# Software-Improved Hardware Lock Elision

Yehuda Afek
Tel Aviv University

Amir Levy
Tel Aviv University

Adam Morrison*
Technion

## ABSTRACT

With hardware transactional memory (HTM) becoming available in mainstream processors, lock-based critical sections may now initiate a hardware transaction instead of taking the lock, enabling their concurrent execution unless a real data conflict occurs. However, just a few transactional aborts can cause the lock to be acquired non-transactionally resulting in the serialization of all the threads, severely degrading the amount of speedup obtained.

In this paper we provide two software extension mechanisms that considerably improve the concurrency and speedup levels attained by lock based programs using HTM-based lock elision. The first sacrifices opacity to achieve higher levels of concurrency, and the second retains opacity while reaching slightly lower levels of concurrency.

Evaluation on STAMP and on data structure benchmarks on an Intel Haswell processor shows that these techniques improve the speedup by up to 3.5 times and 10 times respectively, compared to using Haswell's hardware lock elision as is.

## Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*concurrent programming*

## Keywords

Lock elision; Lock removal

## 1. INTRODUCTION

Hardware transactional memory (HTM) [16] is now available on mainstream mass-market processors (e.g., Intel's latest Haswell microarchitecture, IBM POWER architecture [2, 8]), and programmers are expected to use the offered hardware-based *lock elision* [21] to improve the performance of their (legacy) coarse-grained lock-based programs [1]. However, naively using hardware lock elision can lead to disappointing performance results. We therefore provide software techniques for *assisting* the hardware's lock elision, to get around the limitations which introduce excessive serialization in an otherwise potentially concurrent application. Our techniques significantly improve the performance of concurrent applications when using lock elision.

In HLE, lock protected code segments are executed speculatively by starting a transaction and "taking" the lock without actually taking it. That is, the lock is read, and if its state is unlocked then it is placed in the transaction read set as locked without affecting the state of the lock, that remains unlocked. However, when a transaction aborts (e.g., due to a conflict) it rolls back to acquire the lock non-speculatively, writing the lock as taken. The globally visible lock acquisition by the aborted thread conflicts with the speculative loads of the lock that has been performed by the speculatively running HLE transactions, and *causes all of them to abort* because of this conflict on the lock location. In addition, *new threads arriving at the critical section see that the lock is taken* and thus they do not start their transaction. In the case of fair locks this conflict on the lock serializes the run until a quiescent period in which no thread tries to access the lock. This *lemming effect* [12] which causes unnecessary serialization limits the concurrency exposed. To mitigate this effect, Intel recommends retrying an aborted transaction several times [1]. However, our experiments (Section 7) reveal that this simple technique does not completely alleviate the problem, especially when the lemming effect is severe such as with fair locks or with a high degree of concurrency.

**Software-assisted lock removal (SLR):** Our first software technique to overcome this lemming effect can be viewed as a software hardware hybrid implementation of the Rajwar Goodman hardware lock removal technique [22]. They observed that one can simply execute transactions with the same scope of the critical section (i.e., start a transaction instead of acquiring the lock and commit instead of releasing the lock) *without accessing the lock at all*, provided the TM offers some progress guarantee for conflicting transactions. (Otherwise, livelock situations, in which transactions repeatedly abort each other, can occur.) However, Haswell's HTM has a simple "requestor wins" conflict resolution policy [1] which is prone to livelock [7]. Our *software-assisted lock removal* SLR scheme guarantees progress despite this by doing the following. It uses the HTM to transactionally execute critical sections without accessing the lock until it is ready to commit. Then it reads the lock and commits if the lock is not taken; otherwise, it aborts and retries. If it fails a few times it gives up and executes non speculatively, by acquiring the lock. In SLR a thread acquiring the lock does not automatically conflict with running transactions nor does it prevent an arriving thread from starting its transaction speculatively. Since in SLR a speculative transaction may run concurrently to a transaction that holds the lock, the speculative transaction may see an inconsistent state (which guarantees that it will fail to commit and

*Work done while a PhD student at Tel Aviv University.


*PODC'14,* July 15–18, 2014, Paris, France

http://dx.doi.org/10.1145/2611462.2611482.

abort). In many cases this loss of opacity [14] is safe because of the transactions sand boxing.

**Software-assisted conflict management (SCM):** To prevent the lemming effect in HLE transactions without resorting to lock removal, and without losing the opacity property, we propose a simple *conflict management* technique that allows the non-conflicting threads to continue their speculative HLE-based run *without any interference* from conflicting threads. To do this we add a *serializing path* to the lock implementation, in which an aborted thread has to acquire a distinct *auxiliary lock* (without using lock elision) in order to rejoin the speculative execution with the other threads. Using this approach conflicting threads are serialized among themselves and do not interfere with other threads. Only if the thread fails due to a conflict many times it must give up and acquire the original lock.

While SCM provides the most benefits when employed with HLE, the two schemes, SCM and SLR can be combined together to further reduce any progress problems caused when SLR threads give up and acquire the lock non-transactionally. Furthermore, to the best of our knowledge, SCM is the only scheme that enables HLE-based fair locks, with starvation freedom and progress guarantees and with no performance degradation.

We implemented our software-assisted methods in a library that uses the standard `pthreads` lock interface. This allows using our methods without requiring changing or recompiling the program.

**This paper's contributions are therefore:**

- Analyzing the performance dynamics of Haswell's HLE and quantifying the impact of the lemming effect on it (Section 4).
- Introducing software assisted lock removal (SLR) that regains the concurrency and speedup that were lost due to the HLE lemming effect (Section 5). SLR achieves higher levels of concurrency while sacrificing opacity.
- Introducing the software assisted conflict management (SCM) an alternative scheme that overcomes the lemming effect that works well with fair locks, such as MCS, Ticket or CLH locks (Section 6). SCM retains opacity while reaching slightly lower levels of concurrency (compared to SLR).
- Evaluating the two schemes showing that they improve performance by up to 3.5 times in the STAMP application benchmarks and up to 10 times in data structures benchmarks as compared to using Haswell's HLE as is (Section 7).

## 2. RELATED WORK

Rajwar and Goodman [21] introduced the concept of speculative lock elision (SLE). They subsequently proposed transactional lock removal [22], which uses hardware-based conflict management to serialize conflicting transactions. Our approaches achieve a similar goal, but using software to assist the hardware implementation.

Dice et al. [12] studied transactional lock elision (TLE) using Sun's Rock processor and mentioned the lemming effect. In response, they sketch a non-backoff software mechanism to speedup *recovery* from the lemming effect. In contrast to their technique, our conflict management scheme prevents the problem in the first place and manages to prevent the continuous zigzag between speculative and standard executions altogether.

Implementing elision-friendly locks using Intel's Haswell processor is discussed in [1]. However Intel's optimization guidelines essentially turn fair locks into TTAS locks. This has two disadvantages: (1) wasting time when arriving while the lock is taken (as our experiments on STAMP show, this is significant), and (2) the lock no longer guarantees starvation-freedom and loses its fairness.

```
1  shared variable:
2    lock : 1 bit (boolean), initially FALSE
3
4  lock() {
5    while (TRUE) {
6      while (lock = TRUE) {
7        // busy wait
8      }
9      ret := XACQUIRE test&set(lock)
10     if (ret = FALSE)
11       return
12 } }
13
14 unlock() {
15   XRELEASE lock := FALSE
16 }
```

Figure 1: Applying hardware lock elision to a TTAS (Test&Test&Set) lock.

Roy et al. [23] and Afek et al. [5] implement lock elision completely in software, using specialized software transactional memory algorithms. These implementations instrument the critical sections to track read and written memory locations. In contrast, our algorithms are based on hardware TM and thus do not require such instrumentation.

We have sketched the software-assisted conflict management in a previous poster publication [3]. Concurrently to the current work, Calciu et al. [9] proposed using a mechanism similar to software-assisted lock removal as a fallback for transactional memory systems.

## 3. BACKGROUND: INTEL'S HASWELL HTM

Intel's transactional synchronization extensions (TSX) [2] defines two interfaces to designate the scope of a transaction:

**Hardware lock elision (HLE):** In HLE, the scope of a lock-protected critical section defines a transaction's scope. HLE is implemented as a backward-compatible instruction set extension of two new prefixes, XACQUIRE and XRELEASE. Upon executing an XACQUIRE-prefixed instruction that writes to memory (e.g., a store or `compare-and-swap`) (see Figure 1), the processor starts a transaction and *elides* the actual store, treating it as a transactional read instead (i.e., placing the lock in the read set). Internally, however, the processor maintains an illusion that the lock was acquired: if the transaction reads the lock, it sees the value stored locally. Upon executing an XRELEASE store, the transaction commits. HLE requires that an XRELEASE store restores the lock to its original state; otherwise, it aborts the transaction. If an HLE transaction aborts, the XACQUIRE store is re-executed non-transactionally to acquire the lock in order to ensure progress. Notice that such a non-transactional store conflicts with every concurrent HLE transaction eliding the same lock, since such a transaction has the lock's cache line in its read set. This is the root cause for the lemming effect.

**Restricted transactional memory (RTM):** RTM is Haswell's generic TM interface with three new instructions: XBEGIN, XEND, and XABORT. XBEGIN begins a transaction, XEND commits, and XABORT allows a transaction to abort itself. Upon an abort a fallback code that is pointed by an operand of the XBEGIN instruction is executed and uses an *abort status* register in which the processor records the cause for the abort, whether due to an XABORT, a data conflict, or an "internal buffer overflow" [2].

RTM can be used to implement custom lock elision algorithms [1] by replacing the lock acquisition code with custom code that begins a transaction and reads from the lock's cache line. However, such a lock elision scheme fails to maintain the illusion that the thread

wrote to the lock, as the lock's cache line is indistinguishable from any other line in the read-set.

## 3.1 Haswell's TSX Implementation

Haswell appears to use a *requestor wins* conflict management policy. A transaction aborts if either a coherency message (read or write) arrives for a cache line in its write set, or if an eviction due to a write arrives for a cache line in its read set.

Experiments we conducted (described in [4]) show that transactions are prone to *spurious aborts* that are not explained by data conflicts or read/write set overflow. Spurious aborts imply that even in a perfect conflict free workload, degradation such as the lemming effect, described below, is possible.

**HLE compatible locks** Haswell's HLE mechanism conservatively requires that the store releasing the lock restores the lock to its original state prior to the acquisition [2]. Unfortunately, the popular (fair) ticket lock [18] (used in the Linux kernel [20]) and CLH lock [17, 11] do not meet this requirement. As an additional contribution, we adapt these locks for use under HLE, thus enabling HLE-based code to maintain the progress guarantees fair locks provide, and making HLE applicable to programs that use ticket locks or CLH locks.

We adjust both locks in a way that guarantees that a thread running alone (which is the illusion given by HLE) restores the lock to its original state when it releases the lock. The idea is that a thread releasing the lock first tries to optimistically restore the original state using a `compare-and-swap` instruction. If this fails the thread reverts to using the standard lock algorithm. But if the CAS succeeds, the lock's state is restored, which is exactly what HLE requires. The algorithms appear in Appendix A and (due to space constraints) their correctness proofs appear in [4].

# 4. LEMMING EFFECT IN HASWELL HLE

In this section we experimentally quantify the serialization penalty due to transactional aborts during an HLE execution. We focus our analysis on the HLE-based test-and-test-and-set (TTAS) lock (Figure 1) and the fair HLE-based MCS [18] lock. We use the MCS lock as the representative of the class of fair locks because it is compatible with HLE, unlike other fair locks such as ticket locks or CLH locks. However, we have verified that both these locks suffer from the same problems reported below for the MCS lock.

We use a red-black tree data structure protected by a single global lock. Varying the number of threads, the operation mix, and the tree size allows us to control the conflict level and the length and amount of data accessed in the critical section. Small tree and/or many mutating `insert/delete` threads result in higher conflict levels. Increasing the size of the tree reduces the chance that two operations' data accesses conflict, as the elements accessed are more sparsely distributed. For a given size, $s$, we initially fill the tree with random elements from a domain of size $2s$. Then, we run for a period of 3 seconds in which each thread continuously performs random `insert`, `delete` and `lookup` operations, according to a specified distribution. (We use an equal rate of `inserts` and `deletes` so that on average the tree size does not change.)

Experiments were performed on a Core i7-4770 3.4 GHz Haswell processor, with 4 cores, each with 2 hyperthreads. Each core has private L1 and L2 caches, whose sizes are 32 KB and 256 KB respectively. There is also an 8 MB L3 cache shared by all cores. Each test point is the average on 10 runs (with little observed variance). We measured: (1) the total number of operations completed, (2) $S$, the number of successful speculative operations, (3) $A$, the number of aborted speculative operations and (4) $N$, the number of

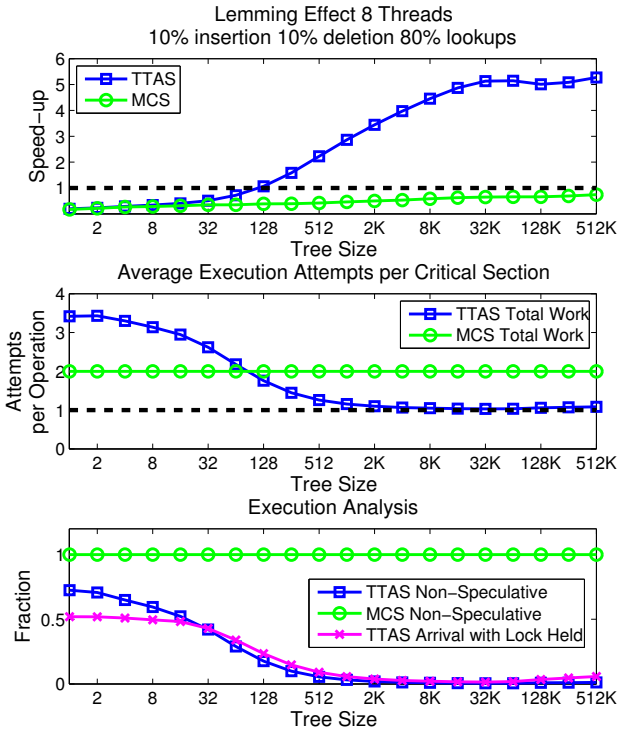



Figure 2: Impact of aborts on executions under different lock implementations. For each tree size we show the average number of times a thread attempts to execute the critical section until successfully completing a tree operation, and the fraction of operations that complete non-speculatively. CLH and ticket results are omitted, as they are similar to the MCS lock results.

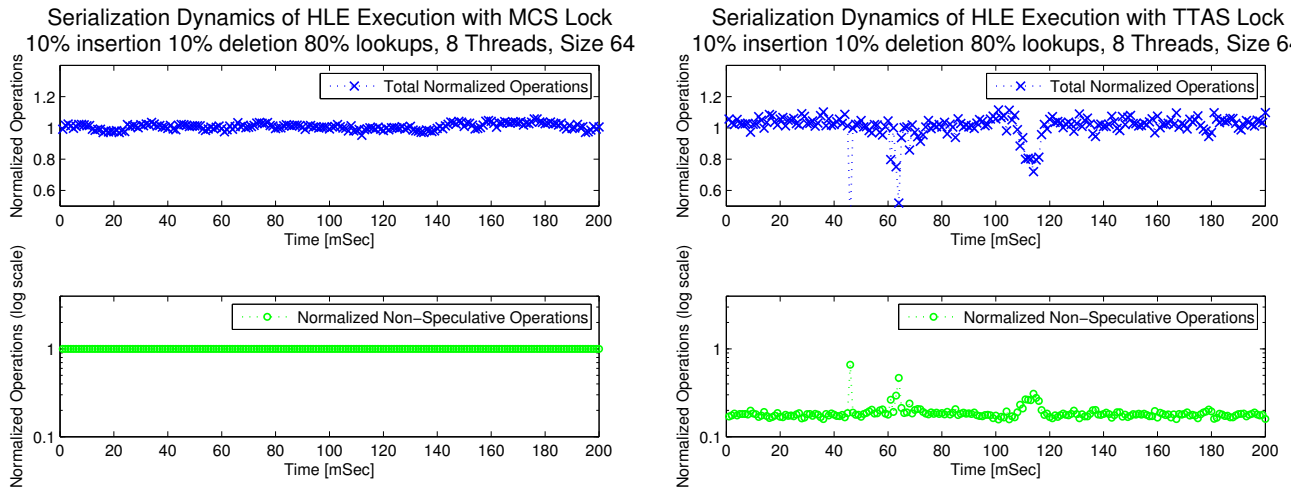
operations that complete via a non-speculative execution. The total number of operations performed is $S+N$. In some lock implementations an operation can start and abort several speculation attempts before completing, so there is no formula relating $A$ to $S$ and $N$.

Figure 2 shows the amount of serialization caused by aborts, as a function of the tree size, for a moderate level of tree modifications (20%). In addition to the fraction of operations that complete non-speculatively (i.e., $\frac{N}{N+S}$), we report the amount of *work* required to complete an operation, i.e., $\frac{A+N+S}{N+S}$, the number of times a thread tries to complete the critical section before succeeding.

As Figure 2 shows, the serialization dynamics for each lock type are quite different. With an MCS lock, the benchmark executes virtually all operations non-speculatively after an initial speculative section aborts. As a result, an HLE MCS lock offers little if any speedup over a standard MCS lock, even when there is little underlying contention.

The TTAS lock, on the other hand, manages to recover from aborts. At high conflict levels (on small trees) it requires $2-3.5$ attempts to complete a single operation, but nevertheless a fraction of 30% to 70% of the operations complete speculatively. As the tree size increases and conflict levels decrease, HLE shines and nearly all operations complete speculatively. We now turn to analyze the causes for these differences.

**TTAS spinlock (the boxed line in Figure 2 )** The first thread to abort acquires the lock non-speculatively. As for the remaining threads, we distinguish between two behaviors. First, a thread that aborts because of this lock acquisition re-executes its acquiring TAS instruction, which returns 1 because the lock is held. The thread then spins, and once it observes the lock free re-issues its




MCS lock: All operations complete non-speculatively.

TTAS lock: Most operations complete speculatively but there are periods of serialization.

Figure 3: Normalized throughput and serialization dynamics over time. We divide the execution into 1 millisecond time slots. **Top:** Throughput obtained in each time slot, normalized to the average throughput over the entire execution. **Bottom:** Fraction of operations that complete non-speculatively in each time slot.

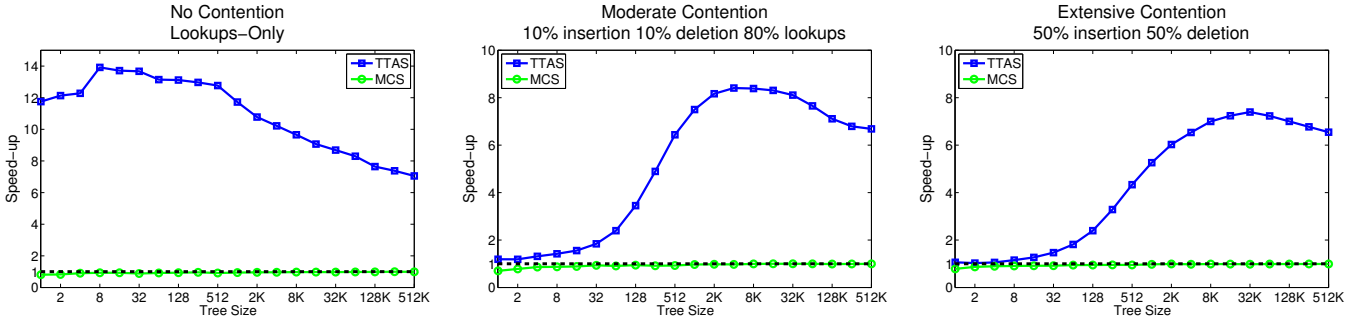



Figure 4: The HLE speedup of 8 threads with different types of locks. The base-line of each speedup line is the standard version of that specific lock (the horizontal dotted black line at y=1). By mixing different access operations we vary the amount of contention: (i) lookups only – no contention, (ii) moderate contention – 10% of the tree accesses are insertions and 10% are deletions and (iii) extensive contention – all the accesses are insertions or deletions.

XACQUIRE TAS and re-enters a speculative execution. Second, a newly arriving thread initially observes the lock as taken and spins. Once the thread in the critical section releases the lock, the waiting thread issues an XACQUIRE TAS as in the first case. The bottom line is that all threads are blocked from entering a speculative execution until the initial aborted thread exits the critical section, but then all the threads resume execution speculatively. The flip side of this behavior is that a thread may thus abort several times before successfully completing its operation, either speculatively or non-speculatively.

**MCS Fair lock (the circled line in Figure 2 )** The MCS lock represents the lock as a linked list of nodes, where each node represents a thread waiting to acquire the lock. An arriving thread uses an atomic SWAP [15] to atomically append its own node to the tail of the queue, and in the process retrieves a pointer to its predecessor in the queue. It then spins on the *locked* field of its node, waiting for its predecessor to set this field to false.

In the case of the HLE-based MCS lock the lemming effect is much worse because all the threads that were aborted form a chain, each spinning on a different location waiting for the predecessor to release the lock. Neither an aborted thread nor newly arriving thread can now enter the critical section speculatively. In either case the thread spins and once its turn arrives enters the critical section *non-speculatively*. Thus, a single abort causes the serialization of all concurrent critical sections, as well as newly arriving threads, all of which will now execute non-speculatively. Essentially, because of the fairness guarantees provided by the MCS lock, it "remembers" conflict events and makes it harder to resume a speculative execution. Even when the original lock holder releases the lock, it moves it into a state that does not allow new threads to speculatively execute. The MCS lock requires a *quiescence period*, in which no new threads arrive, so that all waiting threads acquire the lock, execute the critical section and leave. Only then does the MCS lock return to a state that allows the next arriving threads execute speculatively.

**Performance impact** In Figure 3 we divide the benchmark's execution into 1 millisecond time slots and show the throughput obtained in each slot, normalized to the throughput over the entire execution. We also show the fraction of operations that completed via a non-speculative execution in each time slot. As can be seen, TTAS performance can fluctuate severely, sometimes falling by as much as 2.5×. These throughput drops are correlated with periods in which more critical sections finish non-speculatively, i.e., after serialization caused by an abort. The MCS performance reinforces the results of the previous benchmark (Figure 2): the benchmark executes virtually all operations non-speculatively due to serialization caused by an abort. Finally, Figure 4 depicts the performance advantage of the lock elision usage with different types of locks.

As observed, MCS lock gains no benefit with HLE usage. On the other hand the TTAS lock gains performance boost while using the HLE mechanism.

# 5. SOFTWARE-ASSISTED LOCK REMOVAL

The lemming effect demonstrated in the previous section arises because an HLE transaction runs with the lock in its read set. That is, although HLE only pretends to hold a lock without globally locking, still the lock creates an abort chain effect once one thread aborts. One then wonders why bother touching the lock: if the hardware already detects conflicts (through the coherency protocol), how about starting the transaction without touching the lock at all? Two problems are created when we naively eliminate any reference to the lock, first the threads may easily get into a live-lock scenario, where they keep aborting each other due to data conflicts without any thread making progress. Second, if concurrently to the transactional threads another thread runs non-transactionally with the lock, some transactions may observe an inconsistent state (in which partial writes of the non-transactional thread are observed). In this section we take this approach of trying to eliminate any usage of the lock, sacrificing opacity but ensuring there is no live-lock.

Rajwar and Goodman observed [22] that given transactional capabilities, one can simply run transactions with the same scope as the critical section without accessing the lock. They relied on a hardware conflict management scheme that guarantees *starvation freedom* to prevent the live-lock problem. Unfortunately, Haswell's TM "requestor wins" conflict management policy guarantees neither starvation freedom nor livelock freedom [7].

Our solution to this progress problem is to go back to using the lock, as a progress-guaranteeing mechanism, but only at commit time. However, simply having an aborted thread acquire the lock (to avoid starvation) can lead to an incorrect execution due to the *non-atomicity* of a thread running in the critical section in a non-speculative manner, as depicted in the following *erroneous* example. Memory updates performed by such a thread are made globally visible one at a time, making it possible for concurrent transactions to observe an inconsistent state.

**Erroneous Example:** Consider two code segments protected by the same lock $L$, as depicted on the right. Suppose now that thread $T_1$ transactionally executes $C_1$ without accessing $L$ and reads $X = 0$. Now another thread, $T_2$, executes $C_2$ non-transactionally, acquiring $L$ and then storing 1 to $Y$. Following this $T_1$ reads $Y$ from memory. Since $Y$ is not in $T_1$'s read set, there is no conflict with $T_2$'s previous store and $T_1$ observes $Y = 1$. $T_1$ then commits. Thus $T_1$ observes an inconsistent state, $X = 0$ and $Y = 1$. (Had thread $T_2$ run transactionally in the above example, $T_1$ would observe either both or none of $T_2$'s stores.)

| $C_1$: | $C_2$: |
|---|---|
| `lock(L)` | `lock(L)` |
| `load(X)` | `store(Y,1)` |
| `load(Y)` | `store(X,1)` |
| `unlock(L)` | `unlock(L)` |

**Software-assisted lock removal (SLR, Figure 5)** Our solution to the non-atomicity problem uses the lock to verify that a transaction commits only if it has observed a consistent state. A transaction starts without accessing the lock. Once it reaches the end of the critical section, it reads the lock. If the lock is free, the transaction commits (no transaction wrote on a value read by it); otherwise, there is a concurrent non-speculative transaction, and the transaction aborts itself using XABORT. Returning to our example, the scenarios depicted in Figure 6 demonstrate how SLR enforces correct executions, though sometimes the transaction can observe an inconsistent state. This is usually not a problem, as the transaction

```
1  shared variables:
2    lock : speculative lock
3
4  thread local variables:
5    retries : int
6
7  lock() {
8    retries := 0
9    // speculative path
10   XBEGIN (Line 14) // jump to Line 14 on abort
11   return
12
13   // fallback path
14   retries ++
15   if ( retries < MAX_RETRIES)
16     goto Line 10
17   else
18     lock.lock() // standard lock acquire
19 }
20
21 unlock() {
22   if (XTEST()) { // returns TRUE if the run is speculative
23     if (lock is locked)
24       XABORT // aborts the speculative run
25     else
26       XEND
27   } else {
28     lock.unlock() // standard lock release
29   }
30 }
```

Figure 5: Software-Assisted Lock Removal

is sandboxed by the TM, but program correctness may be violated if inconsistent reads cause the transaction to compromise the lock check. For example, this can happen if the transaction erroneously writes to the lock itself, or jumps directly to an XEND instruction without checking the lock first, and so on. Therefore, to use SLR one must verify that the observable inconsistent states a transaction might encounter cannot cause the transaction to misbehave in such ways. Most common transaction types, such as those used in data structures and STAMP, are safe for SLR.

**Performance impact** SLR alleviates the two problems that lead to the HLE lemming effect. First, with SLR a thread that non-transactionally acquires the lock does not automatically cause all running transactions to abort. If it manages to complete the critical section before these transactions try to commit, they may successfully commit, given they do not conflict on the data. Second, while a thread is executing non-transactionally in the critical section, arriving threads continue to enter speculative transactional execution and are not forced to wait.

**Correctness** Haswell's TM conflict detection checks guarantee that once a transaction accesses a cache line, a conflicting access by a lock holding thread will abort the transaction. In the other direction, if the transaction accesses a memory location following a lock holder's access and yet successfully commits, then the lock was released before the transaction ended. Thus the transaction's execution is indistinguishable from the case in which the entire critical section (of the lock holder) ran before the transaction.

# 6. SOFTWARE-ASSISTED CONFLICT MANAGEMENT

In this section we propose a *software-assisted conflict management* (SCM) scheme which serializes conflicting threads that cannot run concurrently, but does this *without acquiring the lock* to

**Scenarios ending in abort**

| $T_1$ | $T_2$ |
|---|---|
| begin SLR txn | |
| `load(X)` | |
| | `lock(L)` |
| | `store(Y)` |
| `load(Y)` | |
| `load(L)` | |
| $L$ is locked: ABORT | |

| $T_1$ | $T_2$ |
|---|---|
| begin SLR txn | |
| `load(X)` | |
| | `lock(L)` |
| | `store(Y)` |
| | `store(X)` |
| ABORT: conflict | |
| on $X$ | |

**Successfully committing scenario**

| $T_1$ | $T_2$ |
|---|---|
| begin SLR txn | |
| | `lock(L)` |
| | `store(Y)` |
| | `store(X)` |
| `load(X)` | |
| | `unlock(L)` |
| `load(Y)` | |
| `load(L)` | |
| COMMIT | |

Figure 6: How SLR enforces correct executions in different scenarios.

avoid impact on the other speculatively running threads in the system. This scheme is compatible with any lock implementation, and resolves the lemming effect problem in HLE transactions without resorting to using lock removal, and without sacrificing opacity.

For example, when using lock removal in highly contended workloads with lots of aborts, new threads keep starting transactions and causing more aborts and wasted work. Here HLE's ability to grab the lock and stop newly arriving threads from entering speculative execution turns out to be extremely helpful and greatly improves performance (as shown in Section 7). Our conflict management scheme thus maintains this ability of HLE, and prevents the lemming effect in less contended scenarios.

**Preventing livelock, the SCM scheme:** Our scheme uses two locks, the original *main* lock which is taken using the HLE/SLR mechanism and an *auxiliary* standard lock which is only acquired in a standard non-transactional manner. The auxiliary lock groups all the threads that are involved in a conflict and serializes them (see Figure 8). When a transaction is aborted, the aborted thread non-transactionally acquires the auxiliary lock and then rejoins the speculative execution of the original critical section. The process of acquiring the auxiliary lock in order to rejoin the speculative run is called the *serializing path*. As with previous schemes, the thread may retry its transaction before going to the serializing path.

To see why this scheme prevents livelock, consider two transactions, $T_1$ and $T_2$, which repeatedly abort each other. Once $T_1$ acquires the auxiliary lock and re-joins the speculative execution, one of the following can happen: (1) $T_1$ aborts again, but $T_2$ commits, or (2) $T_2$ aborts and thus tries to acquire the auxiliary lock, where it must wait for $T_1$ to commit and release the auxiliary lock. Generalizing this, once a thread $T$ acquires the auxiliary lock any transaction that conflicts with $T$ either commits or gets serialized to run after $T$. Thus the system makes progress.

**Preventing starvation** In the above scheme starvation remains possible due to one of two scenarios: (1) a thread fails to acquire the auxiliary lock (as can happen with a TTAS lock), or (2) a thread holding the auxiliary lock fails to commit. To solve issue (1) we require that the auxiliary lock be a starvation-free (or "fair") lock, such as an MCS lock. Our scheme then inherits any fairness properties of the auxiliary lock. To solve issue (2), the auxiliary lock holder non-transactionally acquires the main lock after failing to commit a given number of times. If all accesses to the main lock go through the HLE/SLR mechanism, then *only* the auxiliary lock holder can ever try to acquire the main lock and is therefore guaranteed to succeed. Otherwise (i.e., if the program sometimes explicitly acquires the lock non-transactionally), the main lock must be starvation-free as well.

While SCM provides the most benefits when employed with HLE, the two schemes, SCM and SLR can be combined together to further reduce any progress problems caused when SLR threads give up and acquire the lock non-transactionally.

```
1  shared variables:
2    main_lock : elided main lock
3    aux_lock : auxiliary standard lock
4
5  thread local variables:
6     retries : int
7    aux_lock_owner : boolean, initially FALSE
8
9  lock() {
10    retries := 0
11   // primary path
12   XBEGIN (Line 17) // jump to Line 17 on abort
13    call HLE or SLR lock() as appropriate
14   return
15
16   // serializing path
17   if (aux_lock_owner = FALSE) {
18      retries ++
19   } else {
20     aux_lock.lock() // standard lock acquire
21     aux_lock_owner := TRUE
22   }
23   if ( retries < MAX_RETRIES)
24     goto Line 12
25   else
26     main_lock.lock() // standard lock acquire
27 }
28
29 unlock() {
30   if (XTEST()) { // returns TRUE if the run is speculative
31     call HLE or SLR unlock() as appropriate
32     XEND
33   } else {
34     main_lock.unlock() // standard lock release
35   }
36   if (aux_lock_owner = TRUE) {
37     aux_lock.unlock() // standard lock release
38     aux_lock_owner := FALSE
39   }
40 }
```

Figure 7: Software-Assisted Conflict Management

**Implementation and HLE compatibility (Figure 7)** Our scheme maintains HLE-compatibility by *nesting* an HLE transaction within an RTM transaction. When used with HLE, we first start an RTM transaction which "acquires" the lock with an XACQUIRE store. Because TSX provides a flat nesting model [2], an abort will abort the parent RTM transaction and execute the fall-back code instead of re-issuing the XACQUIRE store and aborting all the running transactions.

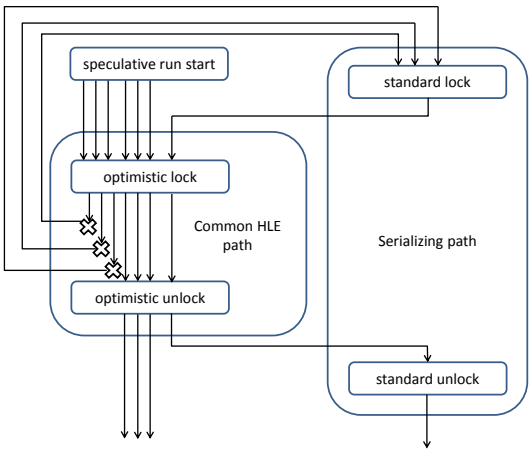



Figure 8: A block diagram of a run using our software scheme. The entry point of a speculative section is the 'speculative run' rectangle. All threads acquire the original main lock using the lock-elision mechanism. If a conflict occurs (described by 'x'), the conflicting threads are sent to the serializing path. Once a thread acquires the auxiliary standard lock in a non-speculative manner, it rejoins the speculative run.

Unfortunately, the initial implementation of TSX in Haswell does not support nesting of HLE within RTM. Therefore, in our experiments we use RTM to implement lock elision (by reading the lock address), which does not provide the self-illusion that the lock is taken. More precisely, in our current implementation we omit Line 31 of the Unlock() function in Figure 7, and perform the following at Line 12 of the Lock():

```
1: // put the main_lock in the read set
2: // and check that it is free
3: if (main_lock is locked) then
4:     XABORT('non-speculative run')
5: end if
```

**Remark** In principle, grouping the conflicting threads in one group may be too strict since a single conflicting thread does not have to conflict with the entire group. A natural extension (left for future work) to explore is dividing the conflicting threads to different groups, each containing only threads that conflict among themselves.

# 7. EVALUATION

In this section we evaluate the benefit provided by our lock elision schemes using two data structure benchmarks and applications from the STAMP suite (commonly used for evaluating hardware TM implementations [12, 24, 19]), which consists of eight applications that cover a variety of domains and exhibit different characteristics in terms of transaction lengths, read and write set sizes and amounts of contention. The premise of HLE is to enable simple coarse-grained programming with the performance of fine-grained locks, thus obviating the need for fine-grained locking. Therefore, we deliberately use coarse-grained benchmarks. Our experience with fine-grained benchmarks, such as those in the PARSEC [6] suite, is that in general applying HLE there shows little performance impact because the benchmarks are already optimized to avoid contention (see [4]).

**Methodology** We evaluate our methods on both the MCS lock and the TTAS lock. For each lock type we test the following six schemes: (1) Standard (non-speculative) version of the lock, (2) HLE version of the lock, (3) *HLE-retries* version, based on Intel's recommendations [1], in which a thread acquires the lock non-speculatively only after retrying speculatively 10 times, (4) HLE version of the lock with conflict management (*HLE-SCM*), (5) Optimistic SLR version, in which a thread only acquires the lock non-speculatively after retrying speculatively 10 times (*Opt SLR*), and (6) Optimistic SLR version with conflict management applied (*SLR-SCM*). As in Section 4, we use a Core i7-4770 3.4 GHz processor with 4 cores, each with 2 hyperthreads. We run the benchmarks on an otherwise idle machine using the `jemalloc` memory allocator which is tuned for multi-threaded programs.

**Conflict management tuning** Because SLR and HLE behave differently when the main lock is taken non-speculatively, we tune the conflict management as appropriate for each technique. Taking the lock non-speculatively in an HLE-based execution has large performance impact, and so the thread holding the auxiliary lock retries to complete its operation speculatively 10 times before giving up and acquiring the main lock. In contrast, SLR is much less sensitive to the main lock being taken and so if the bits in the abort status register indicate the transaction is unlikely to succeed, we switch to a non-speculative execution. We have verified that using other tuning options only degrade the schemes' performance.

## 7.1 Red-black Tree Data Structure Benchmark

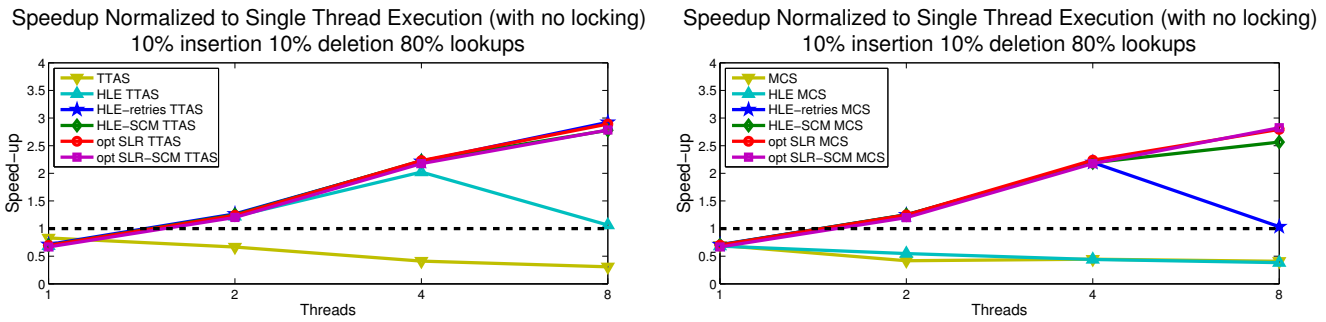
We evaluate our methods using two data structure benchmarks, the red-black tree (described in Section 4) and a hash table. In each test, we measure the average number of operations per second (throughput) when running the benchmark 20 times on an otherwise idle machine. The results of the two data structure benchmarks are comparable, as hash table transactions are always short and therefore "zoom in" on the short transaction portion of the red-black workload spectrum. We therefore discuss only the red-black tree.

**Red-black tree** Figure 9 shows the *speedup* (relative to a single thread with no locking) obtained by the various methods on a 128-node tree under moderate contention (20% updates). With HLE the MCS lock does not scale at all, and even the TTAS does not scale past 4 threads. Simply adding speculative retries allows HLE-TTAS to scale, but when the lemming effect is more severe—as with HLE-MCS at 8 threads—merely retrying the transactions no longer works. We believe that with higher amounts of concurrency, the simple retry policy will not be effective even with TTAS. In contrast, using our schemes, the throughput always scales with the number of threads.

Figure 10 depicts the *speedup* that our methods obtain (relative to the HLE version of the specific lock) across the full spectrum of workloads. Notice that increasing the tree size also increases the size of the critical section, resulting in a lower conflict probability but also lower throughput. Our software schemes improve the speedup compared to the plain HLE version of the specific lock (especially on fair locks).

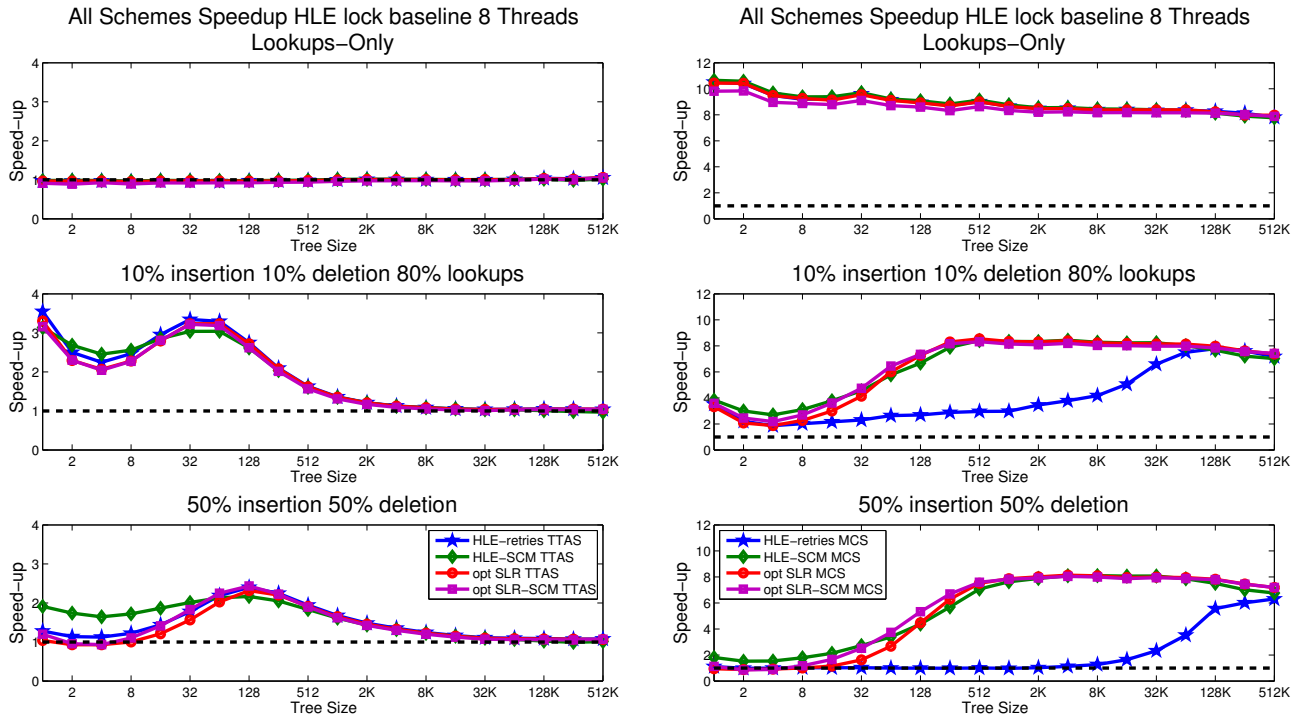
**TTAS lock** On the lookup only (no contention) workload, the HLE-based TTAS is good enough, and none of the software-assisted methods improve on its performance. However, as we increase the level of contention by increasing the fraction of mutating operations, our methods outperform the plain HLE-based TTAS by up to 3.5×. This is the result of letting new arriving threads immediately enter the critical section speculatively, instead of waiting for the aborted thread currently in the critical section to leave.

In general, the HLE-SCM and SLR versions of TTAS give comparable performance, which is also similar to the simple HLE-retries policy. The exception is short transactions, where HLE-SCM out-




(a) TTAS lock (b) MCS lock

Figure 9: The execution results on a small tree size (128 nodes) under moderate contention. The two graphs are normalized to the throughput of a single thread with no locking (the horizontal dotted black line at y=1). The software assisted schemes scale well and the performance gap between MCS and TTAS is closed.




(a) TTAS lock (b) MCS lock

Figure 10: The speedup of our generic software lock elision schemes compared to Haswell HLE. The base-line of each speedup line is the HLE version of that specific lock (the horizontal dotted black line at y=1): on the left – TTAS lock and on the right – MCS lock. Since the performances are scaled using different base lines, the reader can not compare between the performance of the different lock types.

performs SLR, SLR-SCM and HLE-retries by up to 2×, exactly because of the serialization it induces.

**MCS lock** Our software-assisted schemes increase throughput by $2-10\times$ in every MCS workload (even in a read-only workload, the MCS lock experiences a severe lemming effect behavior due to spurious aborts). We again see comparable results for HLE-SCM and SLR, with a 50% advantage to HLE-SCM in short transactions. In contrast to the TTAS case, here the HLE-retires policy does not alleviate the lemming effect. In fact, under high contention HLE-retries does not do better than the plain HLE. Our software-assisted methods thus outperform HLE-retries significantly, by 4× under moderate contention (20% updates) and $2-10\times$ under high contention.

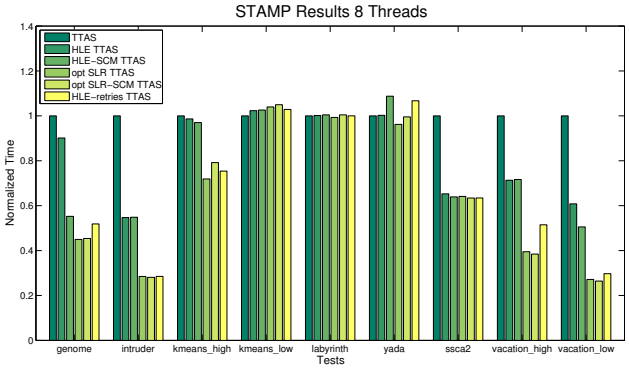
**Analysis** Detailed analysis of the number of attempts per successful operation and fraction of operations that complete in a speculative execution is deferred to [4], due to space constraints.

## 7.2 STAMP

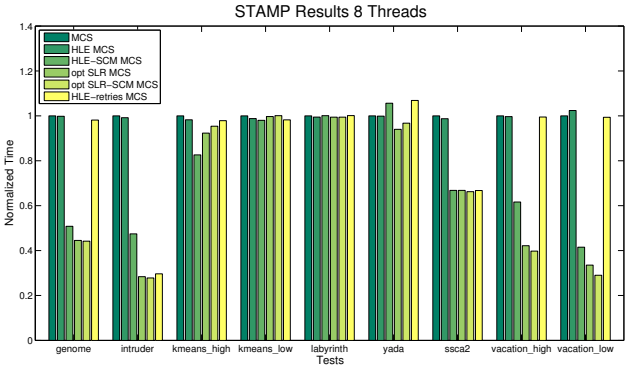
To apply our methods to the STAMP suite of benchmark programs [10], we replace the transactions with critical sections that use the same global lock. Figure 11 shows the runtime of the STAMP programs at maximum concurrency (8 threads) with the various lock elision methods, normalized to the execution time using the plain non-speculative lock. (We exclude the `bayes` benchmark due to its non-deterministic behavior, as done in prior work using STAMP [13].)

As with the red-black tree data structure benchmark, MCS lock gains no benefit from HLE usage. But, MCS lock provides considerable benefit when used with HLE combined with our conflict management scheme. The HLE-SCM scheme typically improves the performance by up to 2.5×.

On the other hand, TTAS lock gains some benefit of HLE usage (up to 2× in `intruder`) but the HLE-SCM scheme with TTAS gives modest improvement with the exception of `genome` (up to




(a) Relative execution time over non-speculative TTAS lock




(b) Relative execution time over non-speculative MCS lock

Figure 11: Normalized run time of STAMP applications (**lower is better**) using standard locking, HLE, and our software-assisted methods.

1.5×). Here too we see that the benefit of HLE usage in TTAS depends on the workload's characteristics.

Both locks benefit considerably from lock removal usage. In most of the tests, the optimistic SLR scheme gives the highest improvement (with one exception of `kmeans-high` MCS HLE-SCM lock), sometimes up to 2× compared to the HLE-based scheme and up to 4× compare to the plain non-speculative version of the lock. The HLE-retries scheme generally performs comparable to SLR, except on `genome`, `yada` and `vacation` where it is slower: marginally so on TTAS (about 10%-30%), but drastically slower when using MCS, where HLE-retries obtains almost no speedup over the standard MCS lock.

For the most part, when the SCM scheme is used with SLR, the performance gain is negligible with only one exception. In the `vacation-low` test, the SLR-SCM gives 15% improvement over the Opt-SLR.

**Summary:** Our SCM scheme improves the performance of HLE-based locks in both data structure benchmarks and STAMP no matter what the contention level is. MCS lock (or any other fair lock) gains the highest performance boost since these locks need quiescence period in order to overcome the lemming effect and return to speculative execution. The impact of our SCM method on the SLR is more modest and depends on the characteristics of the execution (such as transaction length and contention level).

# 8. CONCLUSION

We have described several lightweight software techniques to considerably improve the performance of lock-based code when executed over HTM. Moreover, our techniques, enable HLE-based fair locks with starvation freedom and progress guarantee and with no performance degradation. Our evaluation on a Haswell processor is encouraging, improving the run time of applications from the STAMP suite by up to 3.5 times and of data structure benchmarks by up to 10 times.

As we show, even very few aborts are enough to trigger the lemming effect for some lock implementations. Hence, any processor that may cause the HLE to abort and acquire the lock non-speculatively is likely to benefit from our techniques.

In the future, we plan to explore more refined conflict management policies. In particular, utilizing abort information provided by the hardware (such as the location in which a conflict occurs, and/or the identify of the conflicting thread) appears to be a promising direction.

# Availability

Our implementation is available on Tel Aviv University's Multicore Algorithmics group web site at `http://mcg.cs.tau.ac.il/`.

# Acknowledgments

This work was supported by the Israel Science Foundation (grants 1386/11 and 1227/10) and by Yad-HaNadiv foundation. Adam Morrison is supported in part at the Technion by an Aly Kaufman Fellowship.

# 9. REFERENCES

[1] Intel 64 and IA-32 Architectures Optimization Reference Manual.

[2] Intel Architecture Instruction Set Extensions Programming Reference.

[3] Y. Afek, A. Levy, and A. Morrison. Programming with hardware lock elision. In *PPoPP 2013*.

[4] Y. Afek, A. Levy, and A. Morrison. Software-Improved Hardware Lock Elision. Technical report, Tel Aviv University.

[5] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *DISC 2012*.

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA 2007*.

[8] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *ISCA 2013*.

[9] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *TRANSACT 2014*.

[10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC 2008*.

[11] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, Department of Computer Science and Engineering, University of Washington, 1993.

[12] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems, 2009.

[13] N. Diegues and P. Romano. Time-warp: Lightweight Abort Minimization in Transactional Memory. In *PPoPP 2014*.

[14] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP 2008*.

[15] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13:124–149, January 1991.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*.

[17] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *ISPP '94*.

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.

[19] D. Papagiannopoulou, G. Capodanno, R. I. Bahar, T. Moreshet, A. Holla, and M. Herlihy. Energy-Efficient and High-Performance Lock Speculation Hardware for Embedded Multicore Systems. In *TRANSACT 2013*.

[20] N. Piggin. x86: FIFO ticket spinlocks. `http://lkml.org/lkml/2007/11/1/125`, 2007.

[21] R. Rajwar and J. R. Goodman. Speculative Lock Elision: enabling highly concurrent multithreaded execution. In *MICRO 2001*.

[22] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS 2002*.

[23] A. Roy, S. Hand, and T. Harris. A runtime system for software lock elision. In *EuroSys 2009*.

[24] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *PACT 2012*.

# APPENDIX

## A. ADJUSTING LOCKS TO WORK WITH HLE: DETAILS

**Ticket Lock (Figure 12) Adjustments** The new implementation (Figure 13) handles both speculative and standard (non-speculative) runs. We use a compare-and-swap (CAS) primitive [15] in order to distinguish between the two cases: the release attempts to CAS the lock back to its original value, i.e., decrement the *next* counter instead of incrementing the *owner*. If successful, it removes all traces of the lock acquisition; this occurs in either a speculative execution or a non-speculative single-thread execution. An unsuccessful CAS indicates a standard run with multiple requesters.

The only difference in the lock acquiring function is the XACQUIRE usage. In the adjusted unlock function (Figure 13), if Line 8 is successfully executed, either: (1) the lock is taken in a standard manner and the lock owner is the only running thread (no other requesters) or (2) the lock is taken in speculative manner and the lock owner (can be one of many) removes all traces of its run. Line 9 is used to release the lock when the lock is taken in a standard manner and the lock owner is not the only requester. This behavior is identical to the original implementation.

**CLH Lock (Figure 14) Adjustments** The pseudo-code of the CLH lock implementation, adjusted to the HLE mechanism, is depicted in Figure 15. As in the ticket lock, we need to adjust the CLH lock so that the lock reverts to its original state when released in a solo run. Again, we use a CAS to do this, in an attempt to place *pred* at the tail of the queue, effectively erasing the presence of our node.

```
1  shared variables:
2    next : integer , initially 0
3    owner : integer , initially 0
4
5  lock() {
6    current := F&A(&next, 1)
7    while (owner ≠ current) {
8      // busy wait
9  } }
10
11 unlock() {
12   owner := owner + 1
13 }
```

Figure 12: Ticket lock.

```
1  lock() {
2    current := XACQUIRE F&A(&next, 1)
3    while (owner ≠ current) {
4      // busy wait
5  } }
6
7  unlock() {
8    if (!XRELEASE CAS(&next, owner+1, owner)) {
9      owner := owner + 1
10 } }
```

Figure 13: Lock elision adjusted ticket lock.

```
1  shared variable:
2    struct Node {
3      locked : 1 bit (boolean)
4      next : pointer to Node
5    }
6    tail : pointer to Node, initially points to T =< FALSE, NULL >
7
8  thread-local variables:
9    myNode : pointer to Node, initially points to thread−local Node
10   pred : pointer to Node
11
12 lock() {
13   myNode.locked := TRUE
14   pred := SWAP(&tail, myNode)
15   while (pred.locked) {
16     // busy wait
17 } }
18
19 unlock() {
20   myNode.locked := FALSE
21   myNode := pred
22 }
```

Figure 14: CLH lock.

```
1  lock() {
2    myNode.locked := TRUE
3    pred := XACQUIRE SWAP(&tail, myNode)
4    while (pred.locked) {
5      // busy wait
6  } }
7
8  unlock() {
9    if (!XRELEASE CAS(&tail, myNode, pred)) {
10     myNode.locked := FALSE
11     myNode := pred
12 } }
```

Figure 15: Lock elision adjusted CLH lock.