TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Programming with Hardware Lock Elision

Dissertation submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Amir Levy

The research work for this thesis has been carried out at Tel-Aviv University
under the supervision of Prof. Yehuda Afek
and the consultation of Mr. Adam Morrison

September 2013

Abstract

This thesis addresses performance problems in *hardware lock elision* (HLE), which is being introduced into commercial processors. Using Intel's Haswell HLE as a study vehicle, we show that even a few transactional aborts can severely limit the amount of concurrency and speedup obtained using HLE. We then provide a software-based technique to solve this problem and restore the lost potential concurrency in lock elision executions.

We present a lock elision approach based on Haswell's transactional memory support that serializes only conflicting threads, allowing non-conflicting threads to continue their speculative run. To do this we add a *serializing path* to the lock implementation, in which a thread experiencing a conflict acquires a distinct *auxiliary lock* (without using lock elision) and then *rejoins the speculative execution*.

We evaluate our methods on a Haswell processor, using a set of data structure benchmarks and applications from the STAMP suite. Our methods lead to performance improvement of up to 3.5× on STAMP and up to 10× on the data structure benchmarks, compared to using Haswell's hardware lock elision as is.

We also describe how to extend Haswell's HLE mechanism to achieve a similar effect to our software-assisted scheme entirely in hardware, by distinguishing between conflicts on the lock and on the data cache lines. Our proposal requires no cache-coherence protocol changes.

Acknowledgements

I am deeply grateful to my advisor, Prof. Yehuda Afek for his insightful comments, suggestions and warm encouragement. I would particularly like to thank Mr. Adam Morrison, I have greatly benefited from our joint work.

Finally, I would like to thank my wife Noga for her constant and consistent support, encouragement and good advice during the accomplishment of this work and in general.

Contents

Chapter 1

Introduction

Transactional memory (TM) [12] is being introduced into mainstream mass-market processors (e.g., Intel's latest Haswell microarchitecture, IBM POWER architecture [1, 7]), and practitioners try to use the offered hardware-based *lock elision* [17], a hardware mechanism to dynamically remove unnecessary lock-induced serialization, to improve the performance of their lock-based programs [2].

The premise of Hardware Lock Elision (HLE), and the scope of this work, is to enable simple coarse-grained programming with the performance of fine-grained locks. Hence, our work studies problems and solutions to gain the potential concurrency in coarse-grained global lock based programs.

The idea behind lock elision is to transactionally execute lock-protected critical sections, and serialize them if an actual data conflict occurs. (Two memory operations *conflict* if they both access the same cache line and one of them is a write.) This allows programmers to use coarse-grained locking, which is easier to program and reason about, while getting the performance of fine-grained locking.

However, as this work demonstrates, 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 of the hardware which cause excessive serialization and prevent the materialization of the full concurrency in the application from being exposed. Our techniques significantly improve the performance obtainable when using lock elision.

In this work we use Haswell's hardware lock elision (HLE) as an example. We begin by studying the performance behavior of several lock implementations under Haswell's HLE. We find that the

HLE mechanism can lead to an *avalanche behavior* pathology which causes unnecessary serialization and limits the concurrency exposed. We then propose several simple software-assisted lock elision algorithms to address these problems.

Algorithm 1 TTAS (Test&Test&Set) Lock Using HLE

TTAS Lock

```
1: while true do
2:    while (lock == 1) do
3:       {busy-wait}
4:    end while
5:    ret = XACQUIRE lock.get&set(1)
6:    if (ret == 0) then
7:       return
8:    end if
9: end while
```

TTAS Unlock

```
1: XRELEASE lock = 0
```

Figure 1.1: Applying hardware lock elision to a TTAS lock.

With standard HLE (Figure 1.1), one prefixes the store instruction acquiring the lock with a new XACQUIRE prefix and the store releasing the lock with an XRELEASE prefix. Executing an XACQUIRE store initiates a speculative transactional execution in which the hardware *elides* the lock acquisition, treating it as a load instead. This allows multiple transactions to run concurrently without conflicting on the lock. In a successful conflict-free execution, the XRELEASE store which ends the critical section causes the transaction to commit, making its memory updates globally visible.

Thus, Haswell's HLE mechanism conservatively requires that the store releasing the lock restores the lock to its original state prior to the acquisition [1]. Unfortunately, the popular (fair) ticket lock [14] (used in the Linux kernel [16]) and CLH lock [13, 9] 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.

However, our main contribution concerns HLE's *avalanche behavior* (Chapter 3). When a transaction aborts (e.g., due to a conflict) the execution rolls back to the acquiring store which is

now re-issued *non-transactionally*. In other words, an abort causes the thread to try to enter the critical section in a standard manner. The globally visible lock acquisition by the aborted thread conflicts with the speculative loads of the lock performed by speculative HLE transactions, and *causes all the threads that are in a transaction to abort*. In addition, *new threads arriving at the critical section see that the lock is taken*. This in turn causes such threads to delay their entrance into a transactional execution, or even (in the case of fair locks) serializes the run until a quiescent period in which no thread tries to access the lock.

To prevent the avalanche problem in HLE transactions, we introduce a simple yet effective *software-assisted conflict management* (SCM) 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 acquires a distinct *auxiliary lock* (without using lock elision) and then rejoins the speculative execution with the other threads. Using this approach conflicting threads are serialized among themselves and do not interfere with other threads, greatly reducing the probability that a thread aborts so many times that it must give up and acquire the original lock. 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.

While straightforward, implementing our lock elision technique is more complex than simply adding an instruction prefix as in Haswell's native HLE. We therefore describe in Chapter 7 how to extend (what we believe to be) Haswell's HLE implementation to achieve a similar effect to our lock elision scheme entirely in hardware, by distinguishing between conflicts on the lock and the data cache lines. We show how such a mechanism can be used to allow speculative sections that do not conflict on data lines to continue with their speculative runs even in the presence of conflicting threads that abort, acquire the lock, and serialize.

The contributions of this thesis:

- Analyzing the performance dynamics of Haswell's HLE and quantifying the impact of its avalanche behavior (Chapter 3).
- Solving the avalanche effect using lightweight software-assisted conflict management (Chapter 4).
- Evaluating the lock elision schemes which demonstrates that they can improve performance by

up to 3.5× in application benchmarks and up to 10× in data structure benchmarks compared to HLE (Chapter 5).

- HLE compatible locks: adapting ticket and CLH locks to HLE (Chapter 6).
- Extending Haswell's HLE implementation to achieve a similar effect to our lock elision scheme entirely in hardware (Chapter 7).

This thesis summarizes our two papers: [3, 4].

In the future We plan to explore more refined policies for handling conflict management. 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.

Chapter 2

Transactional Synchronization Extensions

2.1 Overview

In this background section we describe the specification of Intel's transactional synchronization extensions (TSX) [1], the commercial name for Intel's TM. We then describe the initial implementation of this specification in the Haswell processor.

A *memory transaction* is an instruction sequence whose memory accesses appear as if they were executed atomically at some point in time, i.e., a transaction appears to execute instantaneously without observing any concurrent memory updates.

The set of cache lines read/written by a transaction is called its *read/write set*. A *conflict* occurs if another processor reads or writes to a cache line in the transaction's write set, or writes to a cache line in the transaction's read set.

During its execution the processor may *abort* a transaction. (for example, due to a data conflict). Upon an abort the processor *rolls back* the transactional execution, restoring the processor's state to the point before the transaction started (including discarding any memory updates it performed). Execution then continues non-transactionally. TSX makes no progress guarantees and is allowed to abort a transaction even if no conflict occurs.

TSX defines two interfaces to designate the scope of a memory transaction, i.e., its beginning

and end:

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 instruction writing to memory (e.g., a store or `compare-and-swap`) which is prefixed by XACQUIRE (see Figure 1.1), the processor starts a transaction and *elides* the actual store, treating it as a transactional read instead (i.e., placing the lock cache line 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 by the XACQUIRE-prefixed instruction. 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.

Restricted transactional memory (RTM): RTM provides a 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. The XBEGIN instruction takes an operand that points to fall-back code which is executed if the transaction aborts. Upon an abort the processor writes an *abort status* to one of the registers, which the fall-back code can then consult. For example, the abort status indicates whether the abort was due to an XABORT, a data conflict, or an "internal buffer overflow" [1].

RTM can be used to implement custom lock elision algorithms [2] 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 cacheline is indistinguishable from any other line in the read-set.

2.2 Haswell's TSX Implementation

Various statements in Intel's documentation shed light on Haswell's TM implementation [2]. Of importance to us is that Haswell appears to use a *requestor wins* conflict management policy. If a

coherency message (read or write) arrives for a cache line in the write set, the transaction aborts. Similarly, if an eviction due to a write arrives for a cache line in the read set, the transaction aborts.

Experiments we conducted show that transactions are prone to *spurious aborts* that are not explained by data conflicts or read/write set overflow. The existence of spurious aborts is a serious issue because, as the next section explains, an abort can negatively impact performance on HLE executions. Spurious aborts imply that even in a perfect conflict free workload, such performance degradation is possible.

Read and write set tracking According to Intel's manuals, "the processor tracks both the read-set addresses and the write-set addresses in the first level data cache (L1 cache) of the processor." Therefore, running out of cache space (due to capacity or associativity) can cause a transactional abort.

Conflict management Quoting the manual:

- "Data conflicts are detected through the cache coherence protocol."
- "Data conflicts cause transactional aborts."
- "The thread that detects the data conflict will transactionally abort."

This amounts to a *requestor wins* policy. If a coherency message (read or write) arrives for a cache line in the write set, the transaction aborts. Similarly, if an eviction due to a write arrives for a cache line in the read set, the transaction aborts.

Transactional behavior in practice To better understand transactional behavior in practice, we conduct an experiment on a Haswell processor. We use a Core i7-4770 3.4 GHz processor. This processor has 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.

We run a benchmark that initiates a transaction which reads (or writes) every cache line in an array of a given size. We repeat this 10^7 times for various sizes and collect the number of transactions that successfully commit. Figure 2.1 shows the results: It is clear that the L1 cache, whose size is 32 KB, holds the write set. Transactions that attempt to write more than 32 KB of data always abort. However, it appears that a more sophisticated mechanism is used for tracking the read set, as transactions succeed for sizes that exceed not only the L1 cache but also the 256 KB L2 cache.

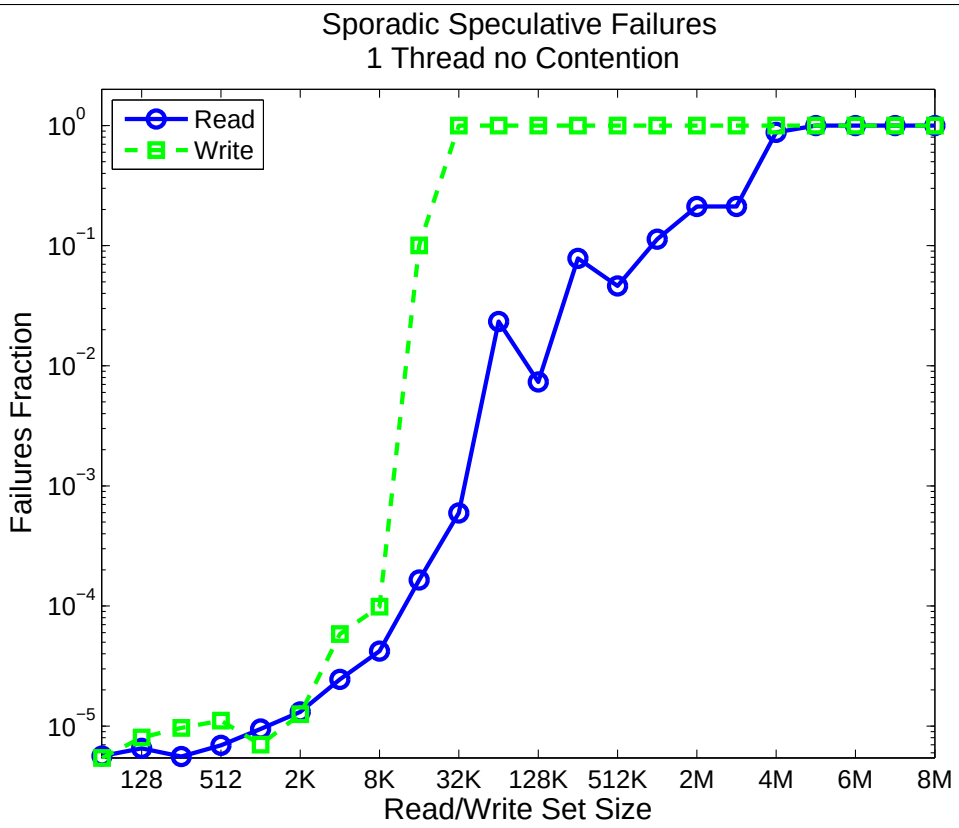


Figure 2.1: Transactional behavior in practice.

Spurious aborts A final observation from this benchmark is that transactions are prone to *spurious aborts*, i.e., aborts which are not explained by data conflicts or read/write set overflow. This is evidenced by the non-zero abort probability that exists even when the array the benchmark accesses fits well in the L1 cache. The existence of spurious aborts is important because, as the next section explains, an abort can negatively impact performance on HLE executions. Spurious aborts imply that even in a perfect conflict free workload, such degradation remains possible.

Chapter 3

HLE Avalanche Effect

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 (Algorithm 1) and the fair HLE-based MCS lock (Algorithm 2). 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.

Our experiments consist of threads accessing a red-black tree data structure which is 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 `insert`s and `delete`s 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 the

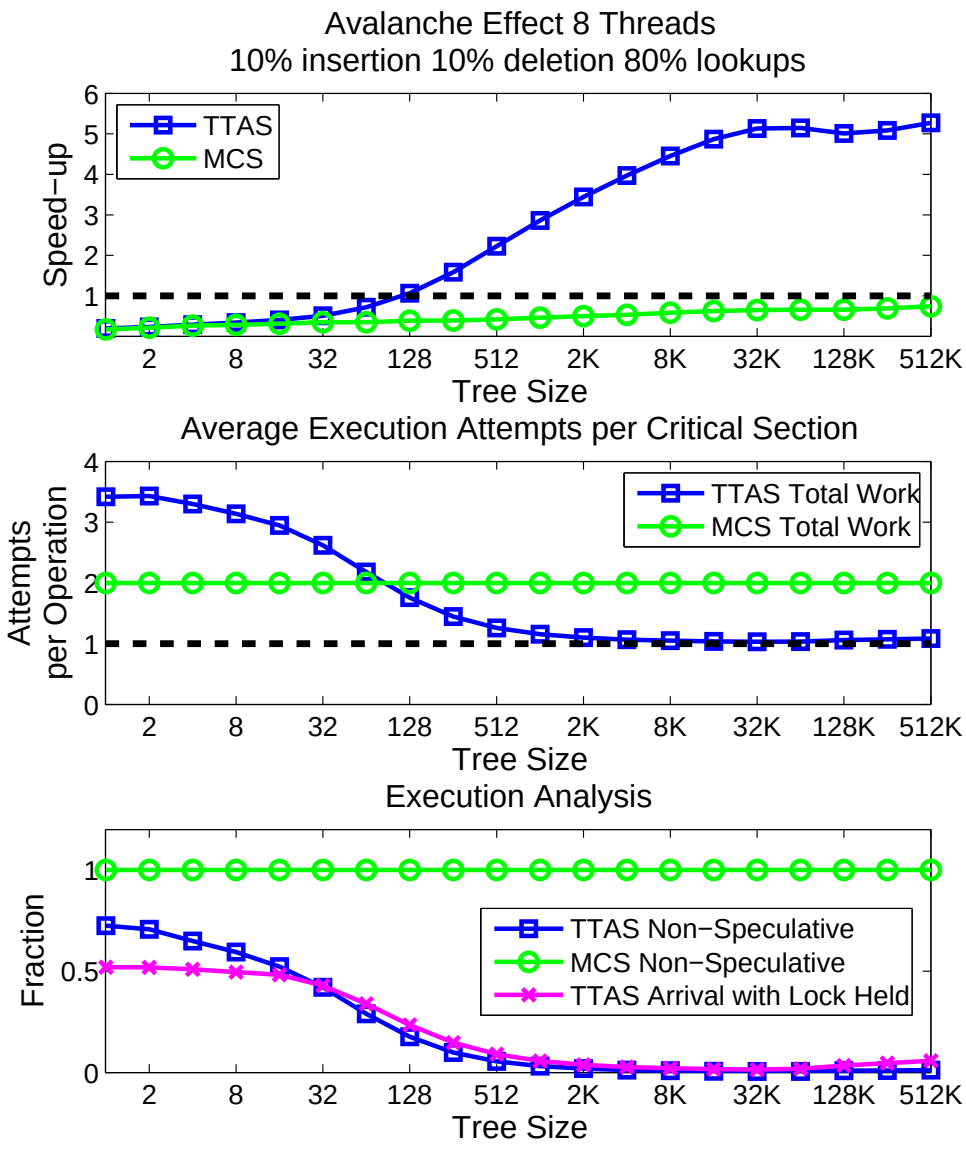


Figure 3.1: Impact of aborts on executions under different lock implementations. For each tree size we show the average number of times a thread attempts to execute the critical section until successfully completing a tree operation, and the fraction of operations that complete non-speculatively. CLH and ticket results are omitted, as they are similar to the MCS lock results.

following: (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 operations that complete via a normal (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.

Figures 3.1 , 3.3 , and 3.4 depict the avalanche effect during an HLE execution. Figure 3.1 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.

Figure 3.1 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 (Algorithm 1, and the boxed line in Figure 3.1) The first thread to abort successfully 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 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.

Fair lock (represented by the MCS lock (Algorithm 2, and the circled line in Figure 3.1)) 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 [11] to atomically append its own node to the tail of the queue, and in the process retrieves a pointer to its predecessor

Algorithm 2 MCS Lock Using HLE

Require:
initialization: tail = NULL
local variables: myNode, pred

MCS Lock

```
1: myNode.locked = true
2: myNode.next = NULL
3: pred = XACQUIRE SWAP(tail, myNode)
4: if (pred != NULL) then
5:    pred.next = myNode
6:    while (myNode.locked) { busy-wait }
7: end if
```

MCS Unlock

```
1: if (myNode.next == NULL) then
2:    ret = XRELEASE CAS(tail, myNode, NULL)
3:    if (ret) then
4:       return
5:    else
6:       while (myNode.next == NULL) { busy-wait }
7:    end if
8: end if
9: myNode.next.locked = false
```

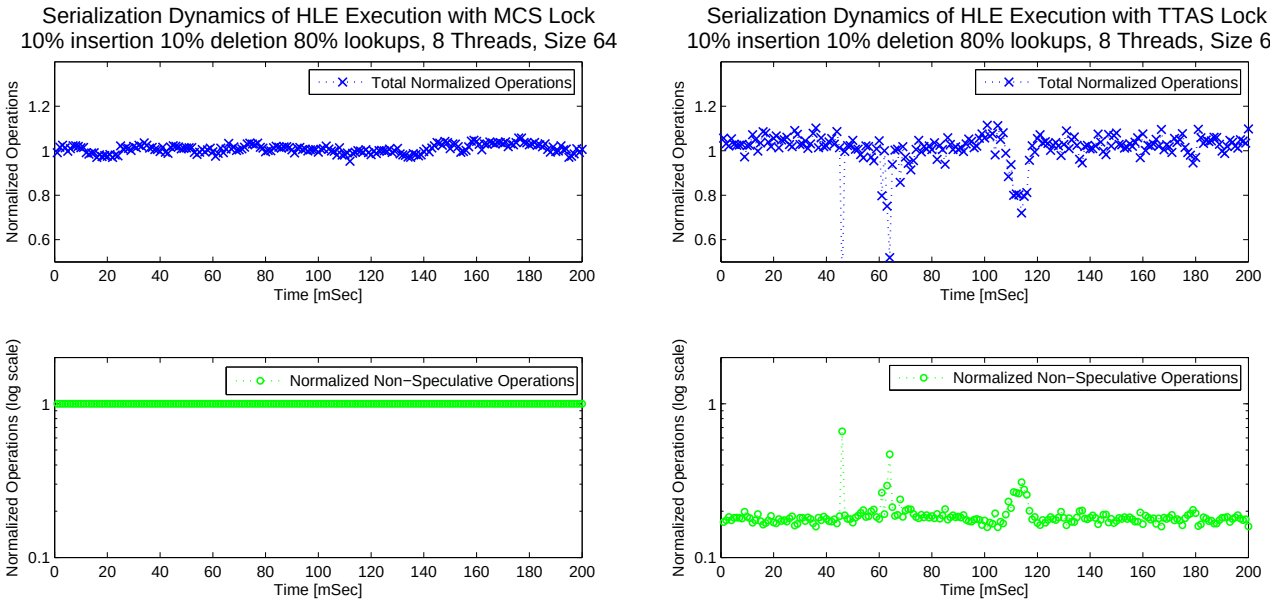
Figure 3.2: Applying hardware lock elision to a MCS lock.

in the queue. It then spins on the *locked* field of its node, waiting for its predecessor to set this field to false.

Similarly to the TTAS lock, the first thread to abort acquires the lock (line 3) and causes all subsequent threads to spin. In contrast to the TTAS lock, in the MCS lock spinning threads announce their presence, which leads to an avalanche effect that makes it hard to recover and re-enter speculative execution.

Consider first a thread that aborted because of the lock acquisition. The processor re-issues its acquire SWAP operation which returns the thread‘s turn in the queue. The thread then spins and once its turn arrives (its predecessor sets its *locked* field to false) enters the critical section *non-speculatively*. Thus, a single abort causes the serialization of all concurrent critical sections, which will now execute non-speculatively.

Now consider a newly arriving thread. It executes an XACQUIRE SWAP to obtain its turn.



(a) MCS lock: all operations complete non-speculatively.

(b) TTAS lock: most operations complete speculatively but there are periods of serialization.

Figure 3.3: 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.

However, it sees a state in which a lock is held and must therefore spin (in the speculative execution) waiting for the lock to be released. As a result, its speculative execution is doomed to abort: when the thread's predecessor releases the lock, the releasing write conflicts with the reads performed in the waiting thread's spin loop. In fact, the speculative execution may abort earlier if the spin loops issues a `PAUSE` instruction, as is often the case. In this case, as discussed above, the thread executes the critical section non-speculatively.

Essentially, because of the fairness guarantees the MCS lock provides, 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 speculative execution.

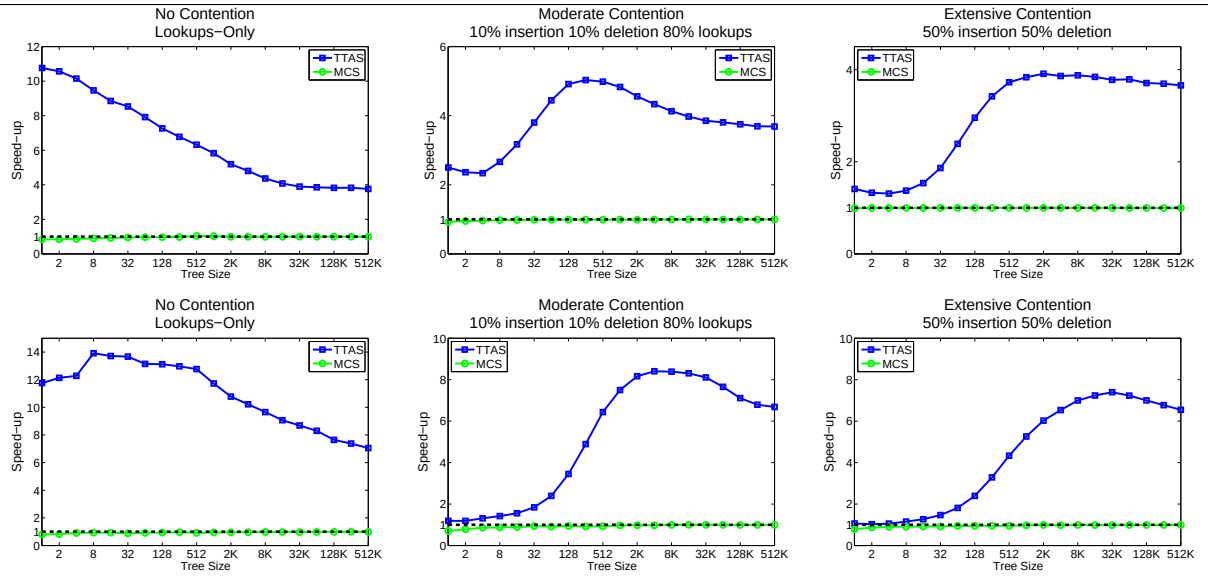
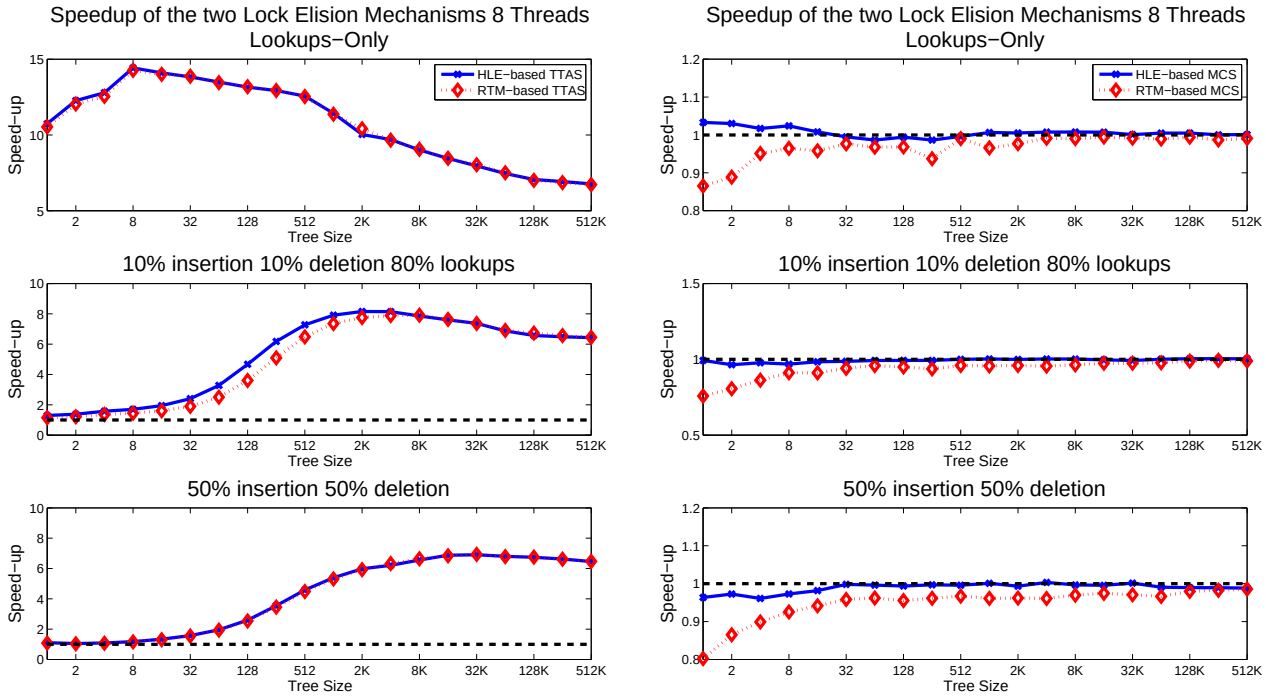


Figure 3.4: The HLE speedup of 8 threads with different types of locks. The base-line of each speedup line is the standard version of that specific lock (the horizontal dotted black line at y=1). By mixing different access operations we vary the amount of contention: (i) lookups only - no contention, (ii) moderate contention - a tenth of the tree accesses are node insertions and another tenth are node deletions and (iii) extensive contention all the accesses are either node insertion or deletion.

Performance impact In Figure 3.3 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: the benchmark executes virtually all operations non-speculatively due to serialization caused by an abort.

Finally, Figure 3.4 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.

The two software schemes presented in the following sections eliminate the serialization effect described here, improving the performance not only of the MCS lock but also of the HLE-based TTAS.



(a) TTAS lock

(b) MCS lock

Figure 3.5: The performance differences between the two lock elision mechanisms. The base-line of each speedup line is the standard version of that specific lock (the horizontal dotted black line at y=1): on the left - TTAS lock and on the right - MCS lock.

Remark It is not possible to count aborts when using Haswell's HLE, since with HLE an abort results in a re-issue of the XACQUIRE write, which is completely opaque to the lock implementation. Therefore, in our tests we use an equivalent lock elision mechanism based on the RTM instructions, which allows us to count aborts before re-issuing the acquiring write. We have verified that the performances of the two lock elision mechanisms are comparable (Figure 3.5).

Chapter 4

Software-Assisted Conflict Management

In this section we introduce the *software-assisted conflict management* (SCM), a simple yet effective lock elision scheme, which mitigates aborts serializing effect of HLE and allows to maintain higher levels of concurrency despite conflicts. The conflict management scheme serializes conflicting threads that cannot run concurrently, but does this *without acquiring the lock* to avoid impact on the other threads in the system. The scheme is *compatible with any lock implementation.*

Our scheme uses two locks, the original *main* lock which is taken using the HLE 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 4.1). 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. We refer to the process of acquiring the auxiliary lock in order to rejoin the speculative run as the *serializing path* (see the flow-chart in Figure 4.2). The thread may retry its transaction before going to the serializing path.

When applied to HLE our conflict management scheme prevents the problem in which an abort causes the lock to acquired, aborting all concurrent transactions in the process, hence resolves the avalanche problem in HLE transactions.

One usability advantage of software-assisted conflict management is that, like HLE, it provides

a transaction the illusion that the lock is acquired while it runs. As a result, one can plug our scheme into a legacy lock-based application by changing only the locking library.

Preventing livelock 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. 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

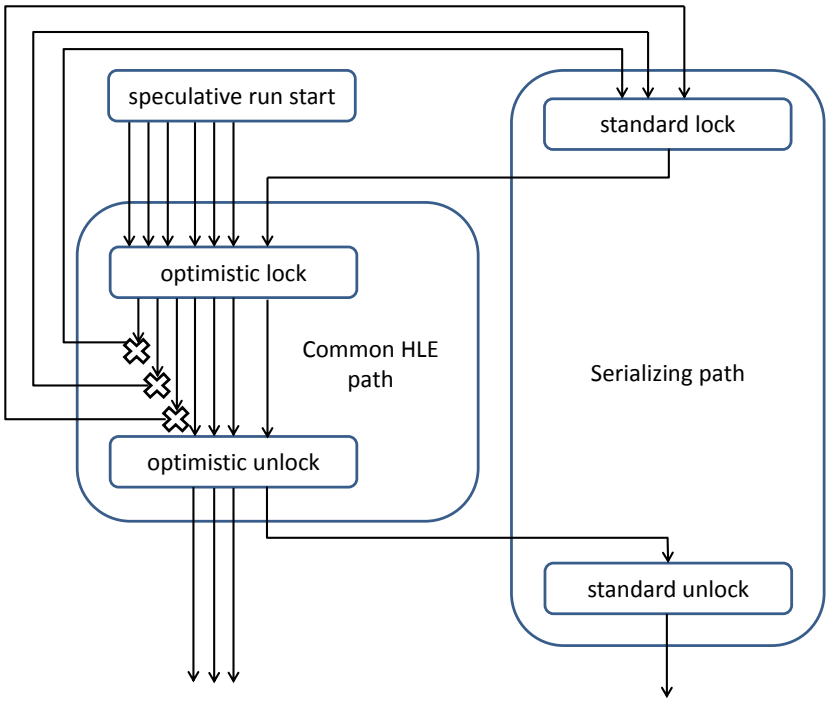


Figure 4.1: A block diagram of a run using our software scheme. The entry point of a speculative section is the 'speculative run' rectangle. All threads acquire the original main lock using the lock-elision mechanism. If a conflict occurs (described by 'x'), the conflicting threads are sent to the serializing path. Once a thread acquires the auxiliary standard lock in a non-speculative manner, it rejoins the speculative run.

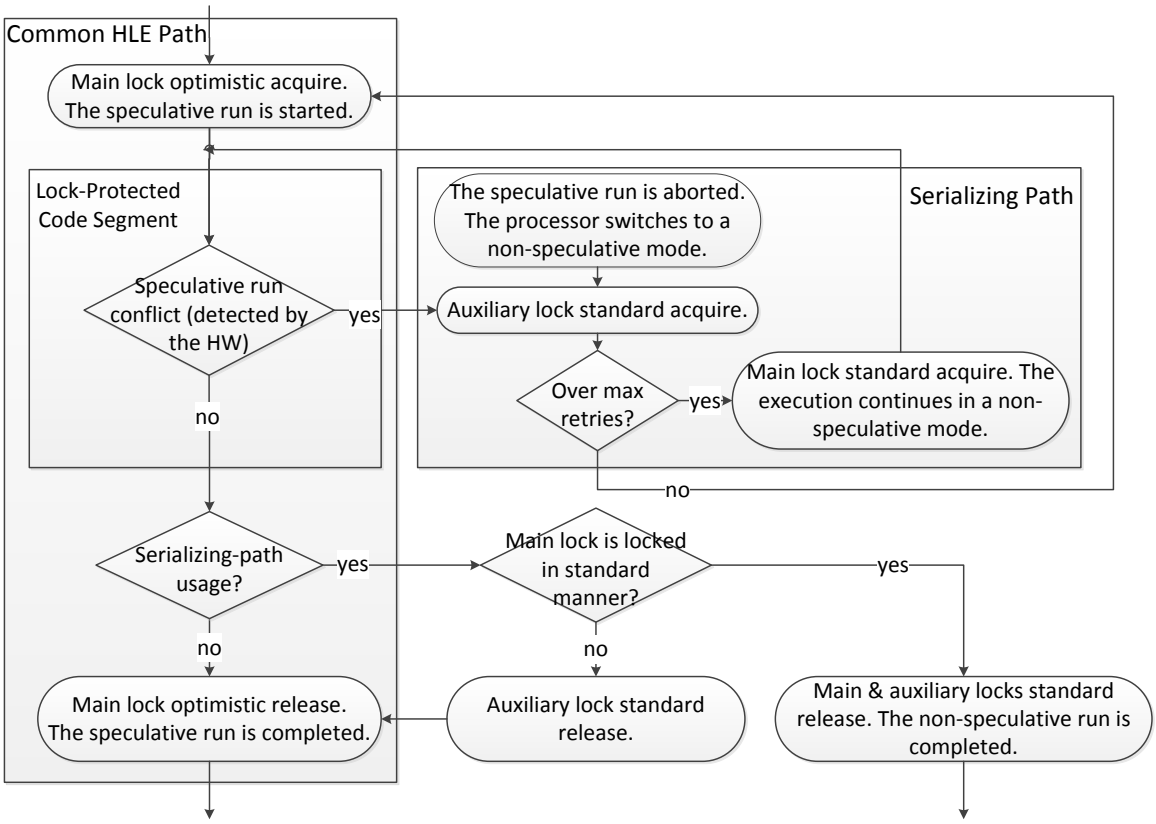


Figure 4.2: The flow-chart of our software-assisted conflict management scheme. The entry point of a speculative segment is the common path. This path enables speculative execution of a lock protected code segment. The serializing path is used only by conflicting threads. The optimistic acquire/release uses the lock elision mechanism.

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 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.

Implementation and HLE compatibility (Algorithm 3) 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 [1], 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.

Algorithm 3 Software-Assisted Conflict Management (SCM)

Require:
main_lock //the main lock
aux_lock //the auxiliary standard lock
thread-local variables:
retries = 0
aux_lock_owner = false

Lock()

```
1: primary_path:
2: XBEGIN (serializing_path)
3: call HLE Lock()
4: return
5: serializing_path:
6: if (aux_lock_owner == true) then
7:    retries++
8: else
9:    aux_lock.lock() //standard lock acquire
10:   aux_lock_owner = true
11: end if
12: if (retries < MAX_RETRIES) then
13:   GOTO primary_path
14: else
15:   main_lock.lock() //standard lock acquire
16: end if
17: return
```

Unlock()

```
1: //XTEST return true if the run is speculative
2: if (XTEST == true) then
3:    call HLE Unlock()
4:    XEND
5: else
6:    main_lock.unlock() //standard lock release
7: end if
8: if (aux_lock_owner == true) then
9:    aux_lock.unlock() //standard lock release
10:   aux_lock_owner = false
11: end if
12: return
```

Lock(): The XBEGIN command (line 2) starts the thread's speculative run (the primary path) and passes the fallback code address (the serializing_path label) as a parameter. In the serializing path (the fallback code for aborted threads), a thread tries to acquire the auxiliary lock in a standard non-speculative manner (lines 9-10) and rejoins the speculative execution (line 13). Only after a

few failed speculative attempts the thread gives up and executes non-speculatively by acquiring the main lock in a standard non-speculative manner (line 15).

Unlock(): If the call is part of a speculative execution (line 2), lines 3-4 complete it (commit the transaction). Otherwise, the main lock is released in a standard non-speculative manner (line 6). In case the auxiliary lock is held by the thread, lines 9-10 release it.

Remark Unfortunately, the initial implementation of TSX in Haswell does not support nesting of HLE within RTM. Therefore, in our experiments we use RTM also to perform 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 3 of the Unlock() function in Algorithm 3, and perform the following at Line 3 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.

Software-assisted lock removal: Rajwar and Goodman [18] 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. However, Haswell's TM has a simple "requestor wins" conflict resolution policy [2] which is prone to livelock [6]. We therefore propose *software-assisted lock removal* (SLR), in which a transactionally executing critical section does not access the lock until it is ready to commit. It then reads the lock and commits if the lock is not held; otherwise, it aborts and retries, giving up and acquiring the lock after a few attempts. A more comprehensive report can be found in [4].

The conflict management scheme applies both to lock elision and lock removal. One can simply replace the HLE Lock()/Unlock() calls in Algorithm 3 (the boxed lines) with appropriate SLR Lock()/Unlock() calls.

Though it benefits mainly HLE, the software-assisted conflict management scheme can be used to further reduce any progress problems caused when SLR threads give up and acquire the lock non-transactionally.

Chapter 5

Evaluation

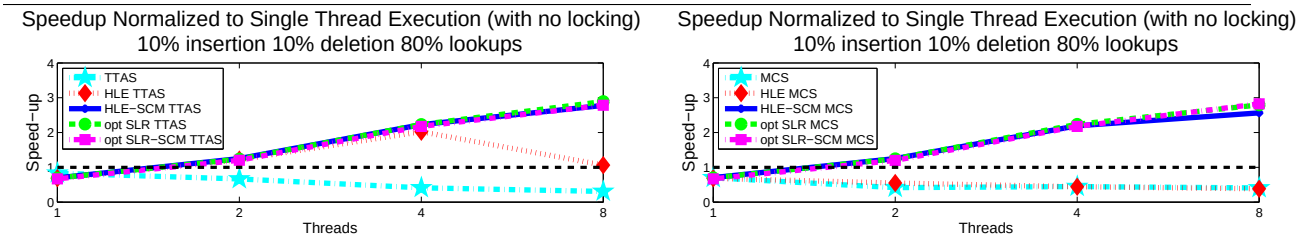
5.1 Overview

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 [10, 20, 15]), 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. PARSEC [5], for example, has been optimized to use fine-grained locks and so applying HLE there is not relevant and would not show any performance improvement.

Hardware setup As in Chapter 3, 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.

Methodology We evaluate our methods on both the MCS lock (as a representative of the class of fair locks) 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 version of the lock with conflict management (*HLE-SCM*), (4) Pessimistic SLR version, in which a thread acquires the lock non-speculatively after one failure (*Pes SLR*), (5) Optimistic SLR version, in which a thread



(a) TTAS lock (b) MCS lock

Figure 5.1: 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.

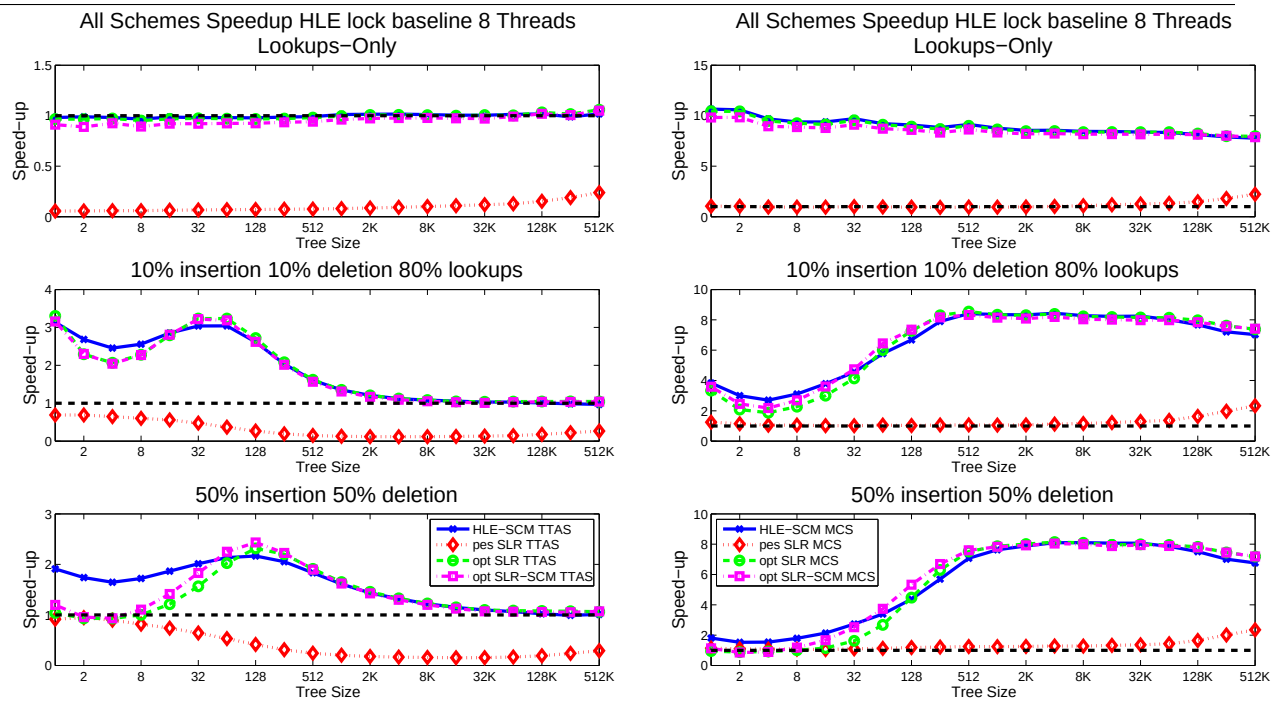
only acquires the lock non-speculatively after retrying speculatively 10 times (*Opt SLR*), and (6) Optimistic SLR version with conflict management applied (*SLR-SCM*).

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.

5.2 Red-black Tree Data Structure Benchmark

We evaluate our methods using two data structure benchmarks, the red-black tree (described in Chapter 3) 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.

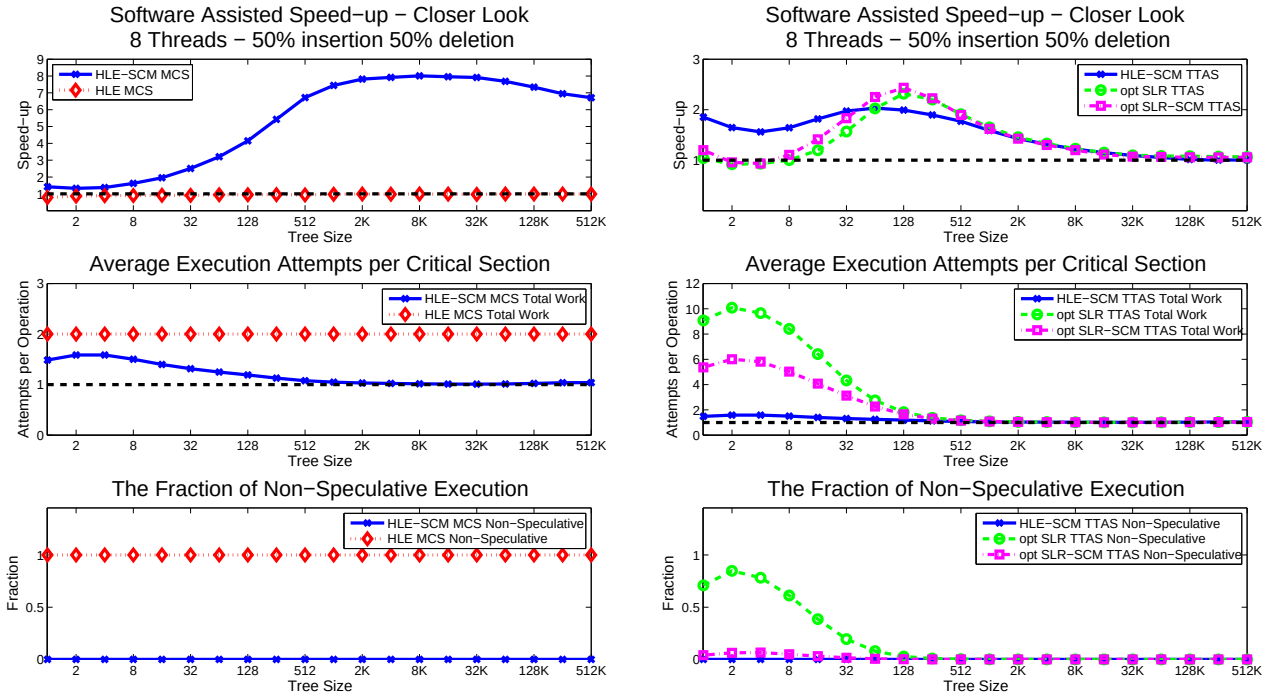


(a) TTAS lock (b) MCS lock

Figure 5.2: 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.

Red-black tree Figure 5.1 shows the *speedup* (relative to the throughput of a single thread with no locking) obtained by the various methods on a 128-node tree under moderate contention (20% updates). It can be seen that using our scheme, the throughput scales with the number of threads. In contrast, with HLE the MCS lock does not scale at all, and even the TTAS does not scale beyond 4 threads. Using our methods eliminates the performance gap between MCS and TTAS.

Figure 5.2 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 (except the pessimistic SLR with TTAS) improved the speedup compared to the plain HLE version of the specific lock (especially on fair locks).



(a) The impact of the software assisted conflict management on high contended HLE based MCS lock

(b) The impact of the different software assisted schemes on high contended HLE based TTAS lock

Figure 5.3: Impact of aborts on executions under different schemes. 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.

TTAS lock On the lookup only (no contention) workload, applying our methods to the TTAS lock shows no performance improvement – the HLE-based TTAS is good enough. 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×. 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. The pessimistic SLR version fails to scale and gives overall poor results.

The HLE-SCM and SLR versions of TTAS give comparable performance in general, except for short transactions. There, HLE-SCM outperforms SLR and SLR-SCM by up to 2×, exactly because of the serialization it induces (see below).

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 severe avalanche behavior due to spurious aborts). We again see comparable results for HLE-SCM and SLR, with a slight advantage to HLE-SCM in short transactions. The pessimistic SLR version gives comparable performance to the plain HLE-based MCS lock, and provides a little speedup in longer transactions.

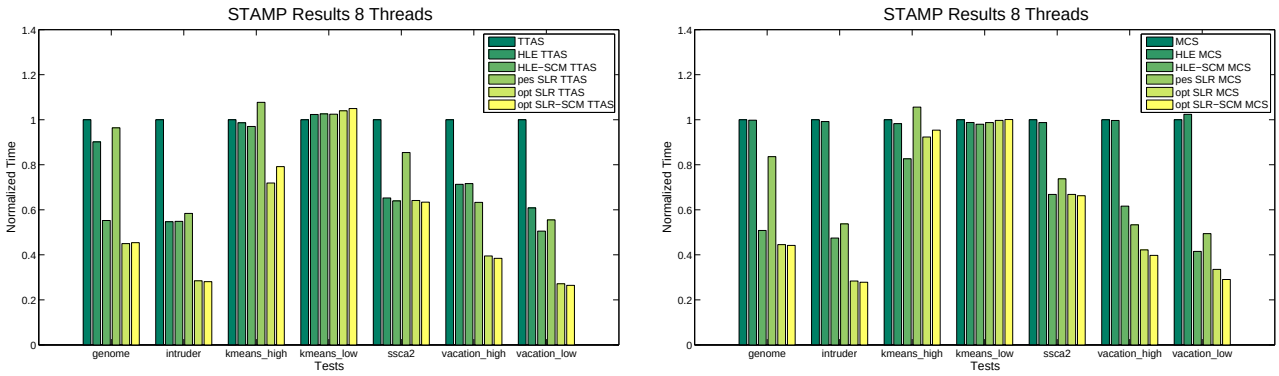
Analysis To gain deeper insight into the behavior of the benchmarks, we run them (see Figure 5.3) with statistics turned on (at the cost of a 5-10% degradation in throughput). Figure 5.3 shows the amount of serialization caused by aborts, as a function of the tree size, for a high level of tree modifications. On the left part, one can see the impact of the SCM scheme on the HLE-based MCS lock. As the conflict level decreases (as the tree size increases), the HLE-SCM requires less attempts in order to complete a single operation (converges to single attempt) and the speedup increases. HLE-SCM manages to complete very high fraction of the operations speculatively. On the right part, one can see the impact of multiple software assisted schemes on HLE-based TTAS lock. The SLR scheme enables (at least partial) speculative execution while the lock is non-speculatively taken. Yet, serializing of conflicting threads to prevent recurrence of known conflicts helps to reduce the number of aborts and eventually to increase the performance. In the highest contention part (small tree sizes) the HLE-SCM performs significantly less attempts per operation, hence gains the better speedup.

5.3 STAMP

To apply our methods to the STAMP suite of benchmark programs [8], we replace the transactions with critical sections that all use the same global lock. Figure 5.4 shows the runtime of the STAMP programs with the various lock elision methods, normalized to the execution time using the plain non-speculative lock.

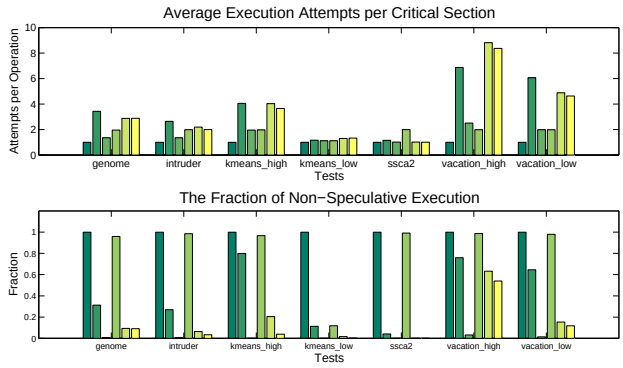
As with the red-black tree data structure benchmark, MCS lock gains no benefit from HLE usage. But, MCS lock provides considerable benefit when used with HLE combined with our conflict management scheme. The HLE-SCM scheme typically improves the performance by up to $2.5\times$.

On the other hand, TTAS lock gains some benefit of HLE usage (up to $2\times$ in *intruder*) but

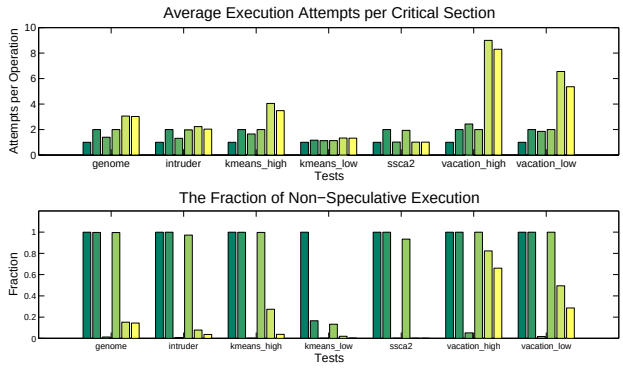


(a) Relative execution time over non-speculative TTAS lock

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



(c) Impact of aborts on executions under TTAS based LE and LR schemes



(d) Impact of aborts on executions under MCS based LE and LR schemes

Figure 5.4: Normalized run time of STAMP applications using standard locking, HLE, and our software-assisted methods.

the HLE-SCM scheme with TTAS gives modest improvement with the exception of `genome` (up to 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.

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. In general, the pessimistic SLR gives modest performance gain but for the most part substantially lower than the other software assisted schemes.

Conclusion: One can see the impact of our SCM scheme on 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 avalanche behavior 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).

Chapter 6

Adjusting Fair Locks to Work with HLE

6.1 Overview

HLE can be applied to a lock implementation only if an unlock of the lock restores the lock to its original state (see Chapter 2). Unfortunately, the techniques used to provide fairness in the two popular ticket lock [14] and CLH lock [13, 9] algorithms do not meet this requirement.

Ticket lock: The ticket lock maintains two counters, *next* and *owner*, which are usually implemented in one memory word. An arriving thread atomically `fetch-and-adds` [11] the *next* counter and then spins, waiting for the *owner* variable to match the value returned by the `fetch-and-add` operation. Upon exiting the critical section the thread releases the lock by advancing the *owner* counter. See Algorithm 4 in Section 6.1 for the pseudo-code.

CLH lock: The CLH queue lock maintains a queue of waiting threads using a linked list of nodes. Each thread has two node pointers, *myNode* and *pred* (its predecessor in the queue). An arriving thread uses `SWAP` [11] on the tail variable to atomically enqueue its own *myNode* pointer and retrieve a *pred* pointer. It then waits to acquire the lock by spinning on the *locked* flag of its predecessor (*pred*) node. To release the lock, the thread sets the *locked* flag in *myNode* to false, and recycles *pred* for use as *myNode* on its next lock acquisition. (See Algorithm 6 in Section 6.1 for the pseudo-code.) Thus, a CLH lock release does not write to the queue's tail and is therefore

not compatible with HLE.

HLE adjustments 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 – which can never happen in HLE – 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. Sections 6.2 and 6.3 contain the full details, and Section 6.4 proves the correctness of these adjustments.

6.2 Ticket Lock Adjustments

The new implementation handles both speculative and standard (non-speculative) runs. We use a compare-and-swap (CAS) primitive [11] 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, either line 1 or 3 are successfully executed. If line 1 is successfully executed, either: (i) the lock is taken in a standard manner and the lock owner is the only running thread (no other requesters) or (ii) the lock is taken in speculative manner and the lock owner (can be one of many) removes all traces of its run. Line 3 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.

6.3 CLH Lock Adjustments

The pseudo-code of the CLH lock implementation, adjusted to the HLE mechanism, is depicted in Algorithm 7. 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

Algorithm 4 Ticket lock

Require:
initialization: next = 0, owner = 0
local variable current

Ticket Lock

```
1: current = F&A(next, 1)
2: while (owner!=current) {busy-wait}
3: {
4:       CS
5: }
```

Ticket Unlock

```
1: F&A(owner, 1)
```

Algorithm 5 Lock elision adjusted ticket lock

Require:
initialization: next = 0, owner = 0
local variable current

Ticket Lock

```
1: current = XACQUIRE F&A(next, 1)
2: while (owner!=current) {busy-wait}
3: {
4:       CS
5: }
```

Ticket Unlock

```
1: ret = XRELEASE CAS(next, current+1, current)
2: if (!ret) then
3:    F&A(owner, 1)
4: end if
```

pred at the tail of the queue, effectively erasing the presence of our node.

6.4 Lock Adjustments Correctness Proofs

Theorem 1. Correctness of the adjusted ticket lock *(i) For speculative runs, the new ticket lock implementation preserves the initial state on release. (ii) The new lock implementation fulfills the mutual exclusion, deadlock free and lockout free properties, similarly to the standard implementation of the lock.*

Algorithm 6 CLH lock

Require:
initialization: tail.locked = false
local variables: myNode, pred

CLH Lock

```
1: myNode.locked = true
2: pred = SWAP(tail, myNode)
3: while (pred.locked) { busy-wait }
4: {
5:       CS
6: }
```

CLH Unlock

```
1: myNode.locked = false
2: myNode = pred
```

Algorithm 7 Lock elision adjusted CLH lock

Require:
initialization: tail.locked = false
local variables: myNode, pred

CLH Lock

```
1: myNode.locked = true
2: pred = XACQUIRE SWAP(tail, myNode)
3: while (pred.locked) { busy-wait }
4: {
5:       CS
6: }
```

CLH Unlock

```
1: ret = XRELEASE CAS(tail, myNode, pred)
2: if (!ret) then
3:    myNode.locked = false
4:    myNode = pred
5: end if
```

Proof. (i) For a speculative run, threads are not aware of each other and every thread behaves as if it runs alone. When a thread calls the unlock() function (Algorithm 5), its *current* value equals the lock's *next* value, since *next* never changes by a speculative thread. Hence, the condition in line 1 of the unlock function is always satisfied and the unlock() ends. (ii) We separate the possible runs into three cases – only standard (non-speculative) threads are run, only speculative threads are run and both speculative and standard threads are run (mixed).

- **Standard run** The lock() is identical to the original one (the XACQUIRE prefix is ignored). During unlock(), either line 1 is successfully executed (solo run) and all traces of the run are removed, or line 3 is successfully executed, which is equivalent to the standard implementation.
  - Mutual Exclusion – Assume the CS is empty and there are one or more requesters. lock():1 creates order between the requesters (each one has a unique *current*). Hence, exactly one of them has the current value of *owner* and it is the only one that can enter the CS. While the winner is in the CS, all other requesters are busy waiting (lock():2) and cannot enter the CS. When the winner releases the lock, either unlock():1 is successfully executed and no other requester is waiting, or unlock():3 is successfully executed and *owner* is incremented by one. The unique requester whose *current* equals the new *owner* value can then enter the CS.
  - Deadlock Free – lock():1 is always successfully executed, and an order is created between the requesters. Each requester is aware of its turn and no extra synchronization is needed between the requesters to guarantee progress.
  - Lockout Free – lock():1 creates order between the requesters. Given a thread with a *current* value, exactly *current* minus *owner* requesters will enter the CS before it does.
- **Speculative and mixed runs** When the first case is proven, the correctness of the other two is trivially achieved by the correctness of the HLE mechanism itself.

□

Theorem 2. Correctness of the CLH lock *(i) For speculative runs, the new CLH lock implementation preserves the initial state on release. (ii) The new lock implementation fulfills the mutual exclusion, deadlock free and lockout free properties, similarly to the standard implementation of the lock.*

Proof. (i) For a speculative run, threads are not aware of each other and every thread behaves as

if it runs alone. When a thread calls unlock() function (Algorithm 7), its own node ($myNode$) is pointed out by the lock's *tail* even though *tail* is never changed by a speculative thread. Hence, the condition in line 2 of the unlock function is always satisfied, and the unlock() ends. (ii) We separate the possible runs into three cases – only standard non-speculative threads are run, only speculative threads are run and both speculative and standard threads are run (mixed).

- **Standard run** The lock() is identical to the original one (the XACQUIRE prefix is ignored). During unlock(), either line 1 is successfully executed (solo run) and all traces of the run are removed, or lines 3,4 are successfully executed, which is equivalent to the standard implementation.
  - Mutual Exclusion – Assume the CS is empty and there are one or more requesters. The CLH lock holds some *tail* node with clear *locked* flag (the CS is empty). lock():2 creates order between the requesters. Every requester has a unique *pred*, which is a specific position in the queue. Hence, exactly one requester has a clear *pred.locked* flag and it is the only one that can enter the CS. While the winner is in the CS, all other requesters are busy waiting (lock():3) and cannot enter the CS. When the winner releases the lock, either unlock():1 is successfully executed and no other requester is waiting, or unlock():3 is successfully executed and *locked* flag is cleared. The unique requester whose *pred* equals the last winner's node can then enter the CS.
  - Deadlock Free – lock():2 is always successfully executed, and an order is created between the requesters. Each requester is aware of its turn and no extra synchronization is needed between the requesters to guarantee progress.
  - Lockout Free – lock():2 creates order between the requesters. Given a thread position in the queue, new requesters cannot precede.
- **Speculative and mixed runs** When the first case is proven, the correctness of the other two is trivially achieved by the correctness of the HLE mechanism itself.

□

Chapter 7

Extending Haswell's HLE Implementation

Here we present an alternative hardware-based solution (no software-assistance is needed), to cope with the avalanche phenomenon described in Chapter 3. We suggest extending HLE's conflict detection to distinguish between conflicts on the lock and conflicts on the data cache lines, allowing speculative threads to make progress even when encountering a held lock. In turn, this allows speculative sections that do not conflict on data lines to continue with their speculative runs while the conflicting ones do serialize. Our proposal does not require cache-coherence protocol changes.

Our proposal is based on the following observation. In contrast to a conflict on the *data* in the critical section, which means the conflicting sections need to be serialized, a conflict on the lock's cache line is merely a *synchronization signal*. It indicates that some thread T has acquired the lock, but T does not necessarily conflict with all running speculative threads. Yet, as Lemma 1 below shows, allowing a speculative thread to ignore a conflict on the lock and continue running concurrently with T (until either it experiences a data conflict with T or commits) can lead to incorrect executions. That is why the Haswell processor implements a *conservative* approach that guarantees correctness, terminating all speculative threads when the lock is non-speculatively acquired.

Lemma 1. *Concurrent execution of both speculative and non-speculative threads can lead to inconsistent state.*

Proof. When different critical-section segments are protected by the same lock, any two speculative threads that run any of these segments see each other as a transaction. However, when a thread T holds the lock and runs non-speculatively, this does not hold. Memory updates performed by such a thread are made globally visible one at a time, making it possible for a concurrent speculative thread to observe T's individual writes, hence to observe inconsistent state (as depicted in the following *erroneous* example).

Example: Consider the case of two code segments protected by the same lock L:

C_1:	C_2:
`lock(L)`	`lock(L)`
`load(X)`	`store(Y)`
`load(Y)`	`store(X)`
`unlock(L)`	`unlock(L)`

Suppose now that thread T_1 transactionally executes C_1 without accessing L and reads $X = 0$ from memory. Now another thread, T_2, executes C_2 non-transactionally. It therefore acquires L and then stores 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 $Y = 1$. □

Our proposal We define a *lock state change* as any cache-coherence state change of the lock's cache-line, and *data access conflict* as any other conflict. Note that the processor can identify the lock cache line since it is written to by the instruction prefixed with XACQUIRE.

Define group A as a group of threads that are part of a speculative run, all of them using the same lock L. Assume that during this speculative run, a non-speculative thread T takes L in a standard non-speculative manner. Instead of automatically aborting group A, every speculative thread tries to complete its speculative execution. Upon receiving the cache eviction event for L, the processor does not abort the thread. Instead it enters a special state S, in which it behaves as follows: as long as the speculative thread accesses only data which is already in its caches, it can safely continue. If the speculative execution encounters a cache miss (due to a read or a write operation), it reads the lock address again. If the lock is free (i.e., contains the same value as before the XACQUIRE), the speculative execution can continue, otherwise, it is suspended. Hence, while

the lock is taken, speculative threads can proceed as long as they do not incur a cache miss in order to expand their read or write sets. When the lock is released, the resulting cache coherency operation releases the suspended speculative threads which continue their speculative execution by redoing the memory operation that caused the cache miss.

The lock cache-line is placed in the read/write-set only if it gets accessed for data. If the lock's cache-line gets evicted, either (i) it's in the read/write-set, and we abort, or (ii) it's not (only the lock got accessed so far), so we proceed with the described protocol.

Correctness Intel TSX gives full isolation of transactional code from the outside environment. Every transaction, including a live (not yet completed) one, accesses only a consistent state (a state produced by a sequence of previously committed transactions), which ensures that every return value of an operation executed by a transaction is consistent with the return values of all previous operations of the very same transaction. On the other hand, every transaction appears as if it was a black-box operation – only the end result of the transaction is counted.

Our extension of the HLE's conflict detection preserves these characteristics and guarantees that no lock-elided read is done in parallel with lock protected write. Given a speculative thread that successfully completes its run despite some conflicts that occurred during its speculative execution, we claim the following:

- No real conflict has occurred during the speculative run of the thread since in case of data access conflict (other than the lock's cache-line), the speculative run is aborted.
- On any cache-coherence state change of the lock's cache-line
  - If the speculative execution can be completed using local registers and available cache-lines without any further reads or writes (local run), we can serialize the execution as if it has completed before the lock acquisition, hence it is safe to complete it.
  - If the lock is taken and the speculative execution performs a non-cached read address or write-to-memory operation, the speculative execution is suspended by the hardware. Hence there are no speculative read/write memory operations done in parallel with non-speculative execution.
  - Data access conflicts are determined as defined in the basic HLE's conflict detection. Hence, the non-speculative cache-line requester continues its execution (requester wins) while other speculative owners of the cache-line are terminated by the hardware.

– When the lock is released, causing another cache-coherence state change of the lock's cache-line to occur, the speculative execution can continue. There was no real conflict between the speculative and standard non-speculative threads (no conflict other than the lock's cache-line has occurred). In that case we can refer to the standard execution as if it has been completed before the speculative one. The speculative execution can continue its run without the need to verify the validity of the read and write sets since the hardware does it continuously.

The benefit of our proposal We argue here that our proposal can only improve concurrency and performance. One of the main differences between TM and HLE is that when HLE is used, an abort serializes every transactional execution protected by the same lock. The assumption that one data conflict between two threads must lead to data conflict among all the other speculative threads is the most pessimistic behavior. On STM-based solutions, handling zombie threads (threads that eventually must abort but may continue to run even after it has become impossible for them to commit) can cost more than just abort them on the first conflict. This is not the case with HTM-based solutions, where the constant validation of the read/write sets is done completely by the hardware with no cost in terms of computation power spent by the software.

In standard HLE, a non-speculative thread that acquires the lock causes every speculative thread to abort and wait for the lock to be released. However, in our proposal speculative threads use this time to continue with their original execution. Only when they access memory (to expand their read or write sets) do they wait for the lock to be released. In the worst case, a speculative thread might abort at this point (no worse than the original scenario), but in the best case, it might continue executing (i.e., if there is no data conflict with the lock holder). Thus our proposal turns time wasted waiting into time spent working.

Chapter 8

Related work

Rajwar and Goodman [17] introduced the concept of speculative lock elision (SLE) to automatically replace locking with speculative hardware transactions which allows parallel execution of non-conflicting critical sections. The speculative hardware automatically elides the delimiting synchronization operations of a critical section and executes it speculatively, buffering any updates. If the critical section completes with no data conflicts, the updates are committed, and otherwise they are discarded, and the critical section is rerun non-speculatively by acquiring the lock. As in Haswell's HLE, transactional aborts are handled by acquiring the lock and serializing all the threads. However, performing conflict resolution in software allows more flexibility. In our software-assisted conflict management scheme, conflicts are handled in a way that *minimizes the need to switch between speculative and non-speculative executions* providing great performance boost.

Rajwar and Goodman subsequently proposed transactional lock removal [18] With TLR, the lock is never acquired and released but only serves to denote the beginning and end of a transaction. To prevent livelocks in the case of conflicts, the scheme uses hardware-based conflict management and serializing conflicting transactions. Our approach achieves a similar goal, but does so using software techniques without further hardware changes.

Dice et al. [10] studied transactional lock elision (TLE) using Sun's Rock processor which also supports hardware TM. Even though theirs lock elision algorithm uses backoffs, they still point out the *lemming effect* which is similar to the *avalanche behavior* and sketch a non-backoff software mechanism to speedup *recovery* from it. In contrast, our conflict management scheme prevents the problem in the first place and manages to prevent the continuous zigzag between speculative and

standard executions altogether. The scheme serializes conflicting threads (to prevent recurrence of known conflicts), enables them to keep participating in the speculative execution but does it without acquiring the lock to avoid impact on the other threads in the system.

Implementing elision-friendly locks using Intel's Haswell processor is discussed in [2]. 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.

Bahar et al. [15] analyze both lock elision and lock removal schemes in the context of embedded systems with hardware TM. They discuss hardware approaches, whereas we are interested in what can be done using software to assist the TM.

Roy, Hand, and Harris describe how to perform lock elision using software instrumentation and no hardware changes [19]. However, they assume a system without hardware TM, whereas we use software techniques to assist the hardware. But now days, when HTM is on the verge of becoming a mainstream mass-market feature, the excessive cost of handling TM entirely in software is unlikely.

Bibliography

[1] Intel Architecture Instruction Set Extensions Programming Reference. `http://software.intel.com/file/41604`, 2012.

[2] Intel 64 and IA-32 Architectures Optimization Reference Manual. `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, July 2013.

[3] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with hardware lock elision. In *PPOPP*, pages 295–296, 2013.

[4] Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved hardware lock elision. Technical report, Tel-Aviv University, 2013.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 81–91, New York, NY, USA, 2007. ACM.

[7] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *ISCA 2013*, pages 225–236, New York, NY, USA, 2013. ACM.

[8] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE*

International Symposium on Workload Characterization, pages 35–46, Washington, DC, USA, September 2008. IEEE Computer Society.

[9] Travis 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.

[10] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems, 2009.

[11] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991.

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[13] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Washington, DC, USA, 1994. IEEE Computer Society.

[14] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[15] Dimitra Papagiannopoulou, Giuseppe Capodanno, R. Iris Bahar, Tali Moreshet, Aditya Holla, and Maurice Herlihy. Energy-efficient and high-performance lock speculation hardware for embedded multicore systems. In *The 8th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'13)*, 2013.

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

[17] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Sym-*

posium on Microarchitecture, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '02, pages 5–17, New York, NY, USA, 2002. ACM.

[19] Amitabha Roy, Steven Hand, and Tim Harris. 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.

[20] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.