Using Hardware Transactional Memory to Correct and Simplify a Readers-Writer Lock Algorithm

Dave Dice Yossi Lev
Victor Luchangco Mark Moir
Oracle Labs
{dave.dice,yossi.lev,victor.luchangco,mark.moir}@oracle.com

Yujie Liu
Lehigh University
lyj@lehigh.edu

Abstract

Designing correct synchronization algorithms is notoriously difficult, as evidenced by a bug we have identified that has apparently gone unnoticed in a well-known synchronization algorithm for nearly two decades. We use hardware transactional memory (HTM) to construct a corrected version of the algorithm. This version is significantly simpler than the original and furthermore improves on it by eliminating usage constraints and reducing space requirements. Performance of the HTM-based algorithm is competitive with the original in "normal" conditions, but it does suffer somewhat under heavy contention. We successfully apply some optimizations to help close this gap, but we also find that they are incompatible with known techniques for improving progress properties. We discuss ways in which future HTM implementations may address these issues. Finally, although our focus is on how effectively HTM can correct and simplify the algorithm, we also suggest bug fixes and workarounds that do not depend on HTM.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords readers-writer lock, hardware transactional memory

1. Introduction

We recently observed incorrect behavior in a test using the readers-writer lock (RW-lock) algorithm of Krieger et al. [8] (which we call `KSUH`). This led us to discover a subtle bug in `KSUH` that has apparently gone unnoticed for almost two decades. The bug involves synchronization for removing a node from a doubly-linked list. It turns out that a node can be modified by a late update after it has already been reused; this bug can manifest in a variety of ways.

It was straightforward to devise an HTM-based variant of the `KSUH` algorithm that is significantly simpler than the original, eliminates the bug, and eliminates inconvenient (and implicit) requirements for client code. For evaluation, we used a system based on the prototype Rock chip developed by Sun Microsystems [2, 13], which supports HTM. This evaluation shows that our HTM-based algorithms perform competitively with (a corrected version of) `KSUH` under reasonable workloads.

While this is encouraging, there are two issues with this simple algorithm. First, "best effort" HTM such as Rock supports does not guarantee to be able to commit transactions, so threads can starve, even running alone. Second, one variable of the algorithm (the tail of the list) is frequently modified in transactions, raising concerns about performance under contention (even in read-dominated workloads). Although the RW-lock might arguably have been designed differently in the first place to avoid contention on a single variable, we decided to retain the basic structure and approach of the `KSUH` algorithm in order to evaluate the effectiveness of HTM to improve and simplify an algorithm even if the use of HTM has not been anticipated in the original design.

The first issue can be addressed by using Transactional Lock Elision (TLE) [1, 11], whereby a thread that repeatedly fails to commit a transaction acquires a lock, and then executes the code of its transaction nontransactionally. To preserve correctness, all transactions are modified so that they cannot commit when the lock is held. It was straightforward to apply TLE to our transactional RW-lock, and our evaluation confirmed that the overhead was low.

To address the second issue, we have explored a variety of techniques for transforming the algorithm to reduce contention. Many of these transformations preserve the semantics of the code, and thus could potentially be applied by compilers. Others depend on knowledge of the algorithm, but still could be used in a fairly systematic way by programmers.

Unfortunately, some of the optimizations we have explored to reduce contention are not compatible with the simple TLE technique for ensuring progress. While we were able to use similar, but algorithm-specific, techniques to accommodate at least one such optimization, doing so in general quickly complicates the algorithm, thereby at least partially defeating the purpose of using HTM in the first place. We discuss ways in which future HTM implementations could reduce or eliminate these drawbacks.

Our main focus is on the ability of HTM to simplify synchronization algorithms. However, we also discuss ways to address the bug we identified with the `KSUH` algorithm without using HTM; these include solutions that do not modify the RW-lock implementation, but impact how clients use it, and vice versa.

In Section 2, we describe the `KSUH` algorithm, explain how the bug in it arises, and describe the changes we made to eliminate it. In Section 3, we describe our basic transactional variant of `KSUH`, as well as several optimizations aimed at reducing contention on the `Tail` variable. In Section 4, we contrast the difficulty of reasoning about the correctness of `KSUH` and of our basic transactional variant, highlighting how HTM makes algorithms simpler. We present performance results in Section 5, discuss alternative solutions to the bug we identified in Section 6, and conclude in Section 7.

2. The KSUH Algorithm

The KSUH RW-lock algorithm is derived from Mellor-Crummey and Scott's well-known mutual exclusion algorithm (MCS) [10]. Before discussing details, it is useful to describe abstract versions of these algorithms, both to convey intuition about the algorithms, and to provide useful reference points for correctness proofs, as discussed further in Section 4.

2.1 Abstract algorithm descriptions

We begin with an abstract description of MCS, which maintains an ordered list of client-provided nodes—one for each thread that is in the critical section or is waiting to enter. When invoking a lock procedure, a thread must pass a node that is not already being used by any thread (including the invoking thread), and when invoking an unlock procedure, it must pass the same node that it passed to the corresponding lock procedure. We say that the thread *owns* this node from the time it invokes the lock procedure until it returns from the corresponding unlock procedure. If a thread inserts its node into an empty list, it enters the critical section; otherwise it waits until its node is first in the list. When leaving the critical section, a thread removes its node from the list. Thus, a thread in the critical section always owns the first node in the list.

The abstract description for KSUH is similar, but supports acquiring the lock in read or write mode, and allows a reader to enter the critical section provided all of the nodes ahead of its node in the list belong to readers. Because readers might not exit the critical section in the same order they entered, the algorithm allows the removal of reader nodes from within the list (in MCS, only the first node is ever removed).

2.2 Details of KSUH

Figure 1 shows KSUH (with our fix marked by ***); we discuss important details below. As in MCS, the Tail variable (called L in [8, 10]) identifies the most recently added node, and if the next field of a node n is non-NULL, it identifies the node that was added after n. To enable departing readers to remove their nodes from within the list, each reader node is linked to its predecessor (if any) in the list via a new prev field, and a per-node lock is used for node removal, as described below; henceforth we refer to these per-node locks as *mutexes* to avoid confusion with the RW-lock being implemented. KSUH also adds a state field to each node, which contains READER or WRITER when the node is added to the list, indicating the mode of the lock request; a reader changes its state to ACTIVE_READER before entering the critical section (line 29) to allow its successor to follow it into the critical section. As in MCS, the waiting described in the abstract algorithm is achieved by a thread spinning on a spin field in its node; another thread informs it that it can enter the critical section by resetting this field.

A node is inserted into the list by using SWAP to record the previous value of Tail while simultaneously storing a pointer to the new node into Tail (lines 5 and 21). This establishes the order in which nodes are added, but additional steps are needed to link added nodes into the list (lines 7, 23, and 24). Because these steps may be delayed, sometimes a thread must wait until such linking has been performed (lines 13, 44, and 53).

We approximate the property that a reader can enter the critical section if all the nodes before its node in the list belong to readers by propagating the information that the first reader in the list has entered the critical section down the list. Before entering, a reader r releases its successor if it is a reader (line 28), and then sets its state to ACTIVE_READER (line 29), so that a subsequently arriving reader can determine that it does not need to wait (line 25). Such a successor may arrive too late for r to release it, and start spinning before r sets its own state to ACTIVE_READER. To ensure progress

```
 1  procedure writerLock( Tail , I)
 2    I→state = WRITER;
 3    I→spin = 1;
 4    I→next = NULL;
 5    pred = SWAP(Tail, I );
 6    if (pred != NULL)
 7      pred→next = I;
 8      while (I→spin);

10  procedure writerUnlock( Tail , I)
11    if (! I→next && CAS(Tail, I, NULL))
12      return;
13    while (! I→next);
14    I→next→prev = NULL;
15    I→next→spin = 0;

17  procedure readerLock( Tail , I)
18    I→state = READER;
19    I→spin = 1;
20    I→next = I→prev = NULL;
21    pred = SWAP(Tail, I );
22    if (pred != NULL)
23      I→prev = pred;
24      pred→next = I;
25      if (pred→state != ACTIVE_READER)
26        while (I→spin);
27    if (I→next && I→next→state == READER)
28      I→next→spin = 0;
29    I→state = ACTIVE_READER;

31  procedure readerUnlock( Tail , I)
32    pred = I→prev;
33    if (pred)
34      LOCK(pred);
35      while (pred != I→prev)
36        UNLOCK(pred);
37        pred = I→prev;
38        if (! pred) break;
39        LOCK(pred);
40      if (pred)
41        LOCK(I);
42        pred→next = NULL;
43        if (! I→next && !CAS(Tail, I, I→prev))
44          while (! I→next);
45        if (I→next)
46          I→next→prev = I→prev;
47          I→prev→next = I→next;
48        UNLOCK(I);
49        UNLOCK(pred);
50        return;
51    LOCK(I);
52    if (! I→next && !CAS(Tail, I, NULL))
53      while (! I→next);
54    if (I→next)
55      succIsWriter = (I→next→state == WRITER);    ***
56      I→next→spin = 0;
57      if (! succIsWriter )                         ***
58        I→next→prev = NULL;                        // see caption
59    UNLOCK(I);
```

Figure 1. KSUH algorithm with fix (marked by ***). In [8], line 58 reads "I→prev→prev = NULL". We believe this is a typo, as it does not make sense and the algorithm is easily seen to be incorrect with this version. The corrected code usually makes sense, but does not avoid the bug we have discovered; to address this bug, we introduced the two lines marked with ***.

in this case, the successor is eventually released by the owner of its predecessor exiting the critical section (line 56).

When a thread t removes a reader node r that has a predecessor node p and a successor node s, t acquires the mutexes of p and r, and then updates links in p and s in order to splice r out of the list (lines 32–50). If r has a successor s but no predecessor, t acquires the mutex only for r (line 51). After removing r from the list in this case, t releases the owner of s by setting its spin field to 0 (line 56), and then clears s's prev field (line 58). This is important if s is a reader node, because the subsequent removal of s should not access the removed node r. Because r's mutex is held while s's prev field is reset, s cannot be removed before this occurs.

If s is a writer node, however, there is a problem. Because r's owner has cleared s's spin field, the owner of s can enter and exit the critical section, and then call writerUnlock. This procedure does not check s's predecessor in the list, so the owner of s may return from writerUnlock even though r's mutex is still held. Now there is a pending write to the prev field of s, which can occur at any time, including after s has been recycled. This is the root of the problem that manifests in at least four different ways, depending on when the late store happens: a read-write exclusion violation, a write-write exclusion violation, a segmentation fault, and an infinite loop. For example, if s is reused as a reader node, the late store to reset its prev field can break the list, so that a departing reader can believe it is the last one, and therefore release a writer into the critical section, causing a read-write exclusion violation.

We initially thought that the bug could be fixed by reversing the order of the stores performed in lines 56 and 58. However, this does not work: it allows a successor reader to believe it is at the head of the list, so it can remove itself without coordinating with its predecessor, which may be poised to set the successor's spin flag, even after the successor's node is recycled.

We explored several fixes for KSUH, settling on the one indicated by *** in Figure 1, whereby we reset the successor's prev field (line 58) only if it is not a writer. To enable this, we record whether the successor is a writer (line 55) before releasing its owner into the critical section (line 56); this ensures that the node has not been recycled before recording whether the successor node is a writer. (Recall that, if the successor node is a reader, the mutex protocol prevents it from being recycled prematurely.)

Finally, we note a critical issue regarding how clients manage the nodes they use with the KSUH algorithm. (We assume an environment without garbage collection.) MCS allows nodes to be stack allocated, so that explicit recycling of nodes is unnecessary. This is not possible with KSUH because nodes can be modified arbitrarily long *after* they have been removed from the list; if the memory used for a node has been reallocated for another purpose, such late updates can lead to arbitrary behavior. Thus, KSUH requires clients to store nodes in Type Stable Memory (TSM) [7], i.e., ensure that they are never deallocated or reused for a different purpose.

The corrected KSUH must still keep nodes in TSM because readerUnlock (line 34) attempts to acquire the mutex of a node it previously identified as the predecessor of the node to be removed. There is no guarantee that this node is not removed from the list before the mutex is acquired. The transactional RW-lock algorithms presented next eliminate this issue, so that—as with MCS—nodes can be stack allocated, a significant advantage for developers.

3. Transactional RW-lock algorithms

In this section, we first present a basic transactional RW-lock algorithm TxLock, which retains the structure of KSUH, but uses transactions to simplify and improve the algorithm. Then we introduce a variant that seeks to reduce contention on Tail, while retaining much of the simplicity of TxLock. Finally, we discuss a number of optimizations and variations on these algorithms.

3.1 Basic transactional algorithm: TxLock

Our first transactional RW-lock algorithm, TxLock, shown in Figure 2, retains KSUH's overall approach. However, using transactions to update the list (and to record state needed to determine actions to be taken after a transaction commits), simplifies the algorithm and eliminates the need for the per-node locks used by KSUH. Shared variables are accessed only within transactions, with the following exceptions. Initialization of a node's fields before it is made "public" by inserting it into the list are performed without synchronization. Accesses to spin variables, both by threads waiting for these variables to change, and by threads changing them, are performed nontransactionally. This is for two reasons.

First, if a thread inserts a node and spins on its spin field within the same transaction, it will never exit the spin, because the node will not become visible so no thread will reset its spin field. We could perform the spinning in a separate transaction after the transaction that inserts the node into the list has committed. However, a waiting transaction would be caused to abort by the event for which it is waiting, so it would have to retry, harming performance. Furthermore, it does not complicate the algorithm significantly to perform the spinning nontransactionally.

Second, waiting threads are also released using nontransactional stores, because a transaction attempting to modify a variable on which another thread is spinning may have its transaction aborted due to the spinning thread. Again, updating spin variables nontransactionally did not significantly complicate the algorithm.

The writerLock and writerUnlock procedures are fairly self-explanatory. The first transaction in readerLock (lines 26–32) inserts a new node into the list and also records (in predState) the state of the node previously pointed to by Tail, if any. If that node exists and its status is not ACTIVE_READER, the thread then spins, waiting for its predecessor to release it. After this, it uses another transaction (lines 36–38) to atomically change its state to ACTIVE_READER and to record the next node in the list, if any. If there is a next node and it is a reader, the thread releases the thread that owns the next node (line 40).

The readerUnlock procedure uses a transaction (lines 44–54) to remove the departing thread's node from the list. If the removed node has no predecessor, the transaction records the state of the next node, and if it is a writer, its thread is released into the critical section (line 55) as the departing thread is the last reader before it.

Our use of transactions makes this algorithm simpler and easier to reason about than KSUH in several ways. First, the locks used in KSUH are no longer needed, which both saves space and eliminates the need to reason about the locks.

Next, note that TxLock resets the successor's prev field without regard to whether it is a reader or a writer (line 49). This may be surprising, as it was exactly this behavior that caused the bug in KSUH. This does not compromise TxLock's correctness or require it to use TSM because the write to the prev field is in a transaction that confirms that the node being written is still the successor of the writing thread's node, which itself is still in the list.

There is no analogue in TxLock for the first loop in the readerUnlock procedure of KSUH, which is needed because the list may change after a thread determines its node's predecessor and before it is able to lock the predecessor. If this occurs in TxLock, the transaction will abort and be retried, and this is hidden by the transaction construct. This is the reason that our algorithm allows nodes to be freed after use.

In both readerLock and writerLock, the next field of the node previously pointed to by Tail is set to point to the newly introduced node, atomically with setting Tail to point to this node. Therefore, there is no "intermediate" state in which Tail already points to a new node, but the previous node's next field has not yet been set to point to the new node. (KSUH has two loops to deal with this case.)

Finally, our use of transactions ensures that, when a reader r sets its status to ACTIVE_READER before entering the critical section, either r will see a successor reader s and thus reset its spin field so s can also enter the critical section, or s will see r's status as ACTIVE_READER when it links in its node, and thus can enter the critical section. As a result, s never has to wait while r is in the critical section. In contrast, this can happen with KSUH, as

```
procedure writerLock( Tail , I )
  I→state = WRITER;
  I→spin = 1;
  I→next = NULL;
  atomically
    pred = Tail ;
    Tail = I ;
    if (pred) pred→next = I;

  if (pred) while (I→spin) Pause();

procedure writerUnlock( Tail , I )
  atomically
    next = I→next;
    if (! next)

      Tail = NULL;
    else
      next→prev = NULL;
  if (next) next→spin = 0;

procedure readerLock( Tail , I )
  I→state = READER;
  I→spin = 1;
  I→next = I→prev = NULL;
  atomically
    pred = Tail ;
    Tail = I ;
    if (pred)
      I→prev = pred;
      pred→next = I;
      predState = pred→state;

  if (pred != NULL && predState != ACTIVE_READER)
    while (I→spin) Pause();
  atomically
    I→state = ACTIVE_READER;
    next = I→next;
  if (next && next→state == READER)
    next→spin = 0;

procedure readerUnlock( Tail , I )
  succState = UNDEF_STATE;
  atomically
    pred = I→prev;
    next = I→next;
    if (pred) pred→next = next;
    if (next)
      next→prev = pred;
      if (! pred)
        succState = next→state;
    else

      Tail = pred;
  if ( succState == WRITER) next→spin = 0;
```

Figure 2. The basic transactional RW-lock algorithm `TxLock`.

```
procedure writerLock( Tail , I )
  I→state = WRITER;
  I→spin = 1;
  I→next = NULL;
  while ( Tail == POISON || (pred = SWAP(Tail, POISON)) == POISON)
    Pause ();

  if (pred) pred→next = I;
  Tail = I ;
  if (pred) while (I→spin) Pause();

procedure writerUnlock( Tail , I )
  atomically
    next = I→next;
    if (! next)
      if ( Tail == POISON) retry;
      Tail = NULL;
    else
      next→prev = NULL;
  if (next) next→spin = 0;

procedure readerLock( Tail , I )
  I→state = READER;
  I→spin = 1;
  I→next = I→prev = NULL;
  while ( Tail == POISON || (pred = SWAP(Tail, POISON)) == POISON)
    Pause ();
  I→prev = pred;
  if (pred)
    atomically
      pred→next = I;
      predState = pred→state;
  Tail = I ;
  if (pred !=NULL && predState != ACTIVE_READER)
    while (I→spin) Pause();
  atomically
    I→state = ACTIVE_READER;
    next = I→next;
  if (next && next→state == READER)
    next→spin = 0;

procedure readerUnlock( Tail , I )
  succState = UNDEF_STATE;
  atomically
    pred = I→prev;
    next = I→next;
    if (pred) pred→next = next;
    if (next)
      next→prev = pred;
      if (! pred)
        succState = next→state;
    else
      if ( Tail == POISON) retry;
      Tail = pred;
  if ( succState == WRITER) next→spin = 0;
```

Figure 3. Final `PoisonTheTail` algorithm.

explained in Section 2.2. For the same reason, in our algorithm, `readerUnlock` resets the spin of the successor only if it is a writer, because the above-described scenario in which reader s may wait for reader r to release it even though r has already entered the critical section cannot happen.

While `TxLock` (Figure 2) is simpler than `KSUH` and does not require nodes to be stored in TSM, it performs considerably worse than (corrected versions of) `KSUH` under heavy contention. We have explored a number of ways to improve its performance. In the remainder of this section, we discuss algorithmic changes we have explored, in which we sacrifice some of the simplicity of `TxLock` in order to improve performance.

3.2 Poison the Tail

In `TxLock`, `Tail` is modified by every lock operation and some unlock operations. Heavy contention on a single shared variable can lead to livelock with simple "requester wins" conflict resolution mechanisms, such as Rock uses [2, 3]. We were therefore motivated to try to reduce contention on `Tail`, while continuing to exploit HTM to keep the algorithm simpler than the `KSUH` algorithm.

To avoid retries in cases that always modify `Tail` (`writerLock` and `readerLock`), we changed these procedures to use nontransactional atomic instructions (such as SWAP and CAS) to modify `Tail`. In other cases, we sought to make transactions access `Tail` less frequently and to be vulnerable to conflicts on this variable for shorter periods of time, and in some cases to access `Tail` without using a transaction. (This approach assumes that the HTM supports strong atomicity, allowing `Tail` to be concurrently accessed both by CAS and within transactions; Rock supports this behavior.) This effort led us to the `PoisonTheTail` algorithm, which we describe via a series of modifications to `TxLock`, using the `writerLock` procedure as an example. The final algorithm appears in Figure 3.

We first change `writerLock` and `readerLock`, which always modify `Tail`, to "claim" the right to insert the next node by modifying `Tail` with a nontransactional atomic instruction, and then complete the insertion using a transaction. To limit the complexity introduced by separating these steps, we prevent any transaction that accesses `Tail` from completing between them. To do so, the "claiming" is achieved by changing `Tail` from a node pointer to a special value `POISON`, recording the replaced pointer to enable the subsequent transaction to link the new node into the list. The resulting `writerLock` procedure is:

```
procedure writerLock( Tail , I )
  I→state = WRITER;
  I→spin = 1;
  I→next = NULL;
  while ( Tail == POISON || (pred = SWAP(Tail, POISON) == POISON))
    Pause ();
  atomically
    Tail = I ;
    if (pred)
      pred→next = I;
  if (pred) while (I→spin) Pause();
```

As a result of this change, the previous value of `Tail` is known before the transaction that links in the node is executed. Therefore, the analogous change in `readerLock` allows the new node's `prev` field to be initialized before the transaction (line 28 in Figure 3) because the node is "private" until it is linked in, so it does not matter if its `prev` field is set before the node is linked into the list.

We further modify the algorithm so that, whenever a transaction accesses `Tail` it also checks if `Tail` contains `POISON`, retrying if so. The only differences between the `writerUnlock` and `readerUnlock` procedures in Figure 2 and their counterparts in Figure 3 are due to this change. This ensures that, when `Tail` contains `POISON`, it is accessed only by the thread that most recently set it to `POISON` performing its transaction to finish linking in its node. This way, contention on `Tail` is avoided for the transaction that links in the node, while keeping the algorithm simple.

Next various transformations are used to make transactions smaller, or otherwise more likely to succeed, as well as replacing some transactions with nontransactional accesses, thus avoiding the overhead of a executing—and possibly retrying—a transaction.

The path through a transaction can sometimes be determined by local variables, allowing this path to be factored out into its own transaction. For example, the transaction in the `writerLock` code shown above can be replaced with the following:

```
if (pred)
  atomically
    Tail = I ;
    pred→next = I;
else
  atomically
    Tail = I ;
```

This transformation allows the separate transactions to be optimized independently. Furthermore, eliminating conditional branches within transactions can be beneficial with HTM such as in Rock, which can fail due to misspeculating.

Another transformation we apply moves an assignment to `Tail` to the end of the transaction. (Other threads do not observe partial effects of the transaction, so the order in which updates happen within the transaction does not affect the correctness of the algorithm.) However, it reduces the time a transaction spends after modifying `Tail` and before committing, thus reducing the likelihood of such transactions aborting due to contention on `Tail`. Furthermore, a transaction that will access only a single shared variable—such as the one in the else clause above—can be replaced by code that performs this access nontransactionally. Applying these two changes to the transaction shown above yields the following code:

```
if (pred)
  atomically
    pred→next = I;
    Tail = I ;
else
  Tail = I ;
```

All of the transformations described above are easily seen to preserve the semantics of the algorithm, and could be performed automatically, for example by an optimizing compiler. Next we discuss two transformations that require knowledge of the algorithm.

After `Tail` is set to `POISON` by a thread t, no transaction accesses `Tail` until t updates `Tail` to a non-`POISON` value. Thus, t's update to `Tail` need not be performed atomically with linking in t's node: this access can be performed nontransactionally after that transaction commits. This allows us to refactor the code so that the assignment to `Tail` is performed nontransactionally, regardless of whether `pred` is non-NULL. Furthermore, this change results in a transaction that accesses only a single shared variable; therefore this access too can be performed nontransactionally. As a result of all of these changes, there is no longer a transaction in the `writerLock` procedure of the final `PoisonTheTail` algorithm shown in Figure 3. This progression is reminiscient of that used by Dice et al. [4] to eliminate the use of transactions from common paths of their HTM-based work stealing algorithm, while still exploiting transactions elsewhere in the algorithm to simplify it.

Finally, we note that some care is required in applying some of these optimizations, especially those that make transactional accesses nontransactional, because of issues related to the memory consistency model. For our target platform (Rock, which supports the TSO consistency model [12]), there was no need to insert any additional memory fence instructions due to the transformations we applied, but this may not be the case for some platforms.

3.3 Variations and optimizations

Some of the variations and optimizations we have explored are related to idiosyncrasies of Rock's HTM implementation discussed by Dice et al. [2, 3]. Others are not Rock-specific and are more likely to be relevant with future HTM implementations.

Prefetching As reported by Dice et al., it is sometimes possible to make transactions more likely to succeed on Rock by prefetching some variables to be accessed in the transaction before starting the transaction. After some experimentation, we have settled on prefetching the `Tail` variable before retrying transactions that will access it. We do so unconditionally before retrying transactions that always access `Tail`, and conditionally before retrying transactions that conditionally access `Tail`. Implementations of all transactional algorithms evaluated in Section 5 use this technique.

Shortcut transaction The key idea behind this technique is to precede a transaction with a short transaction that applies the same effects as the original transaction in a case that is expected to be common, so it can be optimized for that case, while having no effect in other cases. When the common case occurs, the original transaction need not be executed. Even when the common case is not encountered, the shortcut transaction can serve to prefetch some of the variables to be accessed when the original transaction is executed. For this paper, we experimented with the following shortcut transaction at the beginning of `readerUnlock`:

```
atomically
  pred = I→prev;
  next = I→next;
  if (pred && next)
    next→prev = pred;
    pred→next = next;
if (pred && next) return;
```

CAS-based ***`writerUnlock`*** We can use CAS to modify `Tail` in the `writerUnlock` procedure of `TxLock` (Figure 2), as we do for `writerLock` and `readerLock` in `PoisonTheTail` (Figure 3). In fact, this was our first attempt to use a nontransactional instruction (CAS) to modify `Tail` to reduce contention on `Tail` and resulting retries. We used the following code:

```
procedure writerUnlock( Tail , I )
  next = I→next;
  if (! next) {
    if (CAS (Tail, I, NULL)) return;
    next = I→next;
  next→prev = NULL;
  next→spin = 0;
```

The correctness of this version depends on knowledge of the algorithm. In particular, when a writer is releasing the lock, its node—call it n—has no predecessor (if it existed, it was removed before the writer entered the critical section). If the node's `next` field is NULL, then it also has no successor. In this case, removing the writer's node from the list amounts to setting `Tail` to NULL.

However, because the read of n's `next` field and update of `Tail` are not guaranteed to be atomic, the algorithm must allow for the possibility that `Tail` no longer points to n when `Tail` is modified. For this reason, we use a CAS instruction to set `Tail` to NULL only if it still points to n. If this succeeds, then there is no subsequent node to be released, and the unlock operation is complete.

If the CAS fails, on the other hand, then another node has been inserted into the list, and n is no longer the last node in the list; because the new node was inserted using a transaction, n's `next` field is already set, so there is no need to wait for it to be set, as is the case in `KSUH`. In this case, just as if n had not been the last node when its `next` field was first read, the unlocking thread simply unlinks its node by setting the next node's `prev` field to NULL, and then releases the thread spinning on that node. These can be performed with simple nontransactional stores because only the writer releasing the lock can access these fields until the next thread is released; note that it is important that these stores happen in the order shown, because otherwise, if the owner of the next node enters the critical section, it may subsequently leave and remove its node before the store to its `prev` field occurs.

While this change complicates the algorithm and correctness argument somewhat, the algorithm is still considerably simpler and easier to reason about than `KSUH`, and it still does not require TSM for nodes. We call the version of `TxLock` with this `writerUnlock` procedure `TxLock+CAS`. We can apply a similar technique to `PoisonTheTail`, resulting in the `PoisonTheTail+CAS` algorithm. The only change is that, if the CAS fails, we must wait until `next` is not null, because a writer may be poised to perform the write at lines 8 in Figure 3. The resulting `writerUnlock` code is essentially the same as that of `KSUH`.

The CAS-based `writerUnlock` procedure improves performance considerably in workloads with moderate to high numbers of write operations (Section 5) because it avoids delaying a writer that is attempting to release the lock, which is on the critical path.

Improving simple algorithms via transformations As the above discussion shows, there are many possibilities for optimizing HTM-based algorithms, but using them often makes the algorithms more complex. We have commented that some of these possibilities preserve semantics and could be applied by a compiler, while others can likely be shown to preserve semantics provided some simple properties of the algorithm are known. Together, these observations suggest that a useful strategy for designing such algorithms may be to start with simple transactional versions with relatively large transactions, and to then apply transformations in a disciplined and perhaps (partially) automated way.

Transactional Lock Elision (TLE) The well-known TLE technique [1] aims to achieve reasonable progress properties for algorithms that use best-effort HTM (such as Rock's), which do not guarantee to be able to commit transactions. Briefly, the idea is to augment transactions so that they read a lock and check that it is not held before committing the rest of the transaction. This way, an operation that is unable to make progress by committing its transaction can give up trying, acquire the lock, and perform its operation nontransactionally. Because all transactions check the lock to ensure they do not commit while the lock is held, this approach preserves the semantics of the transaction while overcoming the weak progress guarantees made by best-effort HTM. We have applied TLE to `TxLock`, resulting in the `TxLock+TLE` algorithm.

However, TLE is *not* compatible with all of the variations and optimizations we have explored. In particular, for algorithms that mix transactions with nontransactional atomic operations such as CAS (`TxLock+CAS` and all variants of `PoisonTheTail`), we do not have an effective way to prevent the instructions from taking effect while the lock is held, so we cannot ensure that transactions executed while holding the lock are atomic with respect to these operations. Consequently, enhancing an algorithm's progress properties by using TLE precludes the use of optimizations that are valuable, as shown in Section 5.

If future processors that support HTM were to also provide variants of simple synchronization operations such as CAS that are able to confirm the expected value of a separate memory location and are as fast as the underlying operation (CAS, in this case), then the "best of both worlds" would be possible: we could apply these optimizations while still using TLE for progress.

4. How transactions simplify algorithms

Clearly, `TxLock` is considerably simpler than `KSUH`. In this section, we underscore this gap by discussing how the use of transactions impacts careful reasoning about the algorithms. This discussion is based on our experiences reasoning in detail about `TxLock` and the (corrected) `KSUH` algorithm. (Note: outlines of correctness arguments, including all needed invariants, are available in [5].)

Models and proof obligations Verifying correctness of a concurrent algorithm requires a careful model of the algorithm and the environment in which it operates. It is important that the granularity of the model match that of the target execution platform. For example, it may be tempting to model the execution of a statement such as line 15 in Figure 1 as a single action. However, this ignores the fact that the two shared memory accesses in this statement are executed separately—so actions of other threads may occur between these steps—and thus fails to consider all possible behaviors. A model of `KSUH` that addresses this issue has over 60 actions in total.

In contrast, a transaction can be modeled by a single action. As a result, a model of `TxLock` requires only about 20 actions. Fewer actions means fewer cases to consider, reducing the potential for error. Furthermore, the properties needed to establish correctness are simpler because they do not have to account for the additional behaviors that are possible with finer-grained actions.

Bugs can also be overlooked by failing to faithfully model the environment or failing to check properties that are important for correct operation in that environment. For example, if a model for `KSUH` does not allow for the possibility of nodes being reused, then it will not exhibit the incorrect behavior we have identified. (Krieger et al. used a model checker to "do a full search of the state space for small numbers of requesters" [8], but did not uncover this bug, which can manifest with three requesters. One possible explanation is that their model did not allow reuse of nodes.) Similarly, to verify that `TxLock` does not require TSM for recycling nodes, the model must allow deallocated nodes to be reused for any purpose.

Exclusion properties The abstract algorithm in Section 2.1 guarantees the exclusion properties. Two writers cannot be in the critical section concurrently: this would imply that both of their nodes are first in the list (recall that threads must use distinct nodes). Similarly, if a reader and a writer were concurrently in the critical section, the writer's node would be first in the list, while all nodes before the reader's node in the list would be owned by readers, a contradiction. Thus, it suffices to show that each algorithm behaves equivalently to the abstract algorithm. Below we discuss two key reasons why this is substantially harder for `KSUH` than for `TxLock`.

Defining the list We relate the concrete data structures maintained by each algorithm to the list used in the abstract algorithm description by identifying the steps that add and remove nodes from the list, describing how to derive the abstract list from any reachable state of the algorithm, and showing that this derived list is changed by the identified steps according to the abstract algorithm.

For `TxLock`, this is relatively straightforward. A node is added to the list by transactions at lines 5–8 and 26–32, and removed from the list by transactions at lines 13–19 and 44-54. These transactions update `Tail`, and the `next` and `prev` pointers of the nodes to maintain the doubly linked list structure, except that a writer does not initialize the `prev` pointer of its node, which is never read (the `prev` field of a node is read only at line 45). Thus, we can determine whether a node is in the list by the program counter of its owner, and we can order the nodes in the list by their `next` pointers. `Tail` points to the last node in the list, unless the list is empty, in which case `Tail` is NULL. Also, the `prev` field of a reader node in the list is NULL iff it is the first node in the list.

For `KSUH`, a node is added to the list when `Tail` is modified on line 5 or 21, and it is removed from the list by a successful CAS on line 11, 43 or 52, when its predecessor's `next` field is written on line 47, or when its successor's `spin` field is set to 0 on line 15 or 56. If the `next` field of a node in the list is not NULL, then it points to the node's successor in the list. But unlike in `TxLock`, a node in the list with a NULL `next` field need not be the last node; it has a successor if the local `pred` variable of some thread at line 6–7, 22–24 or 43–47 points to that node. Such a thread will either update that node's `prev` field, or make that node the last node in the list by removing its own node (by a successful CAS on line 43). It must be shown that this definition of the nodes in the list and the successor of each such node indeed defines a list (e.g., that the successor of a node in the list is also in the list, and there is no cycle).

One key property concerns the `prev` fields of reader nodes in the list. (As in `TxLock`, the `prev` field of a node is only read by its owner in `readerUnlock`, so the `prev` field of a writer node is never read.) In `TxLock`, this is simple: the `prev` field of a reader node in the list (other than the first) points to its predecessor. However, in `KSUH`, the `prev` field of a reader node does not point to its predecessor immediately after it is added to a nonempty list (i.e., when its owner is at line 23), just before its predecessor is removed when its predecessor is not the first reader node in the list (i.e., when its predecessor's owner is at line 15 or 47), or just after its predecessor is removed when its predecessor is the first reader node in the list (i.e., when its predecessor's owner is at line 57–58). These "exceptions" complicate the algorithm and invariants considerably.

Memory lifecycle issues As we have seen, it is important to clearly understand "memory lifecyle" issues, such as how nodes are recycled, and whether and how they can be accessed when not in use. For example, to show that `TxLock` does not require nodes to be stored in TSM, it suffices to show that a thread does not access a node that it does not own unless that node is in the list. This is easily seen for all accesses within transactions because `Tail` and every `prev` or `next` field of a node in the list points to a node in the list (unless it is NULL), except for `prev` fields of writer nodes, which are never read. Outside transactions, such nodes are accessed only at lines 20, 39–40 and 55. For lines 39–40, the thread's node is still in the list, and the thread's local `next` variable points to its successor (if any), which must therefore also be in the list. For lines 20 and 55, the thread just removed its node from the list, and the thread's local `next` variable points to the node that was its node's successor (if any). In these cases, the invariants relating nodes in the list to program counters of the threads that own them imply that the owner of the successor node cannot enter the critical section before the owner of the preceding node sets its `spin` field to zero. Thus, lines 20 and 55 do not modify a node that is not in the list.

In contrast, memory lifecyle issues are much more difficult for `KSUH`. In particular, it is not possible to prove the simple property discussed above because it is not true: threads *can* modify nodes that are not in the list and not owned; this is central to the bug we have identified. This is true even with our fix, because a thread can acquire the mutex of a node previously observed to be the predecessor of its node, even after that node has been recycled. Thus, once again, the invariants used to verify `KSUH` are complicated by the need to capture subtle behaviors that cannot occur with `TxLock`.

5. Performance experiments

Next we evaluate our HTM-based RW-lock algorithms and the version of `KSUH` shown in Figure 1 (including the corrections labeled with ***). The algorithms are implemented in C and compiled with the GCC 4.4.1 compiler at optimization level -O3 in 32-bit mode (except the kccachetest, which requires 64-bit libraries). The experiments were conducted on the Rock-based system described in [2, 3]. We use both a synthetic benchmark (`rwbench`) and a more realistic benchmark that employs RW-locks (`kccachetest`) to compare the performance of `KSUH`, `TxLock`, `TxLock+CAS`, `TxLock+TLE`, `PoisonTheTail`, and `PoisonTheTail+CAS`.

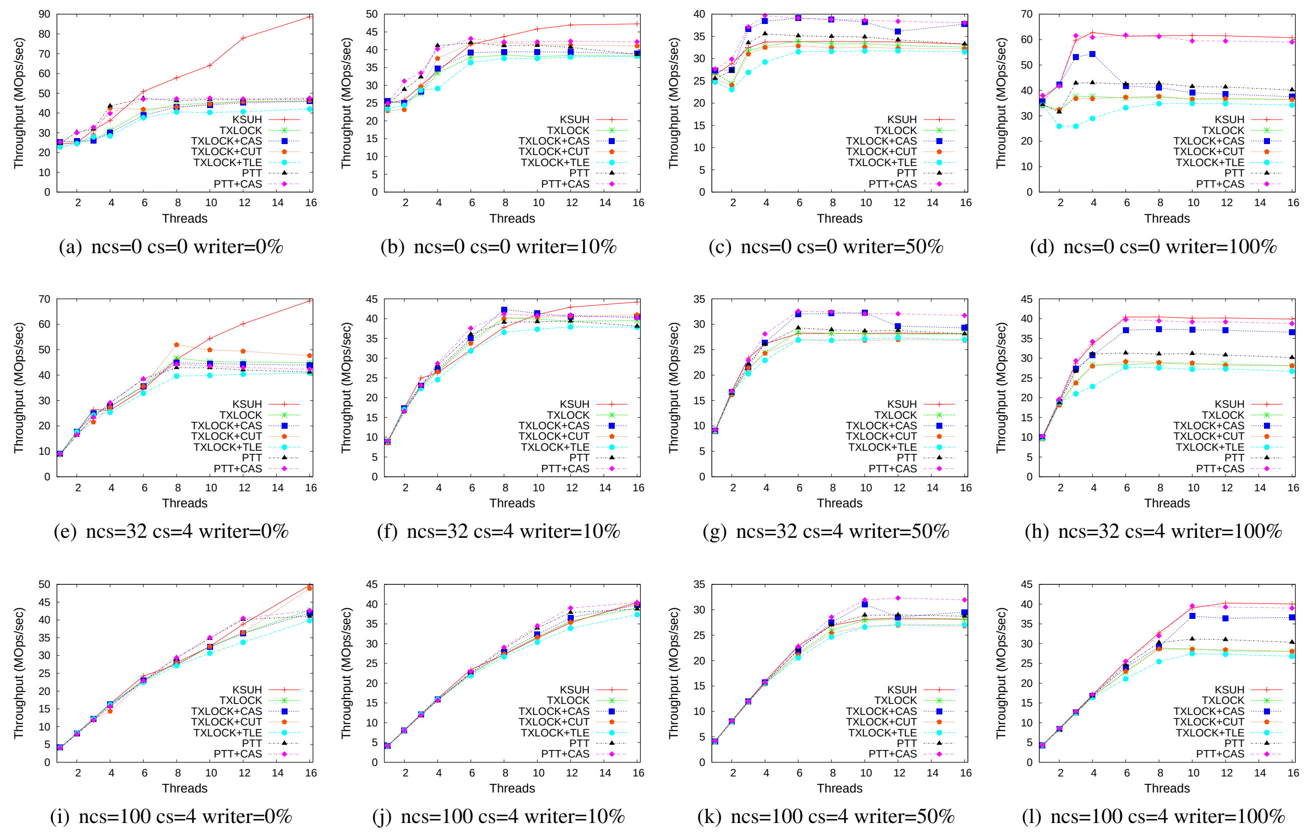
5.1 Synthetic Workloads

In `rwbench`, each thread repeatedly attempts to acquire and release a single RWLock object for reading or for writing. Benchmark parameters `cs` and `ncs` control the amount of work performed in the critical and non-critical sections, respectively. We use `rwbench` to examine the performance of the algorithms under a variety of workload scenarios, including maximum-contention "torture tests" (`cs`=0 and `ncs`=0), as well as workloads with short critical sections and a range of longer non-critical sections. The `writer` parameter controls the fraction of lock acquisitions that are for writing.

In Figure 4, each column has a fixed fraction of writers: in the left column, all acquisitions are for reading, and in the right, all are for writing. The middle two columns have 10% and 50% of acquisitions for writing, respectively. The top row is the "torture test" configuration, with no delay between lock acquisitions and releases. In the other rows, a short critical section (`cs`=4) is used, and the non-critical section length increases towards the bottom.

Some of the extreme cases may seem unimportant, because a well-designed application would not use RW-locks with empty critical sections, would not use them so frequently, and would not use an RW-lock when most acquisitions are for write. However, the focus of our paper is not only on RW-locks, but on the ability of HTM to simplify synchronization algorithms. Therefore, it is important to examine performance under heavy contention, and to explore how changes to software and/or hardware might improve performance. Furthermore, workload behavior can often be influenced by input data, so performance of an RW-lock in heavily write-dominated workloads is not unimportant. We first observe that:

- Comparing `TxLock+CAS` to `TxLock` and `PoisonTheTail+CAS` to `PoisonTheTail`, we see that the CAS-based `writerUnlock`



(a) ncs=0 cs=0 writer=0%
(b) ncs=0 cs=0 writer=10%
(c) ncs=0 cs=0 writer=50%
(d) ncs=0 cs=0 writer=100%
(e) ncs=32 cs=4 writer=0%
(f) ncs=32 cs=4 writer=10%
(g) ncs=32 cs=4 writer=50%
(h) ncs=32 cs=4 writer=100%
(i) ncs=100 cs=4 writer=0%
(j) ncs=100 cs=4 writer=10%
(k) ncs=100 cs=4 writer=50%
(l) ncs=100 cs=4 writer=100%

Figure 4. Performance comparison between `KSUH` and HTM-based RW-locks.

variant provides significant benefit as the fraction of acquisitions for writing increases.

- The `PoisonTheTail` variants almost always perform competitively with or better than their `TxLock` counterparts, and sometimes significantly better, particularly when the fraction of write acquisitions is higher. `PoisonTheTail+CAS` in particular performs well in these scenarios, as it eliminates transactions from both the `writerLock` and `writerUnlock` procedures.
- Comparing `TxLock+TLE` to `TxLock`, we see that the price of using TLE to avoid starvartion is noticeable, but usually small.
- The shortcut transaction provides a small but noticeable improvement in some cases, mostly in read-heavy workloads; we explore this issue in more detail later.

In the most realistic configurations—with longer non-critical sections, and a small fraction of acquisitions being for writing (e.g., Figures 4(i) and 4(j))—there is relatively little difference between the RW-lock algorithms, showing that HTM-based algorithms can be much simpler than the non-HTM-based algorithms on which they are based, while still delivering competitive performance.

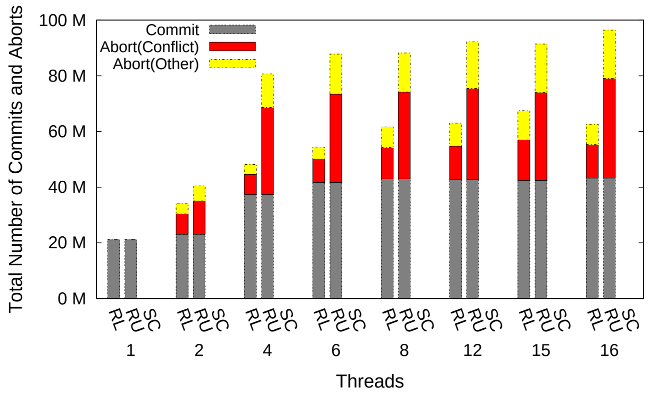
Next we discuss the read-only torture test (Figure 4(a)), where `KSUH` wins by the widest margin. `KSUH` increases throughput up to 16 threads, but the transactional variants do not scale past 8 threads. All of the transactional algorithms use transactions in `readerLock` and `readerUnlock`, and high contention between them in this test results in retrying and backoff, limiting the throughput.

The best hope to avoid this contention is via the shortcut transaction described in Section 3.3. Indeed, `TxLock+CUT` outperforms `TxLock` noticeably in a number of read-dominated cases. For example, it provides about 36% higher throughput at four threads in Figure 4(a). However, with increasing contention (more threads and/or shorter non-critical sections), its benefit fades.

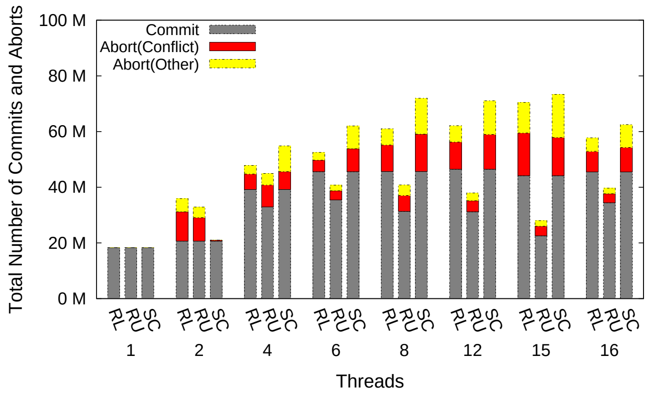
Although the shortcut transaction does not access the highly-contended `Tail` variable, it eliminates the original transaction only when removing a reader from between two nodes, and is still subject to contention with transactions removing neighboring nodes. We believe it may provide greater benefit in workloads in which readers execute longer and more diverse critical sections, resulting in longer reader chains in the list, and making it less likely for a departing reader to conflict with a departing neighbor.

The shortcut transaction (Section 3.3) is compatible with all other optimizations and with TLE. A variant (not shown) that uses *both* the shortcut transaction and the CAS-based `writerUnlock` generally gets the benefit of whichever optimization is most relevant (shortcut transaction in read-heavy cases and CAS-based `writerUnlock` in write-heavy cases). However, in the case of 50% writers, the shortcut transaction imposes overhead for little benefit, as chains of readers build up rarely, so there are few opportunities to remove a reader's node from between two neighbors.

We studied the behavior of the read-only torture test in more detail, collecting statistics on how often transactions are retried and for what reasons, as well as how often `readerUnlock` returns after executing only the shortcut transaction. First, we observed that the shortcut transaction significantly reduced the retry rate for the transactions in `readerLock` *and* `readerUnlock`, as we had hoped. However, as we have noted, the performance improvement was modest. This is primarily because shortcut transaction in most cases did *not* avoid the need to execute the original `readerUnlock`



(a) Shortcut Transaction Disabled



(b) Shortcut Transaction Enabled

Figure 5. Abort Statistics of Read-only Torture Test (RWBench with NCSLen = CSLen = 0)

transaction. It is interesting, therefore, that the shortcut does not *reduce* performance. The reason is that, if the original transaction is executed, the shortcut transaction prefetches some variables, thus making the original transaction faster and less vulnerable to abort.

Although some techniques successfully improve performance somewhat in the read-only torture test, and further improvements may be possible (for example with better backoff tuning), it is clearly difficult to match the performance of the more complex non-HTM-based algorithm under heavy contention on Rock. This is consistent with the observations of Dice et al. [2, 3], who described several implementation-specific idiosyncrasies with Rock's HTM.

To examine this issue in more depth, we also collected statistics on the reasons that transactions aborted. A noticeable fraction were due to Rock idiosyncrasies. Importantly, however, in most cases 50-75% of aborts were due to conflicts (Figure 5). This suggests that any future HTM implementation that has a "requester-wins" conflict resolution policy [2, 3] is likely to similarly make it difficult to achieve good performance in the face of heavy contention. We hope that designers will seek to improve transaction success rates, even in the face of contention, so that software built using it will be more robust to unexpected or transient contention.

With 100% writers, `KSUH` is equivalent to the MCS algorithm, and uses only one atomic instruction per lock or unlock operation, giving `KSUH` a clear advantage over the transactional algorithms that may suffer from transaction aborts and retrying. `PoisonTheTail` closes the gap somewhat, and `TxLock+CAS` closes it even more. In both cases, the reason is that transactions are replaced with atomic CAS instructions that do not abort. `PoisonTheTail+CAS`, which combines both techniques, achieves almost identical performance to `KSUH` in the writer-only workloads, using a simpler algorithm. (Recall that many of the changes made to the simple and easy-to-prove `TxLock` algorithm in order to achieve `PoisonTheTail` were via transformations that could be applied systematically.)

It is interesting to note that, in workloads with mixed readers and writers (in particular, the 50% writer case), `TxLock+CAS` and especially `PoisonTheTail+CAS` noticeably outperform `KSUH`. To understand why, note that the `writerLock` and `writerUnlock` procedures are very similar to those of `KSUH` in this case, while the `readerLock` and `readerUnlock` procedures of `KSUH` entail significant overhead for the complicated synchronization, including acquiring node locks in `readerUnlock`. On the other hand, because of the 50%-writer workload, departing readers will typically have a successor but no predecessor in the list. Thus, readers departing using the HTM-based algorithms perform a simple transaction that is unlikely to encounter contention because it will not access `Tail` and will not have any neighboring readers in the list.

Finally, we reiterate that, while TLE adds only modest overhead to `TxLock`, it is not possible to use the `PoisonTheTail` and the CAS optimizations together with TLE, because of their mixed use of transactions and non-transactional atomic instructions such as CAS. At the end of Section 3, we mentioned some possibilities for overcoming this barrier to using HTM to simplify synchronization algorithms while achieving performance competitive with the complex non-HTM-based alternative.

5.2 Kyoto Cabinet: A More Realistic RWLock Application

`kccachetest` serves as a stress test and performance benchmark for the in-memory "cache hash" (CacheDB) database. It is part of the Kyoto-Cabinet distribution, a popular open-source database package. CacheDB uses RWlocks heavily. We ran the program with the "wicked" argument, which constructs an in-memory non-persistent database and then runs randomly selected database transactions against it. A parameter specifies the number of threads. Each worker thread repeatedly performs a randomly selected operation on the database. Some operations are simple lookups or deletes, while others are more complex transactions. Each thread performs the same number of operations, and we measure completion time. The size of the key range, and thus the memory footprint, is a function of the number of threads. Results of runs with different numbers of threads are therefore not easily comparable. As a result, we do not report absolute performance of kccachetest, but rather the normalized performance compared to `KSUH`.

We ran `kccachetest` with 1 to 16 threads, taking the median of 12 trials in each case. At all thread levels, the transactional algorithms are fairly competitive with `KSUH`. `TxLock` overhead compared to `KSUH` ranges from 0.7% to 4.8%, but is usually less than 3%. The CAS optimization usually provides a small improvement (for both `TxLock` and `PoisonTheTail`). `PoisonTheTail+CAS` is particularly competitive, paying at most 2.5% overhead, but usually less than 2%, and in one case even slightly outperformed `KSUH`. The shortcut transaction mostly hurt performance by at least a small margin, but in one case it provided the best performance, outperforming `KSUH` by about 1%. Finally, to achieve better progress guarantees, `TxLock+TLE` imposes noticeable overhead compared to `KSUH`, ranging from 1.25% to over 6.75%, and it is usually over 3.5%. This show that HTM allows much simpler algorithms to be used without substantially harming performance, and that optimizations can help close the small gap. If future HTM implementations impose lower overhead and deal with contention better, such optimizations may not be as important. Nonetheless, as discussed in Section 3.3, future HTMs could support the use of TLE with such optimizations using fast conditional atomic operations.

6. Discussion

The focus of this paper has been on the use of HTM to simplify algorithms, and the KSUH algorithm and the bug we found in it served as a good example to study. However, in this section, we briefly discuss alternative solutions that do not depend on HTM.

First, other RW-lock algorithms that have been published since KSUH was published should be considered. Some of them deliver signifcantly better scalability; see [9], for example.

Next, we discuss several possible approaches that we believe allow use of (variants of) the KSUH algorithm while avoiding the incorrect behavior we have identified, without depending on HTM. First, we believe our fix (see *** in Figure 1) corrects the algorithm.

Alternatively, if it is not possible or desirable to modify the implementation of KSUH, we believe it will behave correctly if the client separates nodes into two TSM domains: one for read requests and one for write requests. The reason is that this would ensure that the problematic late store described in Section 2 would only ever target nodes used for store requests, and the prev field that gets overwritten by the late store is used only in nodes used for read requests. (This would also allow for an optimization in which the state field is written once when a node is allocated, allowing the initialization of the field during each request to be elided, but of course this would require the KSUH implementation to be modified.)

The solutions mentioned above do not change the fact that nodes must be kept in TSM, which again imposes considerable inconveniences on programmers. We believe that, given our fix to the KSUH algorithm, the only reason TSM is still required is because the locks are stored in nodes, and could be accessed after a node has been removed, as discussed in Section 2. It is interesting to note that known techniques can be used to remove this dependence. In particular, rather than having a lock per node, we could have a persistent array of locks, hashing nodes into the array to determine which lock protects a node. Care must be taken, however, to avoid deadlock, because the hashing loses the properties that ensure deadlock does not arise in KSUH. This is not difficult: if a thread's attempt to acquire the lock to which its own node hashes times out, it can release the lock on its predecessor and retry.

Finally, we note that such a solution has significant overlap with a more direct solution in which we use software transactional memory (STM) to implement the TxLock; by using a privatization-safe STM, we could still achieve a solution that does not require nodes to be kept in TSM. This raises interesting possible research directions, such as making STM compatible with single-location nontransactional accesses like those we used to reduce transaction aborts due to contention on Tail for the PoisonTheTail algorithm. It is also interesting to consider whether techniques such as those presented by Dragojevic and Harris [6] may be used to improve performance of algorithms achieved in this manner.

7. Concluding remarks

We have demonstrated the power of hardware transactional memory (HTM) to simplify and improve synchronization algorithms. We used HTM to significantly simplify a well-known synchronization algorithm, while correcting an error in it that has escaped notice for decades. The resulting algorithm is dramatically easier to prove correct, and furthermore eliminates usage constraints that apply to the original algorithm, making it more convenient to use, and improving its space requirements. We have also presented optimized versions of this algorithm using transformations for making transactions shorter, and in some cases, eliminating them entirely.

Our HTM-based algorithms perform competitively with the original algorithm under reasonable conditions. However, the transactional algorithms are not as competitive under extreme contention. We also find that using Transactional Lock Elision (TLE) to enhance the algorithm's progress properties precludes the use of some valuable optimizations. We have discussed ways in which future HTM implementations might alleviate both isuses.

We have also identified several fixes and workarounds for the original algorithm that do not depend on HTM.

References

[1] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. Transact 2008 workshop, 2008. URL http://labs.oracle.com/projects/scalable/pubs/TRANSACT2008-ATMTP-Apps.pdf.

[2] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, 2009.

[3] 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 Laboratories, 2009.

[4] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, 2010.

[5] D. Dice, Y. Lev, Y. Liu, V. Luchangco, and M. Moir. Using hardware transactional memory to correct and simplify a readers-writer lock algorithm, 2013. URL http://labs.oracle.com/projects/scalable/pubs/PPoPP2013-HTM-RWlocks-appendix.pdf.

[6] A. Dragojević and T. Harris. STM in the small: trading generality for performance in software transactional memory. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 1–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168838. URL http://doi.acm.org/10.1145/2168836.2168838.

[7] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 123–136, New York, NY, USA, 1996. ACM. ISBN 1-880446-82-0. doi: 10.1145/238721.238767. URL http://doi.acm.org/10.1145/238721.238767.

[8] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable rea,der-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 201–204, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 0-8493-8983-6. doi: 10.1109/ICPP.1993.21. URL http://dx.doi.org/10.1109/ICPP.1993.21.

[9] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 101–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584020. URL http://doi.acm.org/10.1145/1583991.1584020.

[10] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. URL http://doi.acm.org/10.1145/103727.103729.

[11] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. 34th International Symposium on Microarchitecture*, pages 294–305, Dec. 2001.

[12] I. Sparc International. The sparc architecture manual, version 8, 1991. URL http://www.sparc.org/standards/V8.pdf.

[13] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC® processor. In *IEEE International Solid-State Circuits Conference*, Feb. 2008.