Pessimistic Software Lock-Elision

Yehuda Afek[1], Alexander Matveev[2], and Nir Shavit[3]

[1] Tel-Aviv University, afek@post.tau.ac.il
[2] Tel-Aviv University, matveeva@post.tau.ac.il
[3] MIT and Tel-Aviv University, shanir@csail.mit.edu

Abstract. Read-write locks are one of the most prevalent lock forms in concurrent applications because they allow read accesses to locked code to proceed in parallel. However, they do not offer any parallelism between reads and writes.
This paper introduces *pessimistic lock-elision* (PLE), a new approach for non-speculatively replacing read-write locks with pessimistic (i.e. non-aborting) software transactional code that allows read-write concurrency even for contended code and even if the code includes system calls. On systems with hardware transactional support, PLE will allow failed transactions, or ones that contain system calls, to preserve read-write concurrency.
Our PLE algorithm is based on a novel encounter-order design of a fully pessimistic STM system that in a variety of benchmarks spanning from counters to trees, even when up to 40% of calls are mutating the locked structure, provides up to 5 times the performance of a state-of-the-art read-write lock.

Keywords: Multicore, Software Transactional memory, Locks, Lock-elision, Wait-free

1 Introduction

Many modern applications make extensive use of read-write locks, locks that separate read-only calls from ones that can write. Read-write locks allow read-only calls, prevalent in many applications, to proceed in parallel with one another. However, read-write locks do not offer any parallelism between reads and writes.

In a ground breaking paper, Rajwar and Goodman [18] proposed *speculative lock-elision* (SLE), the automatic replacement of locks by optimistic hardware transactions, with the hope that the transactions will not abort due to contention, and not fail to execute due to system calls within the transaction. The SLE approach, which is set to appear in Intel's Haswell processors in 2013 [23], promises great performance benefits for read-write locks when there are low levels of write contention, because it will allow extensive concurrent reading while writing. It will of course also allow write-write parallelism that does not appear in locks. However, if transactions do fail, SLE defaults to using the original lock which has no write-read parallelism.

A few years ago, Roy, Hand, and Harris [4] proposed an all software implementation of SLE, in which transactions are executed speculatively in software, and when they fail, or if they cannot be executed due to system calls, the system defaults to the original lock. In order to synchronize correctly and get privatization, their system uses Safe(..)

instrumentation for objects and a special signaling mechanism between the threads that is implemented inside the kernel. In short, speculative lock-elision is complex and requires OS patches or hardware support because one has to deal with the possible failure of the speculative calls.

This paper introduces *pessimistic software lock-elision* (PLE), a new technique for *non-speculative* replacement of read-write locks by software transactional code. At the core of our PLE algorithm is a novel design of a fully pessimistic STM system, one in which each and every transaction, whether reading or writing, is executed once and never aborts. The fact that transactions are pessimistic means that one can simply replace the locks by transactions without the need, as in SLE [18, 4], to ever revert to the original lock based code. In particular, PLE allows read-write concurrency even for contended code and even if the code includes system calls. It provides only limited write-write concurrency, but then again, read-write locks offer none.

All past STM algorithms (see [21]), including the TinySTM algorithm of Felber, Fetzer, and Reigel [17] and the TL2 STM of Dice, Shalev, and Shavit [9], are optimistic or partially optimistic: some transactions can run into inconsistencies and be forced to abort and retry. Welc et al. [5] introduced the notion of irrevocable transactions. Their system was the first to answer the need to execute systems calls within transactions, but did not relieve the programmer from having to plan and be aware of which operations to run within the specialized pessimistic transaction. Perelman et al. [7] showed a partially pessimistic STM that can support read-only transactions by keeping multiple versions of the transactions' view during its execution. Attiya and Hillel [10] presented a partially pessimistic STM that provides read-only transactions without multiple versions. However, their solution requires acquiring a read-lock for every location being read.

Our new fully pessimistic STM design is an encounter-time variation of our earlier commit-time pessimistic STM [3]. Our algorithm executes write transactions sequentially in a manner similar to [5], yet allows concurrent wait-free read-only transactions without using read-locks or multiple versions as in [10, 7]. We do so by using a TL2/LSA [9, 22] style time-stamping scheme (we can reduce the time-stamp to two bits) together with a new variation on the quiescence array mechanism of Matveev and Shavit [2]. The almost sequential execution of the pessimistic write transactions is a drawback relative to standard TL2, but also has some interesting performance advantages. The most important one is that our STM transactions do not acquire or release locks using relatively expensive CAS operations. Moreover, one does not need read-location logging and revalidation or any bookkeeping for rollback in the case of aborts. Our use of the Matveev and Shavit quiescence mechanism is a variation on the mechanism, which was originally used to provide privatization of transactions, in order to allow write transactions to track concurrent read-only transactions with little overhead. A side benefit of this mechanism is that our new fully pessimistic STM also provides implicit privatization with very little overhead (achieving implicit privatization efficiently in an STM is not an easy task and has been the subject of a flurry of recent research [2, 6, 13–15, 19, 20]).

Though our pessimistic and privatizing STM does not provide the same performance as the optimistic non-privatizing state-of-the-art TL2 algorithm, its performance is comparable in many cases we tested. In particular, this is true when there is use-

ful non-transactional work between transactional calls. Our new pessimistic algorithm is encounter-time, which means locations are updated as they are encountered. Our benchmarks show this improves on our prior commit-time updating approach [3] both in performance and in its suitability to handling system calls within lock code. Most importantly, our new pessimistic STM offers a significant improvement over what, to the best of our knowledge, is the state-of-the-art read-write lock: across the concurrency scale and on NUMA machines, it delivers up to 5 times the lock's throughput. The parallelism PLE offers therefore more than compensates for the overheads introduced by its instrumentation.

Finally, we show how PLE fits naturally with future hardware lock-elision and transactional memory support. We explain how to seamlessly integrate PLE into Intel's hardware lock-elision (HLE) or its restricted transactional memory (RTM) [23] mechanisms, scheduled to appear in processors in 2013. In these mechanisms, transactions cannot execute if they include system calls, and they can fail if there is read-write contention on memory locations. The idea is to execute lock-code transactionally in hardware, and use PLE as the default mechanism to be executed in software if the hardware fails: in other words, elide to the better performing PLE instead of the original read-write lock. Moreover, as we explain, PLE itself can run concurrently with hardware transactions, allowing the user the benefit from both worlds: execute locks with fast hardware transactions in the fast path, or with fast software transactions in the slow path.

2 A Pessimistic Lock-Elision System

We begin by describing the new pessimistic STM algorithm at the core of our system. We will then explain how it can be used to provide non-speculative lock-elision in today's systems that do not have HTM support, and how in the future, one will be able to provide it in systems with HTM support.

2.1 Designing a Pessimistic STM

A typical transaction must read and write multiple locations. Its *read-set* and *write-set* are the sets of locations respectively read and written during its execution. If a transaction involves only reads, we call it a *read transaction*, and otherwise it is a *write transaction*. The transactional writes may be delayed until the commit phase, making the STM *commit-time* style, or may be performed directly to the memory, making the STM *encounter-time*. This paper presents a new encounter-time fully-pessimistic STM implementation that is based on our previous commit-time fully-pessimistic STM [3], in which we allow wait-free read transactions concurrently with a write transaction. Read transactions run wait-free and are never aborted. Write transactions use a lightweight *signaling mechanism* (instead of a mutex lock) to run one after the other, where a new write transaction starts when the previous one begins its commit phase; this allows the execution of one write transaction to be concurrent with the commit phase of the previous write transaction, which we show improves performance. To ensure that a read transaction sees a snapshot view of memory, each write transaction logs the previous

value of the address, and at the beginning of the commit phase a write-transaction waits until all the read transactions that have started before or during its execution phase (that does not include the commit phase) have finished. To implement the synchronization between the write and read transactions we use a variant of the quiescence array mechanism of Matveev and Shavit [2] (which in turn is based on epoch mechanisms such as RCU [8]). Read transactions are made wait-free: locations being updated by a concurrent write transaction (there is only one such transaction at a time) are read from a logged value, and otherwise are read directly from memory. In addition, as a side effect, the quiescence operation provides us with an implicit privatization, which is critical for preserving the read-write lock semantics of the program when replacing the locks with transactions.

Section 2.2 presents the global variables and structures, and defines the API functions of read and write transactions. To simplify the presentation, we first consider the case of only one write transaction executing at a time with possible concurrent read transactions. Section 2.3 presents this implementation, and presents the write transaction commit that allows concurrent read transactions to complete without aborts in a wait-free manner. Next, in Section 2.4, we consider the multiple writers case, where we present a *signaling mechanism* between the write transactions that we found to be more efficient than using a simple mutex, allowing concurrency between the current write transaction's commit and the next write transaction's execution.

2.2 Global Structures

Our solution uses a version-number-based consistency mechanism in the style of the TL2 algorithm of Dice, Shalev, and Shavit [9]. The range of shared memory is divided into stripes, each with an associated local version-number (similar to [9, 17, 1]), initialized to 0.

We use a shared global version number (as introduced by [9, 22]). The global version and stripe versions are 64bit unsigned integers. Every transaction reads the global version upon start, and determines each location's validity by checking its associated stripe's version number relative to the global version.

Our *quiescence mechanism* uses a global *activity array*, that has an entry for every thread in the system. The association between the threads and the activity array entries is explained in detail later. For now assume that *N* threads have ids *[0..N-1]*, and thread *K* is associated with the entry *activity_array[K]*. We later show how to reduce the array size to be a function of the number of hardware cores. The entry for a specific thread holds the *context* of the current transaction being executed by this thread. It includes the transaction's local variables, and shared variables; the ones accessed by other threads.

Figure 1 depicts the algorithm's global variables and structures. They include the *stripe_version_array* that has a version number per stripe, the global version number, and the activity array that holds a *context* for every thread in the system. In addition, Figure 1 shows the API of read and write transactions. Every API function gets *ctx* as a first parameter the thread's context, and the thread's associated *activity_array* entry references the same context.

Every transaction's *context* has a *tx_version* variable that is used to hold the result of sampling the global version number. The *tx_version*'s initial value is the maximum

64bit value. When a transaction starts, the *tx_version* is initialized to the current global version value, and when it finishs, it is set back to the maximum 64bit value.

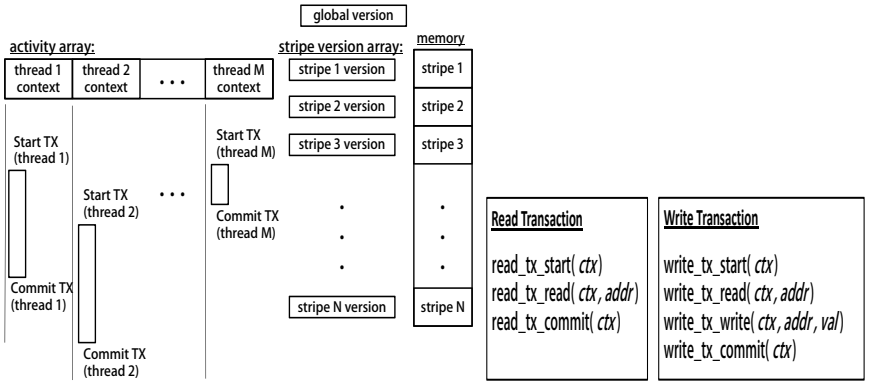

Fig. 1. The algorithm's variables and structures, and the API of a read and write transactions.

2.3 The Core Algorithm

During the write transaction's execution, the write operations are written directly in memory and the overwritten values are logged to the *log_buffer*. Algorithm 1 shows the write transaction's start, read, write and commit functions. A *writer_lock* is used to allow one write transaction at a time. It is acquired on start and released on commit. The write operation logs the write location's old value to the transaction's *log_buffer*, updates the location's stripe version to the next global version and writes the new value to the memory. The order of these operations is important, because the update of the location's stripe version may require the concurrent reads of this location to snoop into the *log_buffer* of this write transaction in order to obtain the most up-to-date value.

When a write transaction commits, the global version is incremented (line 24). This splits the transactions in the system into old transactions and new transactions: ones started before the global version increment and ones that started after it.

It is consistent for new transactions to read the latest values written by the writer, because they started after the global version increment step, and can be serialized as being executed after the writer. This is not the case for old transactions that may have read old values of the locations overwritten by the writer. These transactions are not allowed to read the new values and must continue to read the old values of the overwritten locations in order to preserve their consistent memory view. As a result, old transactions perform a snoop into the concurrent writer's *log_buffer* when reading an overwritten location.

Note that the *log_buffer* values must be preserved as long as there are old read transactions that may read them. Therefore, the writer executes a *quiescence* pass (line 26)

Algorithm 1 Write transaction.

```
1: function WRITE_TX_START(ctx)
2:     mutex_acquire(writer_lock)
3:     g_writer_id = ctx.id
4:     ctx.tx_version ← global_version
5:     memory_fence()
6: end function
7:
8: function WRITE_TX_WRITE(ctx, addr, val)
       ▷ log the old value
9:     n ← ctx.log_size
10:    ctx.log_buffer[n].addr ← addr
11:    ctx.log_buffer[n].val ← load(addr)
12:    ctx.log_size ← ctx.log_size + 1
       ▷ update the stripe version and write the new value
13:    s_index ← get_stripe_index(addr)
14:    s_ver = stripe_version_array
15:    s_ver[s_index] ← ctx.tx_version + 1
16:    store(addr, val)
17: end function
18:
19: function WRITE_TX_READ(ctx, addr)
20:    value ← load(addr)
21:    return value
22: end function
23:
24: function WRITE_TX_COMMIT(ctx)
       ▷ allow new transactions to read the new values
25:    global_version ← global_version + 1
26:    memory_fence()
       ▷ wait for the old transactions to finish
27:    Quiescence(ctx)
       ▷ allow the next writer to proceed
28:    mutex_release(write_lock)
29: end function
```

that waits for the old transactions to finish. These transactions have a *tx_version* less than the new global version (created by the global version increment) because they started before this global version increment. Therefore, it would seem sufficient to scan the *activity_array* for entries having a *tx_version* less than the new global version, and spin-loop on each until this condition becomes false. But, in this way, the scan can miss a transaction, because the *tx_version* modification is implemented as a simple load of the global version and store of the loaded value to the *tx_version*. As a result, a read transaction on start, might load a global version, the concurrent commit may perform the global version increment, and then begin the *activity_array* scan, bypassing the read transaction, because it has not yet performed the store to its *tx_version*. To overcome this scenario, we introduce a special flag, called the *update_flag*. The *tx_version*, is set to this flag value before the reading of global version to the *tx_version*, indicating that a read transaction is in the middle of *tx_version* update. In this case, the writer will wait for the update to finish by spin-looping on *tx_version* until its value becomes different than the *update_flag*'s value. In Algorithm 2 we show the implementation of the quiescence mechanism using this flag, including the read transaction start and commit procedures.

Algorithm 3 shows the implementation of the read transaction's read operation. Upon a read of a location, the transaction first validates that the location has not been overwritten by a concurrent writer by testing the location's stripe version to be less than the read transaction's *tx_version* (lines 2- 9). If validation succeeds, then the location's value is returned. Otherwise the location may have been overwritten and a snoop is performed to the concurrent writer's *log_buffer* (lines 10-28). The snoop simply scans the *log_buffer* for the read location, and returns the location's value from there (note that the scan must start from a 0 index and up, because the location may be overwritten

Algorithm 2 Quiescence.

```
1: function TX_VERSION_UPDATE(ctx)
2:     ctx.tx_version ← update_flag
3:     memory_fence()
4:     ctx.tx_version ← global_version
5:     memory_fence()
6: end function
7: function READ_TX_START(ctx)
8:     tx_version_update(ctx)
9: end function
10:
11: function READ_TX_COMMIT(ctx)
12:     ctx.tx_version ← max_64bit_value
13: end function
14:
15:
16: function QUIESCENCE(ctx)
17:     for id = 0 → max_threads − 1 do
18:         if id = ctx.thread_id then
19:             continue ▷ to next iteration - skip
    this id
20:         end if
21:         cur_ctx ← activity_array[id]
22:         while cur_ctx.tx_version =
    update_flag do
23:         end while                    ▷ spin-loop
24:         while cur_ctx.tx_version <
    global_version do
25:         end while                    ▷ spin-loop
26:     end for
27: end function
```

twice). If this location address is not found in the log, then it means it was not overwritten (the stripe version protects a memory range), and the location's value that was read before the snoop is returned. The relevant *log_buffer* is accessed through the writer's context that is identified by a global index *g_writer_id*. This index is initialized upon write transaction start.

To illustrate the synchronization between the write and read transactions, Figure 2.3 shows 3 stages of a concurrent execution. In stage 1, there is read of transaction 1 and write of transaction 1; both of them read the global version on start, and proceed to reading locations. The read transaction validates that the locations were not overwritten and the writer reads them directly.

In stage 2, the writer performs two writes; (addr1, val1) and (addr2, val2). For every write; (1) the old value is stored in the log buffer, (2) the stripe version is updated, and (3) the new value is written. Then, read transaction 1 tries to read (addr1) from stripe 1 and identifies that the stripe was updated. As a result, it snoops into the concurrent writer's log buffer, searching for (addr1) old value and reading it from there. The second read of (addr3) from stripe 1, also triggers the snoop procedure, but it does not find addr3 in the log buffer and the value of (addr3) is read from the memory.

In stage 3, the writer arrives at the commit point, increments the global version and begins the quiescence step; waiting for old transactions (ones started before the increment) to finish. Specifically, the quiescence waits for read transaction 1. Meanwhile, a new read transaction 2 is started, which reads the new global version. This new transaction can read the new values freely, since it is serialized after the writer. In contrast, read transaction 2 continues to snoop into the concurrent writer log buffer until it is finished, and only then the quiescence step of the writer will finish and the log buffer will be reset. The old values are no longer required because there are no remaining active old readers.

Algorithm 3 Read Operation.

```
1: function READ_TX_READ(ctx, addr)
       ▷ Try to read the memory location
2:   s_index ← get_stripe_index(addr)
3:   s_ver ← stripe_version_array
4:   ver_before ← s_ver[s_index]
5:   value ← load(addr)
6:   ver_after ← s_ver[s_index]
7:   if ver_before <= ctx.tx_version and
     ver_before = ver_after then
8:     return value
9:   end if
       ▷ The read location may had been
     overwritten. Snoop into the concurrent
     writer's log_buffer
10:  wr_ctx ← activity_array[g_writer_id]
11:  log_size ← wr_ctx.log_size
12:  is_found ← False
13:  i ← 0
14:  while is_found = False and i <
     log_size do
15:    p_buf ← wr_ctx.log_buffer
16:    cur_addr ← p_buf[i].addr
17:    cur_val ← p_buf[i].val
18:    if cur_addr = addr then
19:      is_found ← True
20:      snoop_value ← cur_val
21:    end if
22:    i ← i + 1
23:  end while
24:  if is_found = False then
25:    return value
26:  else
27:    return snoop_value
28:  end if
29: end function
```

2.4 The Signaling Mechanism for Write Transactions

In [5], the write transaction coordination is implemented using a global *writer_lock*. Every write transaction tries to acquire this global lock on start and release it upon finish. We have found that these lock acquire and release sequences can cause high cache coherence traffic. To avoid this, we implement a different scheme using a combination of a *writer_lock* and a simple "pass the baton" style *signaling mechanism* in the activity array.

We add to each *context* in the *activity_array* a *writer_waiting* flag. If a write transaction must wait for a concurrent writer, it sets this flag to True and spins on it until it becomes False. The concurrent writer commit scans the *activity_array* for entries having the *writer_waiting* set to True, and signals one of these entries, by changing this entry's *writer_waiting* to False. The signals must be sent in a way that avoids starvation of threads. To make the system fair we scan the activity array for an entry with a waiting writer starting from *thread_id + 1* to the array end, and from 0 to *thread_id-1*. In this way every waiting writer will be signaled after at most *max_threads* write transactions, which is proportional to the *activity_array* length.

In the common case, the write transactions will signal each other using the *writer_waiting* flags, and not by using the global lock acquire and release. That's because usually there is some degree of concurrency between the write transactions. As a result, usually during the commit of a write transaction there will be some entry in the activity array with *writer_waiting* set to True. By setting it to False, only one cache line in a specific core is invalidated, avoiding the global lock release and acquire sequences that invalidate the cache line in all of the cores.

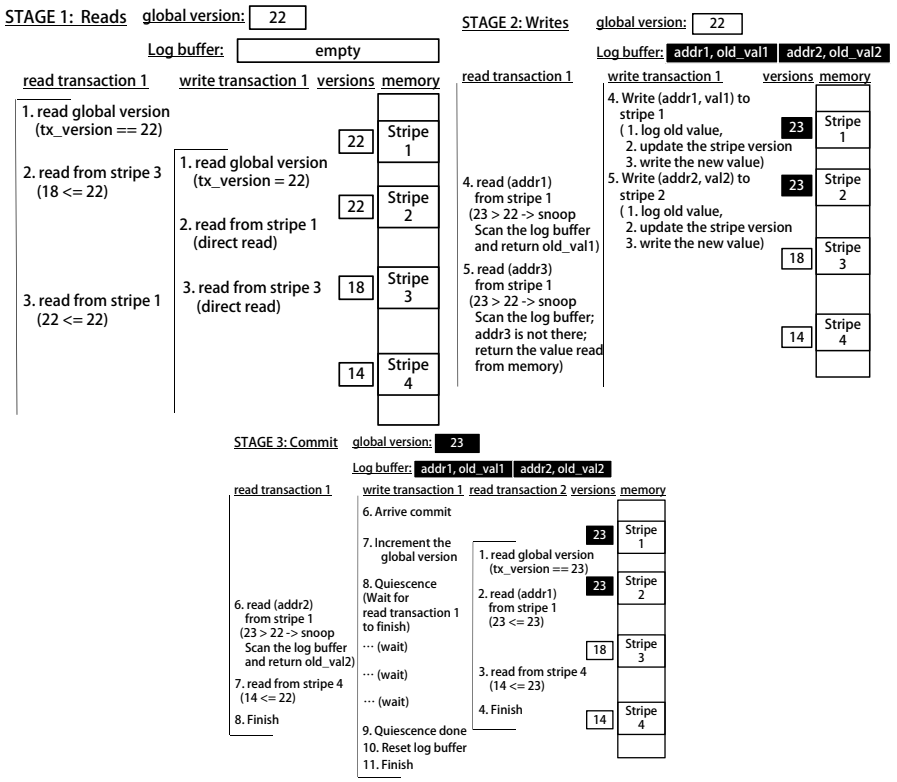

Fig. 2. Three different stages of concurrent execution between read and write transactions are shown.

Now the question is when to execute the signal procedure during the writer commit. The simplest way is to signal the next writer after the commit is done. In general we want to signal the next writer as soon as possible because of the writer's serial bottleneck. The earliest point for the next writer signaling is after the global version increment; immediately before the quiescence step. In this case, the snoop procedure of the read operation is complicated, because now we have the *log_buffer* of the committing writer and the *log_buffer* of the next-started writer. The snoop procedure may need to scan both of the log buffers for the read location. Therefore, we limit the number of log buffers to only two, by not allowing the next writer to signal the following writer until the current writer has finished its quiescence phase.

In summary, we have shown a pessimistic STM algorithm that allows concurrent wait-free reading while writing. We note that there are various elements algorithm that for lack of space we have not described. These include how our signaling mechanism provides better locality of reference in the critical section execution and reduced NUMA traffic by preferring to signal a transaction of a thread on the same chip to run next (up to some threshold so as to maintain fairness). They also include a mechanism to reduce

the version numbers used to only two bits, allowing us to compress more of them into a single cache line in the quiescence array.

3 How to Elide Locks

We present three ways in which PLE can be used to implement lock-elision: non-speculative software-only lock elision, as a fall back (slow path) for the HLE (e.g., Intel's *hardware lock-elision* [12]), and as a fall back using optimistic hardware TM (e.g., Intel's *restricted transactional memory* RTM [12]).

3.1 Non-speculative Software Lock-Elision

To perform non-speculative elision, for every RW-Lock code section, the RW-Lock acquire and release calls are replaced with the PLE transaction start and commit calls (the read acquisition with a read transaction start and write acquisition with a write transaction start). The loads and stores are instrumented according to our above pessimistic STM algorithm with transactional read and write calls. We will denote each code section transformed into a PLE based code section as *the PLE code path* of this segment. This transformation introduces a read-write concurrency to the program that may result in two special cases:

1. *Conflicting I/O:* The concurrently executing read and write critical sections may invoke conflicting I/O requests, like a read and a write to the same file. In this case, a simple solution is to mark the conflicting I/O read critical section as a write critical section; resulting in the conflicting I/O serialization.
2. *Private Operation:* Inside the write critical section there may be a call for an operation that requires privatization (mutual exclusion) on the data it accesses. For example, a call for a free function on a shared memory. PLE provides privatization only after the commit operation and therefore these kind of operations must be moved to after the commit of the write critical section.

3.2 PLE as a Fall Back for HLE

In Intel's HLE, lock-protected code sections typically execute without locking and without interruption if they contain no system calls and if no conflicts are detected by the cache coherence protocol (there may be various other spurious reasons). If the h/w based speculation fails, it falls back to the software based locks that offer no read-write concurrency.

While Intel's HLE does not provide user specified software abort handlers, it does provide an XTEST instruction which returns true if the thread is currently executing in HLE (or RTM), and false otherwise – when an HLE or RTM transaction has been aborted. Thus, by executing XTEST after the XACQUIRE instruction (the HLE transaction start instruction), we can tell whether a fall-back to the hardware mechanism should be executed, or HLE should continue. For this to work we need to prepare at compile time a duplicate of each read-write lock protected code section where the duplicate is transformed into the corresponding PLE code path, as in the previous subsection. If the XTEST fails, then the duplicate PLE path is called.

3.3 PLE as a Fall Back for RTM

As before, each read-write lock-protected code segment is duplicated, one copy is transformed into the corresponding PLE path, as in Subsection 3.1, and the other is converted into an RTM code path as follows. Replace the acquire and release with XBEGIN and XEND, the RTM transaction start and end calls, and specify the fall-back routine (a parameter to XBEGIN) to be the matching PLE code path start. In addition, after the XBEGIN, add a read (load) instruction of a shared variable, called *is_abort*. We use *is_abort* to abort all of the hardware transactions currently executing if one of them has transitioned to PLE.

By default, each read-write lock section is first attempted as an RTM code path transaction. If it fails, a jump to the PLE pessimistic transaction start routine is performed. This routine first executes a small RTM transaction that updates the shared variable *is_abort*. This will cause all of the currently executing RTM transactions to fail. The result is a shift of the whole system to PLE. Now the PLE execution proceeds normally.

If in the RTM design, a hardware transaction is aborted when its cache line is invalidated, then we can allow execution of RTM hardware read only transactions concurrently with PLE transactions (this assumes a specific implementation of RTM which at this time we have no specfic information about [12]). This is because the PLE transactions never abort, and the cache coherence ensures that RTM hardware read-only transactions are atomic. In this case, we can avoid shifting the whole system from RTM to PLE, and shift only the write transactions.

Finally, we note that a transition from PLE back to RTM is also possible, but do not describe it here for lack of space.

4 Empirical Performance Evaluation

We empirically evaluated our algorithm on an Intel 40-way machine that includes 2 Intel Xeon E7-4870 chips on a NUMA interconnect. Each chip has 10 2.40GHz cores, each multiplex- ing 2 hardware threads (HyperThreading), and each core has private write-back L1 and L2 caches and the L3 cache is shared.

The algorithms we benchmarked are:

PLE *Pessimistic Lock Elision:* our fully pessimistic encounter-time STM.

PTM *Pessimistic Transactional Memory:* The commit-time variation of our fully pessimistic STM [3].

RW-Lock An ingress-egress counter based reader-writer mutex implementation (in general, it uses a two global counters. one for read acquires and one for read releases. Writers compute the difference of these two counters to determine when there are no more readers in the system). This is state-of-the-art RW Lock implementation for Intel platform.

MCS-Lock Michael and Scott's MCS Lock [16].

We present two standard synthetic microbenchmarks: a *Red-Black Tree* and a single location counter *(*Counter-1).

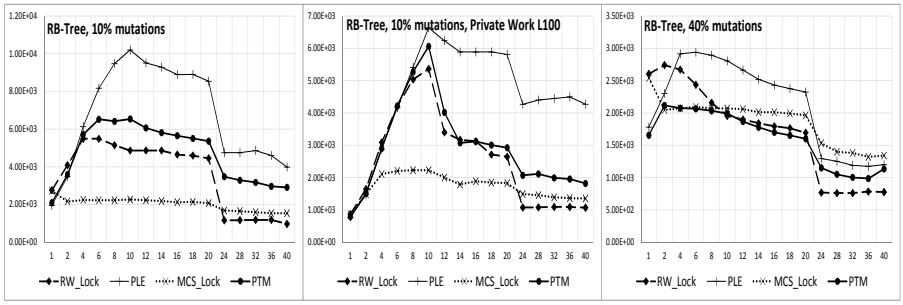

Fig. 3. Throughput of 200K sized Red-Black Tree with varying number of mutations; 10% and 40%, and varying amount of private work after the write transactions; 0 and L100 (100 dummy memory fences). The Y-axis denotes operations per second and X-axis the number of threads. Upto 10 threads every thread runs on its own core. Above 10 threads, the threads are being multiplexed, and we have 2 threads per core. From 20 threads on the second chip is being used; using the NUMA interconnect

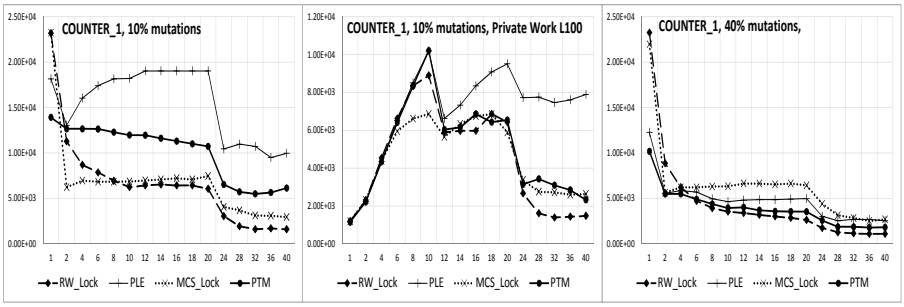

Fig. 4. Throughput of *Counter-1* benchmark with varying number of mutations; 10% and 40%.

The red-black tree implementation exposes a key-value pair interface of *put*, *delete*, and *get* operations. The *put* operation installs a key-value pair, if the key is present, else updates the key's node value. *Delete* removes the key's node, if present, and *get* returns the value associated with a key. We allow the tree to grow to a maximum of 200K elements from an initial 100K elements. We vary the fraction of mutation operations and the number of local private operations of threads after the write methods. For example, 10% mutations means we execute 5% puts and 5% deletes. We tested various rates and also a *Private Work L100* benchmark which executes 100 dummy memory fences after the write transaction.

In all the presented graphs, up to a concurrency of 10, all threads are running on separate cores on a single chip. From 11 to 20 they are being multiplexed on the 10 cores of the same chip, and from 21 to 40 they are multiplexed on the two NUMA cores of the machine.

We began by testing the benefit of allowing write concurrency in PLE and PTM. We noticed that in several benchmarks allowing write concurrency, even though it is minimal and commits are still serialized, provides a 30% performance improvement. Next, we added a priority to the *signaling mechanism* so that it will first try to signal write transactions from the same chip so as to get better locality of reference in consecutive critical section executions and avoid NUMA traffic (See [11]). We defined a constant threshold value that will limit the number of signals in the same chip, in order to avoid starvation.

As a reference point, we also compared our algorithms to the TL2 STM on the RB-Tree, despite the fact that TL2 is optimistic and non-privatizing and cannot be used to provide non-speculative lock-elision. The comparison shows that TL2 is better than PLE above about 10 threads (not included in the graphs), because in TL2 we have concurrency between the write transactions, and in the RB-Tree benchmark the number of aborts is very low. For a high number of aborts, TL2 performance degrades. Also, adding private work after the write transactions makes PLE performance similar to TL2 (upto 20 threads), because the contention is reduced.

We next ran the red-black tree benchmarks in Figure 3. Consider first the results for 10% mutations without private work (left graph) and with private work (L100 case - middle graph). For the case without private work, the MCS-Lock does not scale and the RW-Lock and PLE have similar performance until 4 threads. With more than 4 threads, PLE runs 2 times faster than the RW-Lock until 20 threads is reached. After 20 threads, we cross the boundary of one chip and start to use both of the Intel machine chips. The communication between the chips is NUMA and it is expensive, therefore, we get a performance drop in both the RW-Lock and PLE. Still, in the NUMA range (21-40 threads), PLE runs 4.5 times faster than the RW-Lock. In contrast to PLE, PTM's performance is close to that of the RW-Lock. PTM is a commit-time STM, executing more expensive write transactions. Since writers are a bottleneck, the encounter-time order of PLE makes a difference and runs faster than PTM. When there is private work, we can see that the RW-Lock, PTM, and PLE, all have a similar performance until 10 threads. Beyond 10 threads, the RW-Lock and PTM show a similar drop in the performance, while PLE runs 2 times faster than both of them until 20 threads, and 4 times faster in the NUMA range. Note, that all of the algorithms have a performance drop in the 12 threads range for the private work case. This is because the Intel machine we use starts to multiplex (use HyperThreading) from 11 to 20 threads. Above 20 threads it starts to use the second chip. We executed additional profiling analysis of the L1 cache miss rate for the benchmarks and found that the MCS-Lock has the lowest number of cache misses. Next is the RW-Lock and only then PLE. This means that a large part of PLE's performance gain is due to parallelism despite the overhead of its instrumentation and its lesser locality of reference.

In Figure 3 (right graph) we benchmark the high mutation rate of 40%. In this case, the MCS-Lock performs better than the RW-Lock above 8 threads, and PLE outperforms the RW-Lock and MCS-Lock after 4 threads and until we reach 20 threads; PLE runs 1.4 times faster in this range. In the NUMA range the MCS-Lock outperforms PLE. This is a result of a high mutation rate and lower possible concurrency between

the read and write transactions. The MCS lock causes cache lines to bounce from one core to the other significantly less times.

In the *Counter-1* benchmark we model an extreme contention situation, in which every write transaction increments a shared counter and every read transaction reads this shared counter. Also, we test the case of *Private Work L100*.

Results for *Counter-1* are shown in Figure 4. For 10% mutations (write transactions) PLE is 3 times faster than RW-Lock at up to 20 threads, and 4.5 times faster in the NUMA range. For 40% mutations, the MCS-Lock outperforms both PLE and the RW-Lock because of the extreme contention on the shared counter. Again, the MCS lock has the lowest cache miss rate, though PLE's rates are not as bad as in the red-black tree benchmark. Perhaps the biased preference to signal a thread on the same node reduces bouncing of the counter cache line in PLE. For 10% mutations with private work, PLE and the RW-Mutex are similar until 8-10 threads. At 12 threads we see a performance drop because of the HyperThreading, and then we see that PLE runs 1.4 times faster until 20 threads, and 5 times faster in the NUMA range.

Acknowledgments We thank Dima Perelman and two anonymous PODC referees for inspiring this paper by suggesting that we compare our pessimistic STMs to read write locks. This helped set us along the path of noticing that with pessimistic transactions one could actually perform straightforward non-speculative elision of read-write locks. This work was supported by the Israel Science Foundation under grant number 1386/11 and the US National Science Foundation under grant number 1217921.

References

1. M. Kapalka A. Dragojevic, R. Guerraoui. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM.
2. D. Dice A. Matveev and N. Shavit. Implicit privatization using private transactions. In *Transact 2010*, Paris, France, 2010.
3. N. Shavit A. Matveev. Towards a fully pessimistic stm model. In *TRANSACT 2012 Workshop, New Orleans, LA, USA*, 2012.
4. T. Harris A. Roy, S. Hand. A runtime system for software lock elision. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 261–274, New York, NY, USA, 2009. ACM.
5. A. Adl-Tabatabai A. Welc, B. Saha. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM.
6. H. Attiya and E. Hillel. The cost of privatization. In *DISC*, pages 35–49, 2010.
7. I. Keidar D. Perelman, R. Fan. On maintaining multiple versions in stm. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM.
8. M. Desnoyers, A. Stern P. McKenney, and J. Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 2009.
9. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

10. E. Hillel H. Attiya. Single-version stms can be multi-version permissive. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 83–94, Berlin, Heidelberg, 2011. Springer-Verlag.
11. M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
12. Intel. Intel architecture instruction set extensions programming reference – chapter 8. *Document 319433-012A*, 2012.
13. Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT09*, 2009.
14. H. Machens and V. Turau. Avoiding Publication and Privatization Problems on Software Transactional Memory. In Norbert Luttenberger and Hagen Peters, editors, *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*, volume 17 of *OpenAccess Series in Informatics (OASIcs)*, pages 97–108, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
15. V. Marathe, M. Spear, and M. Scott. Scalable techniques for transparent privatization in software transactional memory. *Parallel Processing, International Conference on*, 0:67–74, 2008.
16. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
17. C. Fetzer P. Felber and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM.
18. R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.
19. T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42:78–88, June 2007.
20. M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. ACM.
21. R. Rajwar T. Harris, J. Larus. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.
22. P. Felber T. Riegel and C. Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.
23. Web. Intel tsx http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell, 2012.