparing them from any HTM-related limitation. SpRWL preserves the correctness of uninstrumented readers in presence of concurrent writers via a simple, yet very effective, synchronization mechanism, whose efficiency is further amplified via ad-hoc scheduling techniques. A key novel aspect of SpRWL

is that it assumes a standard transaction demarcation API that is universally supported by any HTM implementation.

Via an extensive experimental study, we have shown that SpRWL can achieve significant performance gains with respect to RWLock implementations based not only on pessimistic techniques or on plain hardware lock elision schemes, but also with respect to recent RWLock implementations that rely on non-standard HTM features, which are currently supported only by IBM POWER8 and POWER9 processors.

Overall, SpRWL advances the state of the art on HTM-based RWLock elision techniques not only due to its superior performance, but also by broadening the scope of applicability of these techniques to *any* HTM implementation.

This work opens several interesting avenues for future research. Our experimental study highlighted that using a scalable counter, such as SNZI, can be quite beneficial in certain scenarios, but also hamper performance in others. We argue that this finding motivates two research directions: one investigating the use of self-tuning techniques to automatically enable/disable the use of SNZI; another studying alternative scalable counter algorithms and implementations specifically tailored to maximize SpRWL's performance. Another interesting research line is the investigation of alternative scheduling policies for regulating the concurrent execution of readers and writers.

ACKNOWLEDGMENTS

This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014. We wish to thank the anonymous reviewers and our shepherd, Dr. Aleksandar Dragojevic, for their insightful comments and Prof. Pascal Felber for providing access to the POWER8 machine used in the experimental study.

REFERENCES

- [1] Posix.1-2008. <http://pubs.opengroup.org/onlinepubs/9699919799/>, 2013.
- [2] SpRWL open source implementation. <https://github.com/shadyalaa/SpRWL>, 2018.
- [3] ARBEL, M., AND ATTIA, H. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2014), PODC '14, ACM, pp. 196–205.
- [4] BLUNDELL, C., LEWIS, E. C., AND MARTIN, M. M. K. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters* 5, 2 (Feb 2006).
- [5] BRANDENBURG, B. B., AND ANDERSON, J. H. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *2009 21st Euromicro Conference on Real-Time Systems* (July 2009), pp. 184–193.
- [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] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scalable address spaces using RCU balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 199–210.
- [8] CORBET, J. Big reader locks. <https://lwn.net/Articles/378911/>, 2016.
- [9] COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. Concurrent control with “readers” and “writers”. *Communications of ACM* 14, 10 (Oct. 1971), 667–668.
- [10] DICE, D., HARRIS, T. L., KOGAN, A., LEV, Y., AND MOIR, M. Hardware extensions to make lazy subscription safe. *CoRR abs/1407.6968* (2014).
- [11] 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.
- [12] DIEGUES, N., AND ROMANO, P. Self-tuning Intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 209–219.
- [13] DIEGUES, N., ROMANO, P., AND GARBATOV, S. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2015), SPAA '15, ACM, pp. 224–233.
- [14] 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.
- [15] E. JONES. Stand-alone in-memory TPC-C implementation. <https://github.com/evanj/tpccbench>, 2017.
- [16] ELLEN, F., LEV, Y., LUCHANGCO, V., AND MOIR, M. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 13–22.
- [17] FELBER, P., ISSA, S., MATVEEV, A., AND ROMANO, P. Hardware read-write lock elision. In *Proceedings of the 11th European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 34:1–34:15.
- [18] GOEL, B., TITOS-GIL, R., NEGI, A., MCKEE, S. A., AND STENSTROM, P. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (May 2014), pp. 615–624.
- [19] GUERRAOLI, R., HERLIHY, M., KAPALKA, M., AND POCHON, B. Robust Contention Management in Software Transactional Memory. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOO'05)* (2005).
- [20] GUERRAOLI, R., HERLIHY, M., AND POCHON, B. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing* (2005), vol. 3724 of *DISC '05*, pp. 303–323.
- [21] GUERRAOLI, R., HERLIHY, M., AND POCHON, B. Toward a theory of transactional contention managers. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2005), PODC '05, ACM, pp. 258–264.
- [22] GUERRAOLI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPOPP '08, ACM, pp. 175–184.
- [23] ISSA, S., FELBER, P., MATVEEV, A., AND ROMANO, P. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. In *LIPICs 31st International Symposium on Distributed Computing* (Dagstuhl, Germany, 2017), vol. 91 of *DISC '17*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 28:1–28:16.
- [24] ISSA, S., ROMANO, P., AND BRORSSON, M. Green-CM: Energy efficient contention management for transactional memory. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP)* (Sept 2015), pp. 550–559.
- [25] IZRAELEVITZ, J., XIANG, L., AND SCOTT, M. L. Performance improvement via always-abort htm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Sept 2017), pp. 79–90.
- [26] LE, H. Q., GUTHRIE, G. L., WILLIAMS, D. E., MICHAEL, M. M., FREY, B. G., STARKE, W. J., 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.
- [27] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 21–35.
- [28] 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.
- [29] 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.
- [30] MCKENNEY, P. E., AND WALPOLE, J. What is RCU, fundamentally? <https://lwn.net/Articles/262464/>, 2007.
- [31] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1991), PPOPP '91, ACM, pp. 106–113.
- [32] 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.
- [33] 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.
- [34] SCHERER, III, W. N., AND SCOTT, M. L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2005), PODC '05, ACM, pp. 240–248.
- [35] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (2007)*, VLDB '07, VLDB Endowment, pp. 1150–1160.
- [36] TPC COUNCIL. Transaction Processing Performance Council, TPC BENCHMARK™ C. Revision 5.11. February 2010.
- [37] 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.