Speculative Read-write Locks

Tiago João dos Santos Lopes

Abstract

We introduce SpRWL, a Big-reader focused HTM system which uses synchronization to achieve higher performance than typical TM systems, at the cost of reader latency. We explain how its techniques provide read and write synchronization, allowing HTM to scale better than typical TM systems. Our primary motivation was to develop a generic HTM system usable in all existent HTM machines, improving their performance and scalability in workloads where readers cannot run in HTM. Finally we present performance comparisons with typical TM systems in several benchmarks.

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 also in the foreseeable future. In this technological landscape, a crucial problem is how to build efficient synchronization primitives that can enable programmers to take full advantage of the performance potential of modern parallel micro-processors.

The introduction of Hardware Transactional Memory (HTM) by mainstream CPU manufacturers like Intel and IBM [22] responds precisely to this urge. HTM provides a highly-efficient, hardware-assisted implementation of the abstraction of atomic transaction, long used in the context of database systems, and now exposed to programmers/compilers 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.

Interestingly, HTM can be employed both directly, 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 techniques 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 [10, 24, 13] in various application domains. Unfortunately, though, several studies have also highlighted that existing HTM implementations suffer of severe limitations, stemming from the inherently restricted nature of the hardware mechanisms that they employ.

Such a design approach limits the applicability of HTM in a number of ways: not only it explicitly restricts the maximum amount of memory positions that can be accessed by transaction, 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 by introducing SpRWL (Speculative RW Lock), a novel HTM-based synchronization primitive that provides a key benefit: allowing read-only atomic blocks to execute outside the scope of any hardware transaction, thus, effectively sparing them from the inherent limitations affecting existing HTM implementations. SpRWL's name stems from the fact that it exposes to programmers the familiar interface of a classic read-write lock and can, therefore, be seen as a specialized HTM-based technique for eliding this type of locks in legacy applications. However, SpRWL can also be straightforwardly employed in applications that assume a transactional API by mapping the beginning of a read-only or an update transaction to a

request for acquiring a read or write lock, respectively.

As we will show, SpRWL can yield remarkable (up to 6×) throughput gains over plain HTM in workloads that have long read-only atomic blocks. However, these throughput gains are achieved at the cost of an increased latency of update atomic blocks, which can suffer from frequent aborts (and theoretically from starvation) in read-dominated workloads.

We evaluated SpRWL via an extensive experimental study conducted using the HTM implementations available on Intel's Broadwell [3] and IBM's Power8 [18] CPUs and encompassing some standard benchmarks (TPC-C [11], STMBench7 [14]). The results of our study shows SpRWLs throughput can reach 15× typical TM systems in some of the standard benchmarks, at the cost of reader latency.

2 Related Work

Hardware Read-Write Lock Elision (HRWLE) introduced the concept of executing read-only critical sections outside the scope of transactions [12]. To do so, HRWLE relied on two specific features available in the POWER8 HTM implementation: suspend-and-resume and Rollback-Only Transactions (ROTs). Although HRWLE achieved significant gains, its application is limited by such features, which are only supported only in one out the four current commodity HTM implementations. SpRWL, however, is portable across all the different HTM implementations and does not require any specific hardware feature other than HTM support. Even its reliance on invariant time-stamp counters, which is not supported by all processors, is for performance optimization and not correctness, unlike HRWLE. Furthermore, as we will show later in Section 4, SpRWL is capable of outperforming HRWLE on POWER8.

Other than HRWLE, SpRWL is very related to two bodies of work: one which tries to enhance the efficiency of HTM systems and the other concerned with developing efficient synchronization mechanisms for read-dominated workloads.

For the former:

- POWER8-TM [17] extended HRWLE with new techniques to further expand the capacity limitation of update transactions but also relies on the ROTs, which again makes it non-portable to other HTM implementations.

- Tuner [9] used online self-tuning mechanisms to determine when to activate a fallback path upon conflict and upon capacity aborts. This is orthogonal to the problem tackled in this paper, and can still be integrated within SpRWL.

- Calciu et al. [5] suggested lazy subscription of the fallback lock from within the HTM transaction to allow more parallelism by reducing the window where a transaction can be aborted after the activation of the fallback path. Unfortunately, though, Dice et. al [8] found out that it is unsafe for lock elision applications to use lazy subscription with current HTM implementations.

For the latter:

- Multiple-Readers/Single-Writer or Read-Write Lock (RWLock) was first suggested by Hamilton [15] as mechanism to allow more concurrency between critical sections that do not alter the shared state.

- Probably the most popular implementation, nowadays, is the Pthreads RWLock [1] which uses two counters, protected by a mutex, to track the number of active readers and writers providing fairness amongst them.

- Big-Reader Locks (BRLock) [7] which was once part of the Linux Kernel [16] uses per thread mutexes and a global mutex. When BRLock is acquired in read mode, a thread needs only to acquire its private per-thread mutex. Whereas for a writer, it must first acquire the global mutex, then each and every per thread mutex.

- Recently, Liu et. al [19] introduced Passive Reader-Writer Lock (PRWL), which tries to reduce the cost imposed by most RWLocks on the writer mode. The idea behind PRWL is a version based consensus protocol between readers and writers. Writers increment the lock version and wait for readers to signal they have read the latest version.

- When compared with SpRWL, the various algorithms that implement RWLock differ in two aspects: (i) they do not allow concurrency among writers, which SpRWL is capable of thanks to HTM, and (ii) they do not allow concurrency between readers and writers, which SpRWL permits via its efficient scheduling and ability to abort writers, without any side-effects, in case they may break the consistency of un-instrumented readers.

- Read-Copy-Update (RCU) [20] is an alternative synchronization mechanism that targets read-dominated workloads. Unlike RWLocks, with RCU, a read-only critical section does not need to acquire any mutex, it just flags itself, using a memory barrier, at the beginning and end of critical section. To ensure correctness, a writer modifying shared data, would create a copy of the data and apply the modifications to the copy. Readers that existed prior to the

write would continue to access the older, unmodified data, while new readers get to witness the updates. Only when all readers that existed before the writer have completed their critical sections, the unmodified data is replaced by the copy. However to support RCU, programs must be written in a way such that creating copies of shared data is feasible. Further, duplicating and discarding the copies must maintain correct pointers to other referenced data. This limits the applicability of RCU and has been the reason for the few number of RCU-based datastructures [4, 6, 21]. On the contrary, SpRWL does not require any changes neither to legacy RWLock-based programs, nor on the approach of developing new software.

3 Algorithm

As already mentioned, SpRWL exposes a classic read-write lock interface. As such, SpRWL can be used as a drop-in, speculative replacement for conventional read-write locks in applications that already use this synchronization primitive; however, it is straightforward to adapt SpRWL's algorithm to be employed also by TM-based applications, by mapping the begin and commit of read-only and update transactions to lock and unlock requests to a single global lock 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 un-instrumented readers and HTM-backed writers. We then extend this base algorithm in Section 3.2, by introducing two scheduling techniques that aim both at enhancing performance and ensuring fairness.

3.1 Base Algorithm

The pseudo-code of SpRWL base algorithm SpRWL is reported in Algorithm 1. In the following, for brevity, we will refer to the threads that request to acquire the lock in read/write mode as readers/writers, respectively.

As already mentioned, write critical sections are executed 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 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 *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 state 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 — this time without recurring to memory barriers, though.

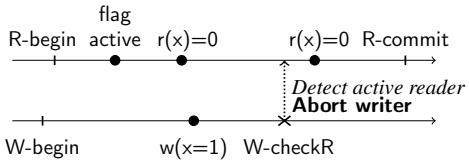

Figure 1: A read access during an active update transaction will abort the latter.

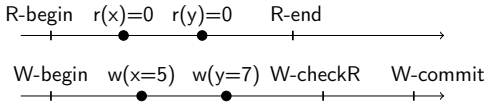

Figure 2: A read access which commits before an active update transaction writes on shared values or verifies the state allows it to successfully commit.

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 in case 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. As in typical HTM systems, SpRWL uses a *single_global_lock* (SGL) as fall-back synchronization method: 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 un-instrumented readers and writers active using the SGL, readers check the SGL after flagging their own state to active, and are allowed proceed only if the SGL is found free (see line 7). 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 36). Overall, this mechanism ensures safety by precluding any concurrency between un-instrumented 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

In order to address the above discussed shortcomings, SpRWL integrates two additional 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 synergistic fashion, 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. The writer synchronization scheme, conversely, stalls writers to prevent them from uselessly consuming their

Algorithm 1 — Basic algorithm (thread *tid*)

```
1:  Global variables:
2:      state[N] ← {⊥, ⊥, ..., ⊥}        ▷ One status per thread
3:      gl                                ▷ global lock for HTM fallback
4:  function SpRWL^basic_READ_LOCK
5:      state[tid] ← #READER              ▷ Flag active reader
6:      MEM_FENCE                         ▷ Make sure writers see reader
7:      READER_GL_SYNC()
8:  function SpRWL^basic_READ_UNLOCK
9:      state[tid] ← ⊥                    ▷ Exit critical section
10: function SpRWL^basic_WRITE_LOCK
11:     attempts ← 0
12:     BEGIN_HTM_TX()                    ▷ Start transaction
13: function SpRWL^basic_WRITE_UNLOCK
14:     if tid is executing in a HTM transaction then
15:         CHECK_FOR_READERS()
16:         TX_COMMIT                     ▷ Commit the HTM transaction
17:     else
18:         RELEASE_GL()
19: function CHECK_FOR_READERS
20:     for i ← 0 to N−1 do               ▷ Abort if any thread...
21:         if state[i] is #READER then   ▷ ...is an active reader...
22:             TX_ABORT()
23: function WAIT_FOR_READERS
24:     for i ← 0 to N−1 do               ▷ Wait for completion...
25:         wait until state[i]= #READER  ▷ of active readers
26: function READER_GL_SYNC
27:     if isLocked(gl) then
28:         state[tid] ← ⊥                ▷ Defer to gl writer
29:         wait until isLocked(gl)       ▷ wait until lock is free
30:         go to 5
31: function BEGIN_HTM_TX
32:     repeat until !locked(gl)          ▷ wait until lock is free
33:     attempts + +
34:     status ← TX_BEGIN()               ▷ Begin HTM transaction
35:     if status == SUCCESS then         ▷ Normal exec. path
36:         if locked(gl) then            ▷ Add lock to read-set and...
37:             TX_ABORT()                ▷ abort Tx if lock is busy
38:     else                              ▷ Branch executed upon abort of a hw tx.
39:         ABORT_HANDLER()
40: function ABORT_HANDLER
41:     if attempts > MAX_RETRIES then    ▷ is budget over?
42:         ACQUIRE_GL()                  ▷ activate fallback
43:         WAIT_FOR_READERS()
44:     else
45:         BEGIN_HTM_TX()
```

budget of attempts using HTM, while striving to achieve maximum concurrency with any active reader.

Also, in this case, we present the two techniques in an incremental fashion, introducing first the reader synchronization scheme and then the writer synchronization mechanism.

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,

Algorithm 2 — Reader synchronization (thread *tid*)

```
1:  Global variables:
2:      ...                                              ▷ As in Alg. 1
3:      clock_w[N] ← {⊥, ⊥, ..., ⊥}                      ▷ per thread
4:      waiting_for[N] ← {⊥, ⊥, ..., ⊥}                  ▷ per thread
5:  function SpRWL^rSync_WRITE_LOCK
6:      state[tid] ← #WRITER
7:      clock_w[tid] ← estimateEndTime()();
8:      SpRWL^basic_WRITE_LOCK()                         ▷ Execute basic alg.
9:  function SpRWL^rSync_WRITE_UNLOCK
10:     SpRWL^basic_WRITE_UNLOCK()
11:     state[tid] ← ⊥
12: function SpRWL^rSync_READ_LOCK
13:     READERS_WAIT()                                   ▷ Sync with writers
14:     SpRWL^basic_READ_LOCK()                          ▷ Execute basic alg.
15: function READERS_WAIT()
16:     wait ← -1
17:     max_wait ← 0
18:     for i ← 0 to N−1 do                              ▷ Wait for longest ...
19:         if state[i]=#WRITER ∧ clock_w[i] > max_wait then
20:             max_wait ← clock_w[i]                    ▷ ...active writer ...
21:             wait ← i
22:         else if waiting_for[i] ≠ ⊥ then              ▷ ...or join..
23:             wait ← waiting_for[i]                    ▷ ...a waiting reader.
24:             BREAK
25:     if wait ≠ −1 then                                ▷ Wait until last writer finishes.
26:         waiting_for[tid]← wait                       ▷ Advertise wait phase.
27:         wait until state[wait]≠ #WRITER
28:     waiting_for[tid] ← ⊥
```

Algorithm 3 — Writer synchronization (thread *tid*)

```
1:  Shared variables:
2:      ...                                              ▷ As in Alg. 2
3:      clock_r[N] ← {⊥, ⊥, ..., ⊥}                      ▷ per thread
4:  function SpRWL^wSync_READ_LOCK
5:      READERS_WAIT()                                   ▷ Sync with active writers
6:      clock[tid] ← 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
14:             WRITER_WAIT()                            ▷ Sync with active readers
15:         BEGIN_HTM_TX()
16: function WRITER_WAIT()
17:     wait ← 0
18:     for i ← 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 ← clock[i]                      ▷ ...in wait.
        ▷ Delay the writer to end δ cycles after the last reader.
22:     wait −= estimateDuration() − δ
23:     repeat until RDTSC() ≥ wait
```

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 order to estimate the expected end time of (write) critical sections in a lightweight, yet accurate, fashion, SpRWL relies on the hardware time stamp counter, which in modern CPUs provides a low-overhead, cycle-accurate time source. Further, in order to cope with programs having critical section of heterogeneous duration, SpRWL gathers independent statistics for different critical sections. More in detail, SpRWL samples the execution time of critical sections on a single thread 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 characteristics. To simplify presentation, we omit describing 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, upon requesting a write critical section, writers advertise their existence and expected end time in the *state* and $clock_w$ arrays, respectively.

Second, before entering a read critical section (via the SpRWL^{basic}_READ_LOCK() function), readers check whether they have to first execute a wait phase (READERS_WAIT() function). More in 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.

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 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 starting 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 the corresponding write critical section completes "shortly after" the last reader. This way, not only writers are guaranteed (or at least is more likely) not to have to abort due to a concurrent reader, they can also maximize the period of time during which their execution overlaps with 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, omit 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, 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 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.

It should be noted 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.

4 Evaluation

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

More in detail, we consider the following baseline solutions: SpRWL is evaluated against: *(i)* pthread's RWLock (rwl), *(ii)* Big Reader Lock [7] (brlock), *(iii)* plain transactional lock elision (tle) and *(iv)* hardware read-write lock elision (herwl) [12], 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 of attempting a transaction 10 times in hardware before activating the fallback path except in case of capacity aborts, where a transactions is directly executed using the fallback path. The only exception is HRWLE, where we used a budget of attempts equal to 10 for update transactions executing using ROTs — the same policy used by the authors of HRWLE in their evaluation. Previous works have shown that dynamically tuning the budget of HTM retries can lead to performance gains in some workloads [23], but in our study we use a common, static retry policy to simplify the analysis of the results.

The evaluation is performed using two different architectures that provide support for HTM, 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 [22].

The source code, which will be made public [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 use to pin a thread per core at the beginning of each run for all the solutions, 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.

4.1 STMBench7

STMBench7 [14] simulates complex applications, such as CAD, composed of several different and large datastructures, such as indexes and graphs. It is one of the most complex TM benchmarks, allowing flexible customization of the generated workload and extensively testing the

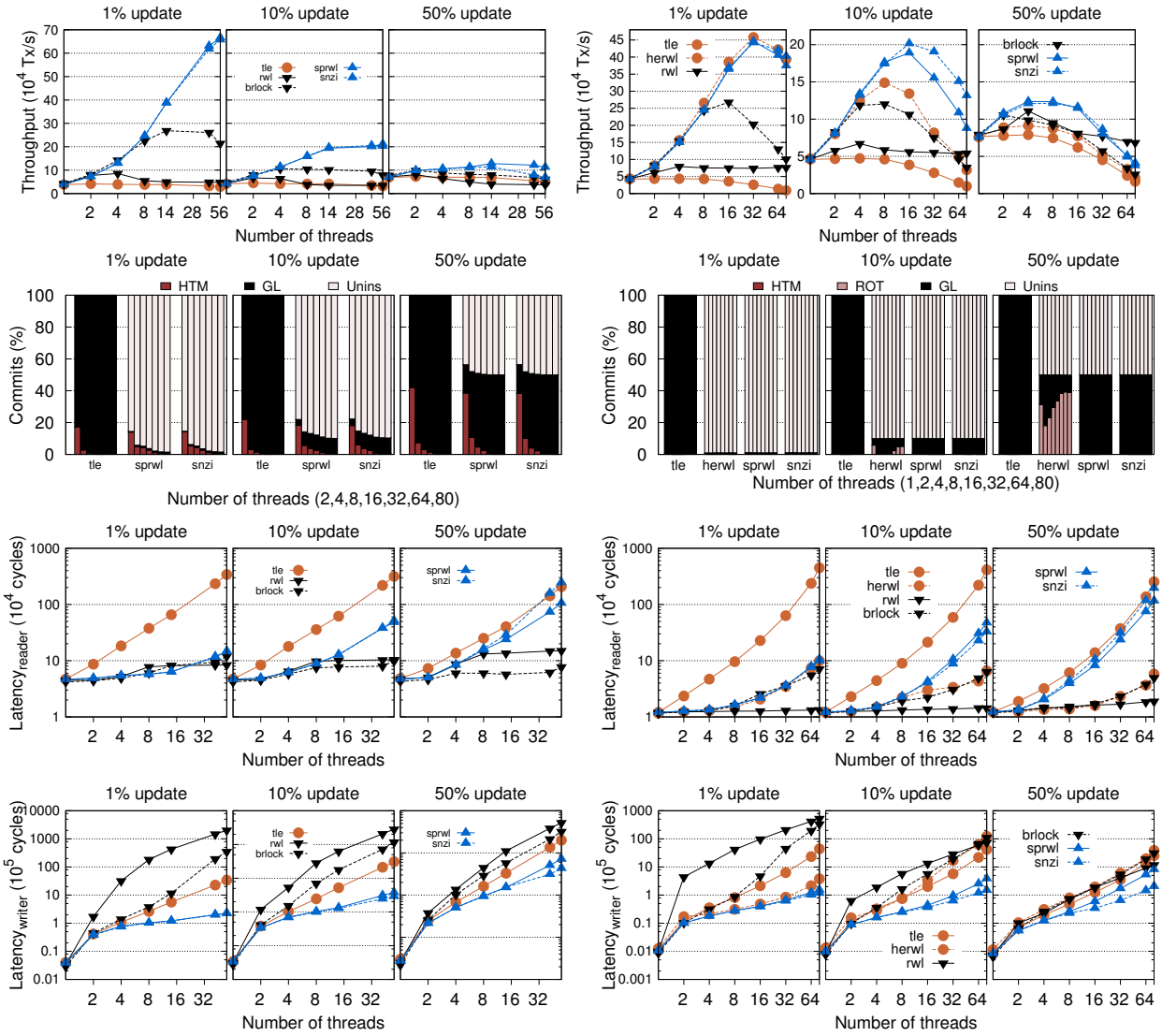

Figure 3: STMBench7: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios on Intel (left) and POWER8 (right).

proposed algorithms in heterogeneous settings. As with the microbench previously tested this workbench only operated transactions which are either large readers which do not fit in HTM or writer which fit in HTM. We also set the benchmark to run in 1%, 10% and 50% in order to showcase SpRWL in its best workloads.

Overall SpRWLs best workload, both in Intel and POWER8, in STMBench7 is u1% where throughput is 13×/2×, respectively, higher than typical backends (3). Latency wise SpRWL benefits mostly writers, as their latency is 2×/1.5×, Intel and POWER8 respectively lower than other backends (3). In terms of scaling, SpRWL scales up to 56 threads on Intel and 32 threads on POWER8, whereas BRLock only manages 14 and 16 threads, respectively.

On Intel SpRWLs achieves throughputs 13×/2×/1.5× at 1%, 10% and 50% updates, respectively. These throughputs are obtained at 56, 42 and 42, respectively, for the update percentages mentioned before. This confirms that as the amount of update transactions increases, concurrency increases, resulting in a decrease of throughput. SpRWL maintains the same reader latency as other backends when thread count increases in 1% updates, but diverges up to 7× and 5× more ate 10% and 50% respectively. Its writer latency however manages to become 100×/20×/10× smaller than other backends at 1%, 10% and 50% updates respectively.

Moving to POWER8, SpRWL manages to outperform HRWLEat 10% and 50% updates with 1.5× their throughput. As for typical locking techniques, SpRWL manages

to achieve throughputs 2× higher at 1% and 10% updates as well as 1.5× higher at 50% update transactions. Latency plots show that as the update % increases so does overall reader latency, going from an identical value at 1% to 10× higher at 50% updates. Writer tendency also shows the same results as in Intel since SpRWL manages to increase its value less than other backends, reaching values 100×/90×/15× lower at 1%, 10% and 50% update transactions, respectively.

4.2 TPC-C

TPC-C benchmark is used to simulate a complete online transaction environment and is representative of a warehouse supplier application. TPC-C uses five different types of transactions (new-order, payment..), with very diverse profiles, such as long read-only transactions, long and contention-prone vs short and almost contention-free update transactions. We use this benchmark in a realistic warehouse simulation.

Overall Both in Intel and POWER8 SpRWLs ideal workload is 10% updates, where it can generate throughputs up to 14×/15× times larger than BRLock, respectively. As seen in the plots (Fig.4) SpRWL shows better or equal latency values as other backends, its reader latency being identical to BRLocks, on Intel, while writer latency manages to reachs over 500× smaller writer latency values at 1% updates compared to BRLock. Results also show that SpRWL can scale to a much higher thread count than other backends, scaling up to 80 threads in u1%, in POWER8, both with SNZI and State Array variants.

On a standard workload (Fig.5) Intel and P8 SpRWL generates throughputs up to 45×/2×, respectively, larger than other backends. It also shows that even in a realistic workload SpRWL scales up to 28 threads on Intel and 16 on POWER8. SpRWL also maintains its lower writer latency on a standard workload, achieving 10× lower values than other backends in both Intel and P8.

On the ideal workloads of Intel SpRWLs achieves throughputs 40×/14×/5.5× at 1%, 10% and 50% updates, respectively. All of these optimal throughputs are at 28 threads, which is where SpRWL scales up to. As mentioned above, SpRWL maintains the same reader latency as other backends as thread count increases across all workloads. Its writer latency however manages to become 150×/100×/2× smaller than other backends at 1%, 10% and 50% updates respectively.

In POWER8 SpRWLreaches throughputs 5×/15×/6× higher than other backends at 1%, 10% and 50%, respectively. Unlike in Intel, SpRWL in POWER8 scales more as the amount of writer transactions decreases, achieving its best throughputs at 80/80/16 threads, respectively. As in Intel, SpRWL also manages to become 100×/100×/5× smaller at 1%, 10% and 50% updates, respectively, in POWER8.

As shown in (Fig.5) SpRWL continues presenting up to 45× throughput on Intel, as update transactions running in HTM allow us to maintain concurrency and our un-instrumented readers can operate while avoiding active writers. As for latency the plots show that SpRWL can have better writer latency than all other compared algorithms, up to 18× less than BRLock at 28 threads. SpRWL reader latency however, is much higher than other backends, reaching up to 10× larger the BRLock. This is due to our waiting policy for active writers, which increases overall throughput and ensures fairness at the cost of a reader start delay.

In POWER8 SpRWL continues outperforming other backends, showing throughput up to 2× higher than other backends. This lower throughput difference in POWER8, compared to Intel, is due to the limited capacity of POWER8 as multi-threading increases. This can be viewed in the abort rates where only our SNZI variant can commit in HTM at high thread counts. Nonetheless SpRWL still shows the effectiveness of un-instrumented readers and HTM. Like Intel, SpRWL in POWER8 also shows the same tendencies in latency, having 40× higher reader latency but compensating with up to 10× lower latency on writers.

4.3 Conclusion

We presented SpRWL, a scalable HTM locking technique which leverages HTMs high isolationism to run readers un-instrumented, and its SNZI variant. We explained how SpRWL uses its synchronization techniques to improve the throughput and scalability of transactional memory systems. Performance wise we show that SpRWL excels in workloads where update transactions fit in HTM and read transactions do not. We also show SpRWLs performance in realistic benchmark simulations out-scaling typical locking mechanisms.

References

[1] Posix.1-2008, 2013.

[2] https://github.com/tsepol/SpRWL, 2018.

[3] AFEK, Y., LEVY, A., AND MORRISON, A. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP '13, ACM, pp. 295–296.

[4] ARBEL, M., AND ATTIYA, 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.

[5] 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).

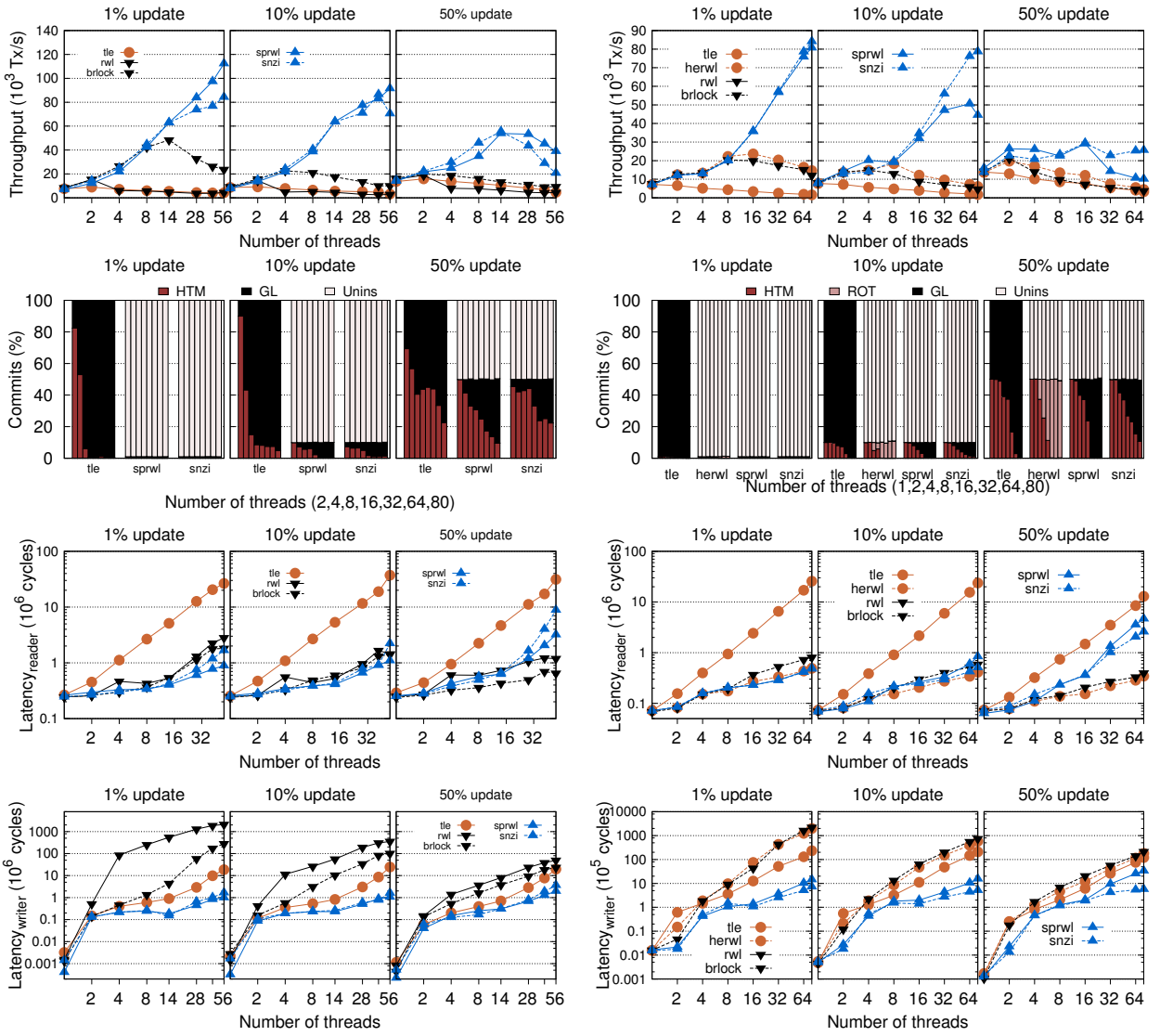

Figure 4: TPC-C: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios with sl and pay transactions on Intel (left) and POWER8 (right).

[6] 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.

[7] CORBET, J. Big reader locks, 2016.

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

[9] 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.

[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] E. JONES. tpccbench. https://github.com/evanj/tpccbench, 2017.

[12] FELBER, P., ISSA, S., MATVEEV, A., AND ROMANO, P. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 34:1–34:15.

[13] 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.

[14] 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 Com-*

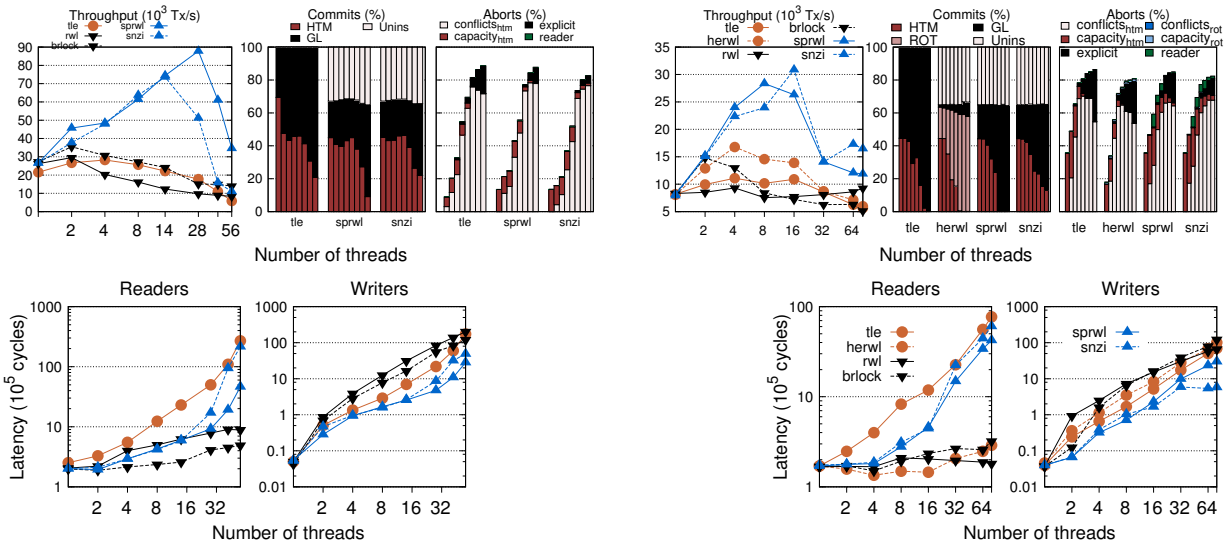

Figure 5: TPC-C: throughput, abort rate, and breakdown of commit modes with the following mix of transactions: sl: 31, del: 4, os 4, pay: 43, no: 18 on Intel (left) and POWER 8(right).

puter Systems 2007 (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 315–324.

[15] HAMILTON, D. Suggestions for multiple-reader/single-writer lock, 1995.

[16] HEMMINGER, S. Kill big reader locks, 2003.

[17] ISSA, S., FELBER, P., MATVEEV, A., AND ROMANO, P. Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume. In *31st International Symposium on Distributed Computing (DISC 2017)* (Dagstuhl, Germany, 2017), A. W. Richa, Ed., vol. 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 28:1–28:16.

[18] 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.

[19] 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.

[20] 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.

[21] MCKENNEY, P. E., AND WALPOLE, J. What is rcu, fundamentally? https://lwn.net/Articles/262464/.

[22] 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.

[23] RUGHETTI, D., ROMANO, P., QUAGLIA, F., AND CICIANI, B. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing* (2014), Springer International Publishing, pp. 475–486.

[24] 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.