Refined Transactional Lock Elision

Dave Dice Alex Kogan Yossi Lev

Oracle Labs

{dave.dice,alex.kogan,yossi.lev}@oracle.com

Abstract

Transactional lock elision (TLE) is a well-known technique that exploits hardware transactional memory (HTM) to introduce concurrency into lock-based software. It achieves that by attempting to execute a critical section protected by a lock in an atomic hardware transaction, reverting to the lock if these attempts fail. One significant drawback of TLE is that it disables hardware speculation once there is a thread running under lock. In this paper we present two algorithms that rely on existing compiler support for transactional programs and allow threads to speculate concurrently on HTM along with a thread holding the lock. We demonstrate the benefit of our algorithms over TLE and other related approaches with an in-depth analysis of a number of benchmarks and a wide range of workloads, including an AVL tree-based micro-benchmark and ccTSA, a real sequence assembler application.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques — concurrent programming

Keywords hardware transactional memory, transactional lock elision, concurrency

1. Introduction

Transactional Lock Elision (TLE) is a well-known technique that exploits hardware transactional memory (HTM) to introduce concurrency into lock-based software [9, 22]. It achieves that by attempting to execute each critical section protected by a lock in one atomic hardware transaction. When a conflict between concurrently running transactions is detected, at least one of the transactions is aborted; the execution of the corresponding critical section is subsequently retried, either speculatively (that is, on another hardware transaction) or pessimistically (that is, by acquiring the lock). The main advantage of TLE is that it can be enabled at the level of a library providing lock implementations while preserving the semantics provided by the lock-based synchronization, thus making TLE readily applicable on any architecture featuring HTM. In fact, recent Intel Haswell processors are equipped with a special Hardware Lock Elision (HLE) mode that enables TLE by using new instruction prefixes and implementing begin-fail-retry logic on the level of hardware.

Numerous studies have shown that the TLE technique can achieve linear scalability with the number of threads under ideal conditions where all or most transactions succeed [14, 17, 26]. However, in realistic applications, when some operations fail to complete on HTM (due to data conflicts, HTM capacity limits, attempts to execute unsupported instructions, etc.), the performance is negatively affected [1, 12, 14, 17, 26]. This is because in order to ensure correctness, TLE disallows concurrent execution of speculating and pessimistic threads. Thus, once there is a (pessimistic) thread executing under the lock, all other threads have to wait for it to release the lock before they can resume their speculative executions. This is true even if the pessimistic and speculating threads do not conflict over data they access.

Over the last decade, a lot of research was done in the community to allow more parallelism in cases when hardware speculation fails. The dominating approach is to use software transactional memory (STM) as a fallback instead of acquiring the lock. This research led to numerous proposals of hybrid transactional memory (TM) systems, e.g., [7, 8, 18, 23, 24]. These systems allow multiple threads to speculate on HTM and software paths concurrently provided they all perform necessary synchronization steps. While the synchronization steps might be trivial for threads executing on hardware, the steps are much more complicated for threads executing on the software path. This is because the latter are required to coordinate access to the shared data among themselves as well as with threads executing on hardware. This in turn may lead to poor performance when multiple threads fail to complete their operations using HTM and switch into the software-only path.

In this paper, we aim to improve performance of TLE by taking a middle ground between TLE and hybrid TM systems. Specifically, we allow concurrent execution of speculating threads to run on HTM along with just one pessimistic thread holding the lock. We argue that this limited concurrency is useful for many interesting cases albeit it is much simpler than full-fledged hybrid TM systems. The simplicity stems from the fact that the metadata used for synchronization of concurrently running threads is updated only by one thread running on software (and holding the lock), and is read only by threads running on HTM. Thus, from the algorithmic perspective, our work is much closer to standard TLE and can be viewed as its refinement. Furthermore, the semantics provided by a program that uses our technique is much closer to that provided by the lock-based program than to what is provided by a transactional program that uses a hybrid TM system. For example, the order in which stores to memory are executed in a critical section become visible to other threads is preserved even for threads that read some of these memory locations outside of a critical section. This allows to use our technique with lock-based programs that may access the same data concurrently inside and outside of a critical section — something that is not allowed by most transactional programs that use hybrid TM solutions (because of the STM component that usually does not support strong atomicity).

We investigate two approaches, which, like hybrid TM systems, rely on a compiler to generate two execution paths for a critical section, a fast (or uninstrumented) path and a slow (or instrumented) path. Every shared data read and/or write performed on the slow

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851162

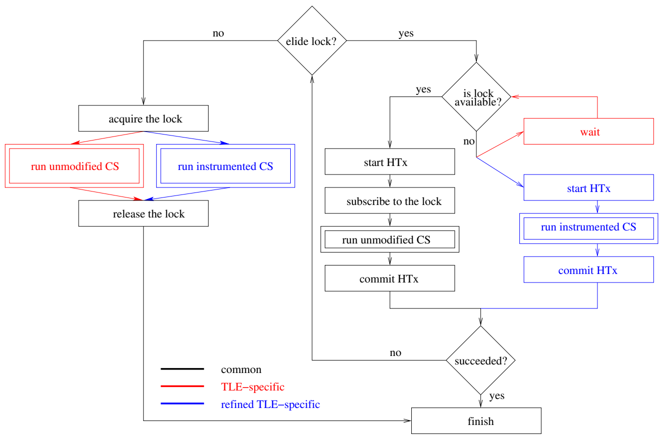

Figure 1: Design scheme for TLE and refined TLE

path for any given critical section is instrumented. Our two approaches differ from each other at the level of instrumentation required (one requires instrumentation of writes only, while another requires both reads and writes to be instrumented), and the implementation of instrumentation barriers.

In our scheme, the speculating threads execute either on the fast or the slow path (or both), while the pessimistic one always executes on the slow path. As in standard TLE, when a thread attempts to execute a critical section, it probes the lock first, and if it is not taken, it runs on the fast path (using HTM, after probing the lock again). If this attempt fails, the thread either retries speculatively or acquires the lock. However, when the thread probes the lock (before starting a hardware transaction) and realizes that it is taken, instead of waiting for the lock as in TLE, it runs on the slow path using HTM, *concurrently* with the thread holding the lock. The (lightweight) instrumentation of the slow path is responsible for making this concurrent execution safe. Figure 1 shows the design schemes of TLE and refined TLE. In the following sections we describe two possible approaches to implementing the *barriers* on the slow path, i.e., the functions invoked for every read or write. One approach, called RW-TLE, requires trivial instrumentation of writes only at the expense of allowing only hardware transactions that do not execute any writes to complete on the slow path. Another approach, called FG-TLE, requires instrumentation of both reads and writes, but allows any transaction to complete on the slow path as long as it does not conflict with the lock-based execution. In both cases, the lock-based execution uses an instrumented path as well to allow detecting conflicts with threads speculating on the slow path.

Given that our ideas rely on the instrumentation of every read and/or write performed in the critical section, one may wonder about their applicability and effectiveness. The issue of applicability can be addressed by compilers, essentially in the way GCC supports compilation of transactional code, while providing the runtime support through its builtin libitm library. The GCC compiler allows to produce both unmodified and instrumented paths, while the libitm library provides the functions to be run for certain events in the execution of a transaction, such as beginning and ending a transaction, and performing read or write. Thus, one way to implement our solution is to replace the libitm library with a custom library that provides the appropriate functionality according to the refined TLE algorithm that it implements. Therefore, our approach can also be applied to transactional programs, and provide the semantics as if all atomic blocks in the program are critical sections protected by a single global lock (SGL).

With respect to the effectiveness of the proposed scheme, we note that the refined TLE would not be helpful when most transactions succeed on HTM (because the slow path would not be utilized) or might even be destructive when most transactions fall back to the lock (because the execution under the lock will take longer due to instrumentation). Our work is motivated by workloads in which *some* of the executions on HTM fall back to the lock, which we believe are the workloads of interest in realistic applications. The actual benefit of applying refined TLE in these workloads depends on the cost of the instrumentation and the number of threads that can execute concurrently with a thread running under the lock without having data conflicts among themselves and with the thread holding the lock. As we show in Section 6, despite the lack of compiler support for inlining of barrier functions, when evaluating the performance of an AVL tree based set implementation with various workloads, we are able to get a decent performance advantage even on a small 4-core machine, and a more significant advantage on a larger 18-core machine.

2. Related Work

The idea of TLE was presented by Dice et al. [9], and is based on the speculative lock elision (SLE) idea by Rajwar and Goodman [22] from 2001. However, TLE became practically useful only in the last few years since the introduction of commercial architectures featuring HTM, such as Intel Haswell, IBM POWER8, etc. As shown in Figure 1, the implementation of TLE is fairly straightforward, and can be done at the level of a library providing lock implementations. In a nutshell, when a thread calls a lock acquisition function, a TLE implementation must decide whether the lock should be elided, and if so, start a hardware transaction and make sure the lock is free. When a thread decides to release the lock, the TLE implementation much check whether the thread runs on HTM, and if so, commit the transaction; otherwise, simply release the lock. If a hardware transaction fails for any reason, HTM is responsible for rolling back any changes that might have been made by the thread in that failed transaction, and the execution returns to the point where the TLE implementation must decide again whether to elide the lock (and start another hardware transaction), or abandon speculation altogether and acquire the lock.

Very recently, several papers pointed out that the decision whether to elide the lock and how many attempts to make on HTM should be dynamic and based on the workload, platform and other available speculation methods [12, 13]. We note that the question of how many attempts to make on HTM is orthogonal to the discussion in this paper. As a result, our experiments use a simple static policy, which retries a constant (five) number of times on HTM before reverting to lock[1].

Prior work has shown that the TLE technique achieves linear scalability when most transactions succeed in their lock elision attempts [14, 17, 26]. However, when some operations fail to complete on HTM, the scalability is severely hampered [1, 12, 14, 17, 21, 26]. This is because when a thread acquires the lock, TLE requires all speculating threads to stop and wait until the lock is released (see Figure 1). Recent work tries to reduce the number of failures to lock by reducing contention between speculating threads. For instance, Afek et al. [1] suggest to use an auxiliary lock to synchronize between transactions that fail due to data conflicts. While this idea is helpful in workloads experiencing contention, it is not very useful when transactions fail for other reasons, such as capacity limits or an attempt to execute an unsupported instruction.

[1] As we discuss in Section 6, based on the measured performance, in our experiments we have changed the number of retries used by the libitm implementation from two to five.

Another way to improve the performance of TLE is to integrate speculative execution on hardware with that on software. This is the idea behind hybrid TM systems, e.g., [7, 8, 18, 23, 24]. There, when threads fail to complete their operations on HTM, they switch to speculative attempts on software, which can be executed concurrently with other threads speculating on HTM. As noted in [18], most hybrid TM systems suffer from significant instrumentation and synchronization overhead required to ensure safety of concurrent speculation on hardware and software. The work by Matveev and Shavit in [18] builds on Hybrid NOREC [7] and presents Reduced NOREC (RHNOrec) that aims to reduce this overhead by introducing a small (aka reduced) hardware transaction into the software speculation path. Comparing to our ideas, the Reduced NOREC has an advantage that threads speculating on hardware may run on the uninstrumented path even when they run concurrently with threads speculating on software. However, the Reduced NOREC requires that all threads speculating in hardware update a global counter (clock), even when no threads are speculating on software[2]. This might unnecessarily increase contention for threads speculating on hardware, especially on architectures with large thread counts. Even more importantly, the software speculation component bears significant instrumentation overhead as it has to keep track of read and write sets, and invalidate them every time the global clock is advanced. In particular, it has to read the clock for every load instruction, adding to the contention of concurrently running hardware transactions. As we show later in the paper, these differences allow refined TLE variants to outperform Reduced NOREC in many interesting workloads despite seemingly higher concurrency allowed by the latter in software.

Concurrently with this work, Afek et al. published an idea of Amalgamated Lock Elision [2] (abbreviated as ALE, not to be confused with the adaptive lock elision mechanism under the same name in [12]), which shares many similarities with refined TLE. In particular, Afek et al.'s ALE lets only one thread in software to execute concurrently with multiple threads running on HTM. This allows the coordination between threads speculating in hardware and the one executing in software to be done through lightweight instrumentation barriers, just like in refined TLE. The major differences, however, lie in the overhead of instrumentation barriers. In particular, ALE requires instrumentation of writes even on the fast path (executed in HTM), and thus pays unnecessarily overhead even when the software-based slow path is not used[3]. Furthermore, unlike refined TLE, the software-based slow path in ALE buffers all writes and attempts to use a hardware transaction to apply them at the end of the critical section. Apart from the overhead of managing the write buffer, this means that if the write-back transaction fails and resorts to the lock, hardware transactions on the fast path will be blocked even if they do not have any conflicts on the data written by the software-based slow path.

3. RW-TLE

In this section, we present RW-TLE, a simple variant of refined TLE that requires minimal instrumentation, but allows only read-read parallelism while the lock is held — that is, it allows hardware transactions that do not execute any writes to execute and commit on the slow path as long as the thread holding the lock has not yet executed its first write instruction. While this restriction may seem too limiting, we note that some realistic workloads include critical sections that do not have any writes, or that may not execute any of their write instructions in practice. Examples of such critical sections are a look up operation in a hash table or an insert operation in a set, which does not modify the data structure when the given key is already present in the set.

```
1 write_barrier(addr, val) {
2    if (on_htm()) htm_abort();
3    write = true;
4    *addr = val;
5 }
```

Figure 2: The pseudo-code for write barrier in RW-TLE

To support RW-TLE we need to guarantee that hardware transactions abort when and if a thread holding the lock executes a write, or if the critical section executed by the hardware transaction needs to execute a write. To achieve that, we augment the lock with a boolean `write` flag. Initially, the flag is `false`. When a thread running under the lock performs a write, the instrumentation (write) barrier turns the flag on. The flag is reset again to `false` when a thread releases the lock. A thread starting on the slow path using HTM reads the value of the flag (after starting a hardware transaction), and aborts if it is set. Note that effectively the thread subscribes to this flag so that any subsequent setting of the flag will abort the execution of the subscribed thread. In the instrumentation barrier for writes, the thread running on HTM simply aborts.

Figure 2 provides pseudo-code for the write barrier. We note that this simple logic can be implemented very efficiently without any if-statements (and consequently, branch instructions) with a few bitwise operations. Also, under the `TSO` memory model no memory fence is required after setting the `write` flag, because it is guaranteed that no other write in the critical section will be visible to a hardware transaction before the store to the `write` flag will. Furthermore, note that it is enough to set the `write` flag (Line 3) only once for each critical section. Thus, the compiler may be able to eliminate some of the write barriers by instrumenting only the first write in a series of writes that are guaranteed to always execute one after another. A simple step in this direction would be instrumenting only the first write in each basic block belonging to a critical section.

Although RW-TLE allows only read-read parallelism while a thread is holding the lock, it is often able to significantly outperform the standard TLE approach, as we demonstrate in Section 6. Part of the reason that RW-TLE is beneficial for a wide variety of workloads is since quite often critical sections contain a prefix of read instructions before executing a write instruction. In many cases, this prefix is enough for short (read-only) transactions to complete using HTM.

4. FG-TLE

In this section we present the FG-TLE algorithm. Comparing to RW-TLE, it puts less restrictions on hardware transactions that can execute and commit while a thread is holding the lock, but requires slightly more complex instrumentation. In particular, both reads and writes of the critical section need to be instrumented.

4.1 Basic idea

Like most STM and some hybrid TM systems, we maintain an array of ownership records (`orecs`) [8, 15] that captures information on the addresses (or more precisely, cache lines) that are accessed

[2] More precisely, the update of the global clock is required for transactions that perform writes. However, in most practical cases it is impossible to know whether any write occurred without instrumentation. Alternatively, one may avoid the clock update when no software transactions are running if she is willing to pay the overhead of an indicator that provides this information.

[3] Given that current compilers do not allow instrumentation of writes only, this property of ALE requires providing a (trivial) instrumentation barrier for reads as well, resulting in even more overhead to the fast path.

by the critical section when executed in the software path. These `orecs` are then used to detect conflicts between concurrent executions of hardware transactions and a thread holding the lock.

Unlike STM, with FG-TLE we do not need to detect conflicts between software executions of the critical section, as there is only one such execution at a time — by the thread that is holding the lock. Thus, only this one thread updates `orecs`, and these updates are only read within hardware transactions. This difference significantly simplifies the solution and provides greater flexibility in the design choices; for example, it is safe for the thread holding the lock to refine the conflict detection granularity by resizing the `orecs` array, as long as all hardware transactions that run on the slow path read the array size. Furthermore, unlike with STM, the execution of the critical section in the software path (by the thread holding the lock) is guaranteed to succeed. This reduces the overhead for that execution and shortens the time in which other threads cannot use the fast path.

Here is a high level description of the FG-TLE algorithm:

- Threads are running on the fast path just like with TLE, checking that the lock is available.
- A thread that decides to abandon the fast path acquires the lock, and executes the critical section while recording information on its read and write instructions in the `orecs` array. In particular, *prior* to every read or write instruction, the thread uses some mapping hash function to find the associated `orec`, and marks it as owned for read or for write. The thread releases ownership of all `orecs` once it is done executing the critical section, and then releases the lock.
- While the lock is not available, other threads can still run using hardware transactions in the slow path. There, they check associated `orecs` prior to every read and write instruction, and self-abort in case of a potential conflict (i.e., if the `orec` is held for write, or if it is held for read and the hardware transaction needs to execute a write).

4.2 Implementation

Figure 3 provides the pseudo-code for the read and write barriers of our FG-TLE implementation described below.

In our implementation, we use two separate `orecs` arrays: one to record read ownership (`r_orecs`), and the other to record write ownership (`w_orecs`). The arrays are separate because otherwise a transition of an `orec` between *unowned* and *read-owned* state would unnecessarily abort all hardware transactions that read addresses that map to that `orec`. With two arrays, the read barrier by a hardware transaction on the slow path checks only the write ownership array (Line 4), while the write barrier checks both arrays (Line 18).

Furthermore, we optimized the `orecs` acquisition and release operations by using an *epoch* based scheme. In particular, we maintain a global epoch counter (`global_seq_number`), that is incremented twice by the thread holding the lock: once right after acquiring the lock, and once just before releasing it. Acquiring an `orec` is simply done by storing in it the value of the epoch counter. Threads that are executing on the slow path using HTM read a snapshot of the epoch counter *before* starting the hardware transaction, and check that an *orec* is unowned by asserting that the epoch number stored in it is strictly smaller than that snapshot. Thus, by incrementing the epoch counter right before releasing the lock, the thread that holds the lock implicitly releases the ownership of all `orecs` it owns, without causing any of the hardware transactions running in the slow path to abort.

```
/***********************************************
local_seq_number is the snapshot of the
epoch counter taken by each thread before starting
a hardware transaction on the slow path.

The fast_hash() function takes a 64bit integer
i and a number r, applies a few bitwise operations
and returns a value in the [0, r−1] range.
For our experiments we implemented a
hash function described in [25].
***********************************************/

 1 read_barrier(addr) {
 2     if (on_htm()) {
 3         uint64_t index = fast_hash(addr, N);
 4         if (w_orecs[index] >= local_seq_number)
               htm_abort();
 5     } else if (uniq_r_orecs < N) {
 6         uint64_t index = fast_hash(addr, N);
 7         if (r_orecs[index] < global_seq_number) {
 8             r_orecs[index] = global_seq_number;
 9             uniq_r_orecs++;
10         }
11     }
12     return *addr;
13 }

15 write_barrier(addr, val) {
16     if (on_htm()) {
17         uint64_t index = fast_hash(addr, N);
18         if (r_orecs[index] >= local_seq_number ||
                w_orecs[index] >= local_seq_number)
               htm_abort();
19     } else if (uniq_w_orecs < N) {
20         uint64_t index = fast_hash(addr, N);
21         if (w_orecs[index] < global_seq_number) {
22             w_orecs[index] = global_seq_number;
23             uniq_w_orecs++;
24         }
25     }
26     *addr = val;
27 }
```

Figure 3: The pseudo-code for read and write barriers in FG-TLE

Next, we addressed two sources of overhead in the slow path for the thread holding the lock. First, every `orec` is updated at most once in each execution of a critical section. We achieve that by only storing a value in the `orec` if that value is greater than the value already stored there. This is important because we avoid not just an unnecessary write, but also, as we discuss later, a memory fence that follows it. Second, we avoid the calculation of the mapping of an address to the appropriate `orec` if all `orecs` were already acquired by that thread. For that reason, we keep thread-local counters, `uniq_r_orecs` and `uniq_w_orecs`, that count how many `orecs` have been acquired for read and for write, respectively. Once one of these counters reaches the total number of `orecs`, the corresponding barrier for the thread holding the lock becomes trivial.

Finally, note that under the `TSO` memory model, it is guaranteed that threads speculating on the slow path using a hardware transaction will always see the effect of the write that acquired an `orec` prior to seeing any write done by the thread holding the lock to any address associated with that `orec`. Thus, there is no risk that the hardware transaction would see the result of a partial execution of an atomic block that is executed under the lock. Without memory fences, though, there is a risk that a hardware transaction that wrote to an address that maps to some `orec` will successfully commit before noticing that this `orec` was already acquired by the thread holding the lock, and thus, would interfere with the execution of that thread. Ideally, we would like to force a thread under the lock to execute a memory fence instruction just before a hardware transaction is about to commit; unfortunately, this is not supported by current hardware. As a result, we place a store-load memory fence after every acquisition of an `orec` (i.e., between Lines 8 and 9, and between Lines 22 and 23, respectively). This is one of the reasons why avoiding writing the same value in an `orec` is important for performance.

4.2.1 Adaptive FG-TLE

While in this paper we focus on evaluating FG-TLE as described above, we note that it should not be difficult to build an adaptive version that either adjusts the number of `orecs` for a particular workload, or even disables the FG-TLE algorithm altogether and switches to the standard TLE approach. As already mentioned, changing the number of `orecs` can be trivially done while a thread is holding the lock. The epoch numbers stored in the `orecs` could be a good indicator for whether the number of `orecs` should be increased or decreased; for example, if many `orecs` are never used, we can decrease the number of `orecs` and by that reduce the instrumentation overhead for FG-TLE (as it will become more likely that a thread executing under the lock will enjoy the optimization where the number of `orecs` that it acquired equals the total number of `orecs`). To switch to the standard TLE algorithm, all we need to do is to add a flag that is initially set and is always read by hardware transactions in the slow path, and then have the thread that is holding the lock to unset this flag before it starts to execute the critical section code without any instrumentation. Experimenting with such adaptive variants is beyond the scope of this paper and is subject for future work.

5. Limitations

As noted in Section 1, the refined TLE technique, just like TLE, tries to enhance the performance of lock-based programs, and thus aims to preserve their semantics. In particular, our technique will work correctly even with programs, which use a synchronization pattern that accesses the same data concurrently from inside and outside of a critical section (assuming that this synchronization pattern is correct in the original, lock-based program).

There are some unconventional lock use cases, however, where a lock itself may be used as a barrier to synchronize between two threads. In these cases, the synchronization between the threads is built on the assumption that a thread cannot complete an execution of a critical section associated with a lock that is held by another thread, even if the critical sections of these two threads do not conflict on any data access.

Consider the example scenario in Figure 4. Here, once Thread 2 sees that Thread 1 sets `GoFlag`, it uses an empty critical section to wait for the other critical section (that set `GoFlag`) to end, and then assumes that `Ptr` is initialized to a non-`NULL` value.

Using the refined TLE technique as described thus far is not safe for implementing this kind of synchronization pattern, as the programmer cannot assume anymore that a thread will fail to execute a critical section as long as the lock that is associated with it is held by another thread. In particular, with refined TLE, Thread 2 may successfully execute the empty critical section using a hardware transaction on the slow path while the lock `L` is held, and may thus see a `NULL` value in `Ptr`. This cannot happen with the standard TLE technique since it will never allow a thread to execute successfully a critical section associated with a lock `L` as long as `L` is held by another thread.

```
GoFlag is initially 0
Ptr is initially null

Thread 1:
  Lock(L);
  GoFlag=1;
  ...;
  Ptr = SomeNonNullValue;
  Unlock(L);

Thread 2:
  while GoFlag == 0;   // wait for GoFlag to be set
  Lock(L); Unlock(L);  // empty critical section
  Ptr->SomeField = 3;  // expects pointer to be non-null
```

Figure 4: Lock usage case not supported by refined TLE.

We note, however, that one might still use refined TLE and cope with these issues by applying the *lazy subscription* optimization [7] on the slow path. In this optimization, the speculating thread subscribes to the lock right before committing its transaction (as opposite to right after starting its transaction, as it is done in the fast path). While this subscription may reduce the benefit of the suggested TLE refinement approaches, numerous papers have suggested that lazy subscription can still be very helpful [1, 4, 11, 17]. Note that while applying this technique to standard TLE is subject to numerous pitfalls [11], applying it to RW-TLE and FG-TLE is always safe due to the instrumentation of the slow path.

6. Performance Evaluation

In this section we provide a detailed performance evaluation of our algorithms using several micro-benchmarks and a real application for genome sequencing, namely ccTSA [3]. We compare our algorithms to TLE as well as to a hybrid TM approach, RHNOrec[18], that uses an STM as the fallback option (and thus allows multiple threads that failed in HTM to run in parallel with each other). We analyze the difference between these approaches, as well as between the different variants of the refined TLE algorithm.

6.1 Experimental setup

We ran our experiments on two machines that support HTM. The first is a Haswell Core i7-4770, a single socket 4 cores machine (2 hyper-threads per core, 8 hyper-threads in all) running at 3.40GHz and powered by Oracle Linux 7. The second is an Oracle Server X5-2, a dual socket system with two Xeon E5-2699 v3 processors; each processor has 18 cores (2 hyper-threads per core, for the total of 36 hyper-threads per socket) running at 2.30GHz and powered by Ubuntu 15.04. We only ran experiments that use a single socket on Xeon (using more than one socket seems to have a significant impact on HTM performance, and investigating this impact is beyond the scope of this paper). The machines were set up in the performance mode (i.e., the power governor was disabled, while all cores were brought to the highest frequency), with the turbo mode disabled. This was done to reduce noise from the power management system. Furthermore, to avoid the impact of random thread placement on the bigger machine (Xeon), we pinned threads to cores such that thread i and $i+18$ are sharing the same core (so that threads 1-18 use all cores in a socket and only then we start sharing the cores with threads 19-36). When not specified otherwise, the evaluation refers to the Xeon machine.

6.2 Experimentation with an AVL tree-based benchmark

We start our presentation with an in-depth analysis of a set micro-benchmark implemented with an AVL tree (a balanced, internal binary tree). The AVL tree data structure is used in various systems, including the OpenSolaris operating system (where the address space of each process is managed by an AVL tree [5]) and the ZFS file system. Our evaluation is based on the variant of the AVL tree implementation used by those systems.

To provide the necessary instrumentation for the code of critical sections for set operations (Insert, Remove and Find), we used the GCC extension for supporting transactional programs, and implemented the refined TLE algorithms in a library that conforms to the libitm ABI.[4] We note that although our approach is designed for lock-based programs (and provides the semantics that are close to such programs), the need for instrumentation of all the code that is run under the lock made the transactional programming model, and the compiler support for it, a good venue for evaluating our algorithms.

In our experiments we had multiple threads accessing the shared set (implemented with an AVL tree) protected by a simple test-and-test-and-set lock with exponential backoff. All threads were synchronized to start at the same time (after a warm-up period) and then perform work for 5 seconds. During that time, each thread performed operations chosen according to a given distribution (e.g., 60% Find, 20% Insert and 20% Remove), and with a key that is chosen uniformly at random from a given key range. In all experiments we initialized the set with half of the keys from that range, and kept the probability for Insert and Remove operations equal in order to keep the data structure approximately at the same size during the experiment. We report the total throughput, i.e., the total number of operations done by all threads per time unit (ms). Each experiment was run 5 times, and the median throughput is reported. We note that the variance of the reported results is negligible. We also present various performance statistics as measured for runs that yielded the median throughput result.

6.2.1 Refined TLE vs. TLE

In this section, we present and analyze the performance of the refined TLE variants comparing to the one achieved with TLE, on a wide range of workloads. We experimented with the following variants of refined TLE: RW-TLE, FG-TLE(1), FG-TLE(4), FG-TLE(16), FG-TLE(256), FG-TLE(1024), FG-TLE(4096) and FG-TLE(8192) (where FG-TLE(X) indicates the FG-TLE algorithm with X `orecs`). For all variants, we set the number of trials before falling over to the lock to be 5, and we spin until the lock is not held after every failure [16]. For the refined TLE variants, we do not hold a hardware transaction failure on the slow path against this count, as the whole idea there is to allow optimistic attempts on HTM while the lock is held. (None of the TLE, refined TLE or our lock implementation address fairness or anti-starvation concerns. It is trivial to add an anti-starvation mechanism to these synchronization methods, though.)

Our workloads consist of key ranges of 8192 and 65536 (i.e., set sizes of 4096 and 32768). The workloads use Insert/Remove probabilities of 0%, 10%, 20% and 50%, and the rest is Find operations. We note that since the set is always holding roughly half of the keys in the range, an Insert or Remove operation only modifies the set in half of its invocations. Thus, even in an experiment with 50% Insert/Remove, 50% of operations end up being read-only.

Figure 5 shows the throughput results for the various workloads on the Core i7 and Xeon machines. This figure also includes curves for the hybrid TM and STM results that we describe in details in the next section. For easier comparison between different setups, we normalized the throughput of all experiments in a given setup by the throughput of a single thread run that uses the lock (the Lock curve), so what the charts really present is the speedup comparing to a single thread running under the lock.

As the figure clearly shows, the refined TLE variants are competitive with TLE when there is no or very little contention (e.g., on the smaller Core i7 machine when running with less than 50% Insert/Delete ops, and on the bigger Xeon machine in the read-only experiment). Once there is even a little bit of contention, though, we can see that TLE performance crashes, and is generally outperformed by a large margin by our refined TLE algorithms.

The figure also shows that the choice of refined TLE variant can make a significant difference. Generally, the only variants that can handle the contention reasonably well for all setups at any thread count are those with large number of `orecs` (FG-TLE(1024) and above). However, at lower thread counts, it is often the case that a lower number of `orecs` are doing significantly better. For example, in the 20% Insert/Remove experiment with 8192 key range on the Xeon machine, we clearly see that the variant with a single `orec` (FG-TLE(1)) outperforms the variants with 4 and 16 `orecs` at lower thread count, especially at thread count 8–18.

To explore this phenomenon, we augmented our code with various lightweight statistics, and calculated the total time spent while the lock is held, how many acquisitions of the lock (and hence successful executions of critical sections) we had during that time period, as well as how many hardware transactions successfully committed (by running in the instrumented path) during that time. That allowed us to examine the "slow path throughput" for all variants, i.e., the throughput of executions under the lock as well as executions that use HTM in the instrumented path during the time period when these are allowed to run concurrently. The results are presented in Figure 6. (Note that the scale for the SlowHTM chart on the left is a factor of 10 higher than that of the Lock chart on the right.)

Looking at the Lock chart first, we can clearly see that the RW-TLE variant incurs the least overhead, and hence delivers the highest throughput. Right after that we have the FG-TLE variants with 1, 4 and 16 `orecs`, and then the four variants with larger number of `orecs` perform approximately the same. The reason for this is simple: recall the optimization that allows us not to check `orecs` when running under lock once the total number of `orecs` we updated reaches the total number of `orecs` we have (cf. Section 4.2). The lower the number of `orecs`, the faster we get to the point that this optimization is effective. Furthermore, with the 256, 1024, 4096 and 8192 `orecs`, we are unlikely to ever get to this point with a tree of 4K nodes, which is why these four variants perform more or less the same (and not nearly as good as the ones with lower number of `orecs`). This effect is demonstrated in Figure 7 that shows execution time under lock normalized to the time measured for the lock-based execution *with the same number of threads*. This normalized execution time shows a clear correlation to the number of `orecs` when this number is smaller than 256. Noticeably, even the very lightweight RW-TLE variant has a non-negligible overhead compared to the lock-based execution. This is due to lack of support for inlining of instrumentation barriers in the GCC compiler.

Along with that, the reason that the variants with the higher number of `orecs` win at the higher number of threads can be seen in the second part of Figure 6 (SlowHTM). This chart shows the throughput of hardware transactions on the slow path, that is, when some thread is executing under lock. As expected, a higher number of `orecs` allows more parallelism between the thread holding the lock and threads executing hardware transactions, which results in significantly higher throughput for these hardware transactions.

[4] As mentioned in Section 1, libitm is an external library, delivered with GCC, that provides the runtime support for transactional programs.

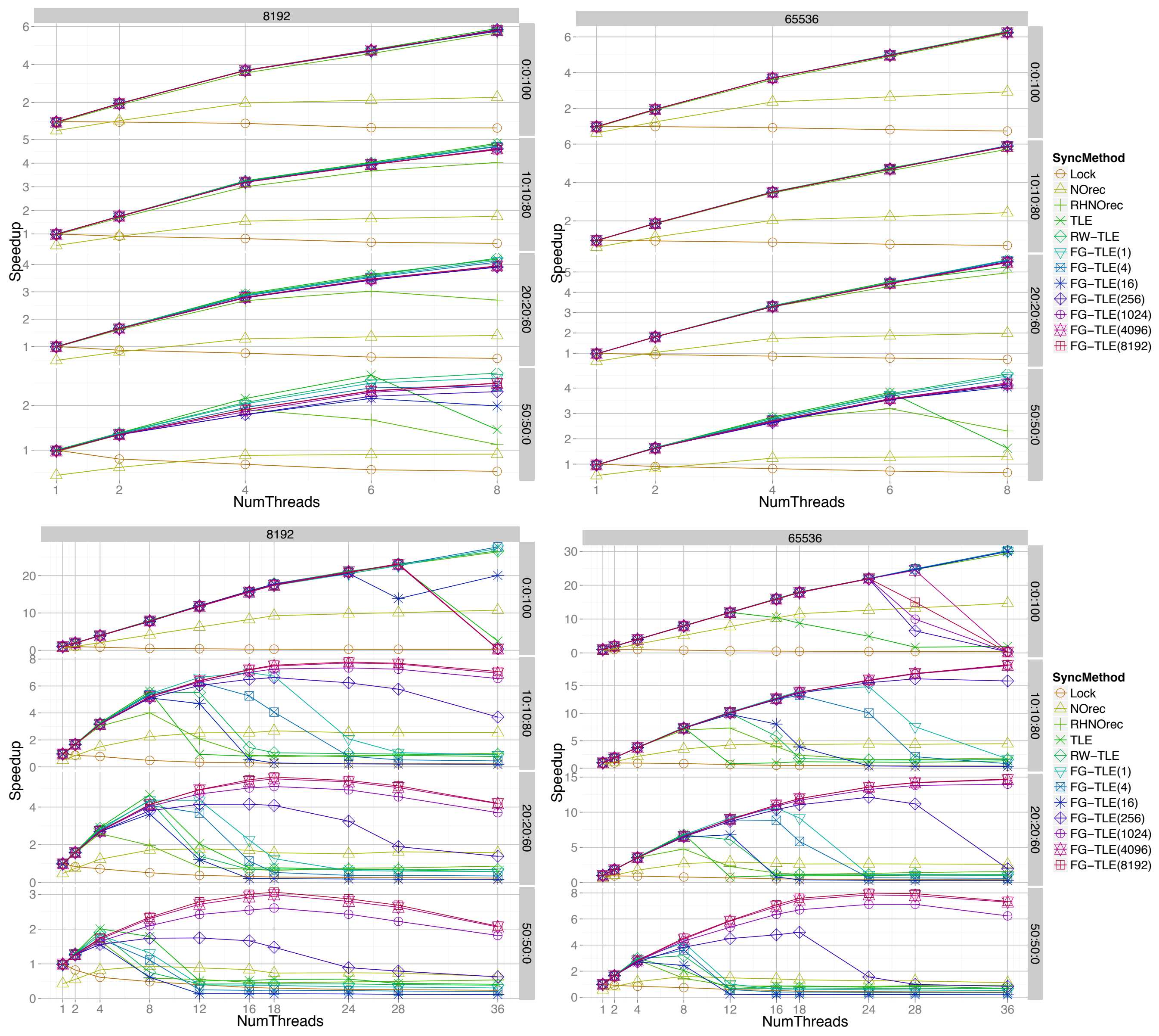

Figure 5: AVL tree-based set throughput on Core i7 (top) and Xeon (bottom), normalized with the throughput of a lock-based single-threaded execution. Head tiles indicate key ranges, side tiles indicate operation distributions (Insert:Remove:Find)

The gap between variants with the higher number of `orecs` from those with the lower number of `orecs` increases once we have enough threads that run in parallel with the thread holding the lock. This explains the tradeoff we see — with only a few threads, the shorter the execution under the lock is better, because the amount of parallelism we gain from running in parallel with the thread holding the lock is not worth the additional cost of instrumentation barriers. At some point, though, there are enough threads that could succeed using HTM if we had enough `orecs` (and thus less conflicts), and allowing these threads to commit can easily cover for the additional cost of instrumentation barriers.

Finally, it is interesting to note that while FG-TLE(1) performs worse (better) than the FG-TLE variants with large number of `orecs` on large (small, respectively) thread counts, it is always better than FG-TLE(4) and FG-TLE(16). As already mentioned, with 16 `orecs` or less, the execution under lock normally updates all the `orecs` and thus might prevent progress of any concurrent execution on the slow path. Yet, the execution under lock is faster for FG-TLE(1) as the optimization that allows not to check `orecs` kicks in sooner (cf. Figure 7). As a result, while transactions on the slow path have more or less the same chance to succeed with 1, 4 or 16 `orecs`, they switch faster to the uninstrumented path as the execution under lock completes faster. Thus, increasing the number of `orecs` must deliver a significant increase in the SlowHTM throughput (as indeed happens with 256 `orecs` or more) in order to cover for the delay in transitioning back to the fast path. As Figure 6 shows, however, the throughput of SlowHTM for FG-TLE(4) and FG-TLE(16) is actually lower than for FG-TLE(1) as executions under lock take longer while executions on the slow HTM path cannot proceed.

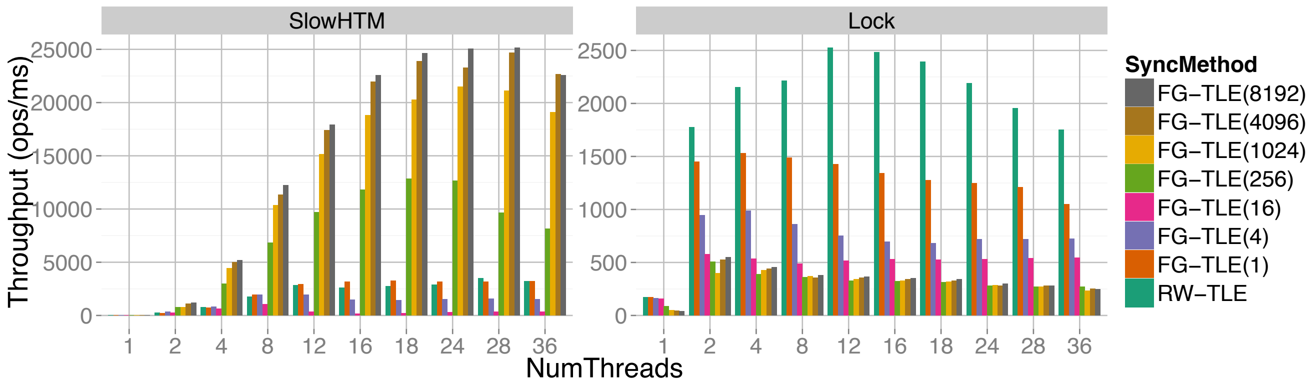

Figure 6: Slow path throughput for all refined TLE variants. Key range is 8192, 20% Insert/Remove operations, run on the Xeon machine.

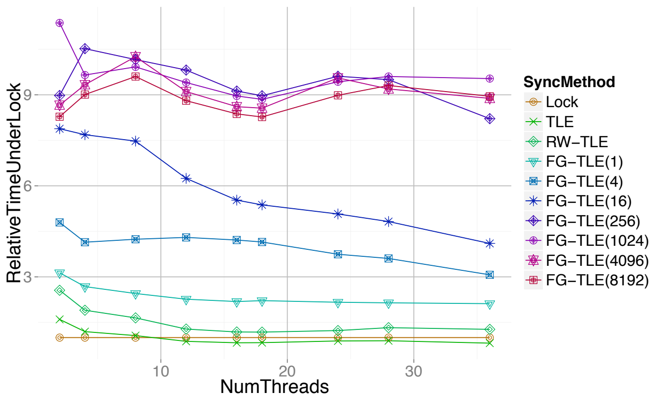

Figure 7: Execution time under lock normalized to the time measured for the lock-based execution with the same number of threads. Key range is 8192, 20% Insert/Remove operations, run on the Xeon machine.

6.2.2 Refined TLE vs. Hybrid TM and STM

A key property of the refined TLE approach is that it only allows one thread at a time to execute a critical section in software (under lock), while all other threads must yield to it; that is, they can execute critical sections using HTM as long as they do not conflict with the thread holding the lock, and abort otherwise. We made this design choice, which significantly limits the potential parallelism for threads running in software, in order to keep the instrumentation code (for software and HTM paths) as simple as possible (and lower its overhead). Besides, we aimed to guarantee progress for the thread falling to software so that we can get back to run on uninstrumented HTM path as soon as possible. A natural question to ask is whether giving up the parallelism in software indeed pays off. In other words, whether we would be better off using a hybrid solution that uses HTM and falls back to STM. To address this question, we evaluated our workloads using the RHNOrec algorithm [18] that uses an enhanced variant of the NOrec STM [6] as the fallback mechanism.

The NOrec STM has some appealing properties: it has relatively low overhead when running single-threaded (or even with multiple threads when most of the transactions are read-only), and it is not susceptible to false conflict aborts (i.e., aborts due to another transaction writing to a place we have not accessed). Thus, using NOrec instead of a single global lock can add some parallelism to the execution in software with low overhead comparing to other STM solutions, and with a low probability for such an execution to abort (as with most of our workloads, the chance for a real conflict on the access to a tree node is quite low).

The hybrid RHNOrec [18] algorithm enhances NOrec in two ways:

1. When a software transaction is ready to commit, instead of acquiring the global lock it attempts to execute the commit phase using a hardware transaction. If committing using HTM fails, the algorithm eventually resorts to acquiring a global lock that halts the execution of all hardware and software transactions for the duration of the commit operation. The commit operation (either in HTM or under lock) includes the increment of a global timestamp that indicates to other transactions that they need to validate their read set.
2. It allows transactions to run entirely in HTM while software transactions are running, as long as the former increment the global timestamp when they commit.

Thus, like the refined TLE algorithm, threads can run using HTM concurrently with threads that fall to software with low additional overhead. In fact, RHNOrec does not even require hardware transactions that run on the "slow" path to use instrumentation, as the increment of the global timestamp is done right before committing the transaction (after checking if any software transaction is currently running). So, in a sense, the overhead for hardware transactions with RHNOrec that run concurrently with software transactions is much lower than with refined TLE. At the same time, RHNOrec requires all hardware transactions to update the same location (global timestamp), which may create conflicts, especially when the number of threads increases.[5]

We implemented the RHNOrec algorithm using the same infrastructure that we used for the refined TLE implementation, i.e., by building an external library that implements the libitm ABI. We used the same number of trials in HTM on the path that does not involve software transactions, as the number we used for the refined TLE variants (5). In addition, we allowed software transactions to retry up to 5 times on HTM before resorting to the global lock. Finally, we also implemented the original NOrec (STM) al-

[5] We are aware of another very recent variant of RHNOrec [19], which is significantly different and does not have all the similarities that we mentioned to the refined TLE approach. Thus, we believe the comparison of refined TLE is more meaningful to the variant of RHNOrec we consider in this paper [18].

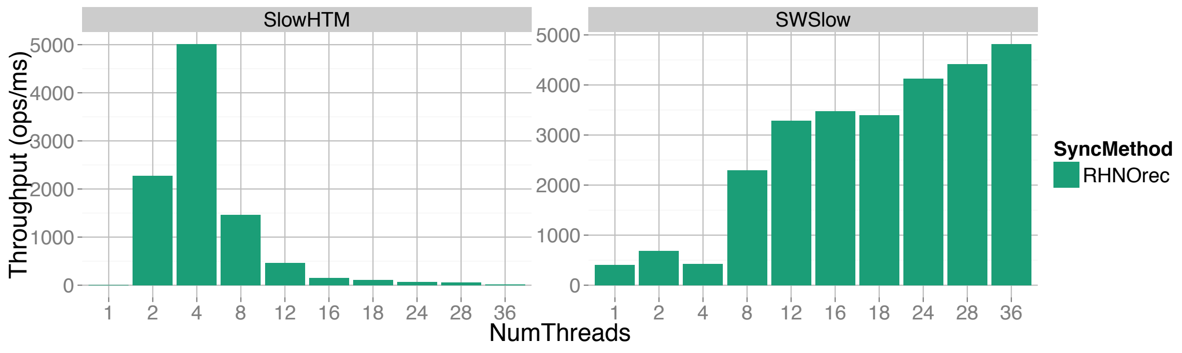

Figure 8: Slow path throughput for RHNOrec. Key range is 8192, 20% Insert/Remove operations, run on the Xeon machine.

gorithm [6], and used it as a reference point for a comparison to a software-only transactional solution.

The throughput results for experiments with RHNOrec and NORec are shown in Figure 5. When there is no or very little contention (e.g., read-only workloads or workloads that include update operations and use a low number of threads on the Core i7 machine), all our refined TLE variants deliver practically the same performance as that of RHNORec. This performance level is higher or the same as the performance delivered by TLE, and significantly higher than what is delivered by the software-only NOrec solution. However, once we increase the level of contention, even with a relatively low number of threads on the Core i7 machine, the refined TLE algorithms outperform RHNORec by a large margin, and this gap increases significantly when moving to the bigger Xeon machine. In fact, even at 16 threads with only 10% Insert/Remove operations with the 8192 key range, the performance of RHNOrec drops substantially so that it gets outperformed by NOrec.

Perhaps, these results may seem surprising at the first glance given the additional parallelism in software and the lighter overhead for hardware transactions that RHNOrec provides. To shed some light on this behavior, we added a statistic that measures the total amount of time spent running software transactions[6]. Given this time, we can calculate the total throughput of software transactions, as well as of hardware transactions that have to increment the global timestamp.

Figure 8 presents this data for the same experiment as the one presented in Figure 6. Comparing these two figures, one can clearly see that the use of STM helps to increase the throughput of executions in software comparing to our refined TLE solutions: at a high thread count, RHNOrec delivers close to 5000 executions per millisecond, while neither of the refined TLE solutions delivers more than 2500 executions per millisecond (and most of them do not get beyond 300 executions per millisecond). However, the higher throughput in software comes in the expense of significantly lower throughput in hardware, as can be clearly seen on the left side of Figure 8. Thus, overall, the total throughput while software transactions are running is significantly lower than that delivered by the refined TLE variants, which explains the superior performance of the latter.

This phenomenon of most executions diverting to the software path is further demonstrated in Figure 9. It shows the ratio between the number of transactions that committed using a hardware transaction in the fast path (no timestamp update), hardware transaction in the slow path (that update the timestamp) software transaction that used a hardware transaction for committing, and a software transaction that fell back to the global lock. The figure clearly shows that at 16 threads and above, there are almost no transactions committing on hardware.

We believe that the reason for this is simply contention on the global timestamp variable. In particular, each read operation in a software transaction has to check the timestamp to see if it needs to validate its read set. These tests keep the cache line for the timestamp in a shared mode for a relatively long time, as read operations

[6] Note that RHNOrec keeps a counter for the number of running software transactions anyway so that hardware transactions can take the fast path and skip the increment of the timestamp when no software transactions are running. Thus, keeping track of the total time when this counter is non-zero is trivial.

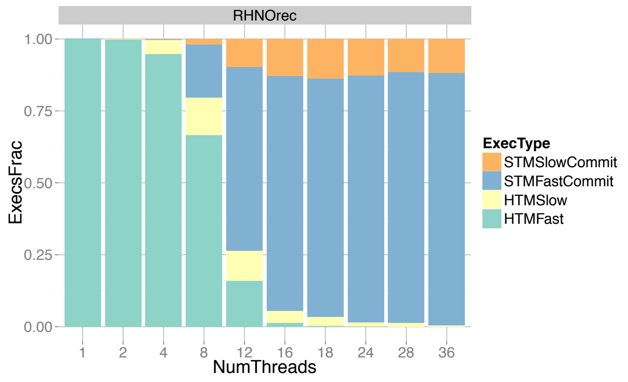

Figure 9: Execution type distribution for RHNOrec. Key range is 8192, 20% Insert/Remove operations, run on the Xeon machine.

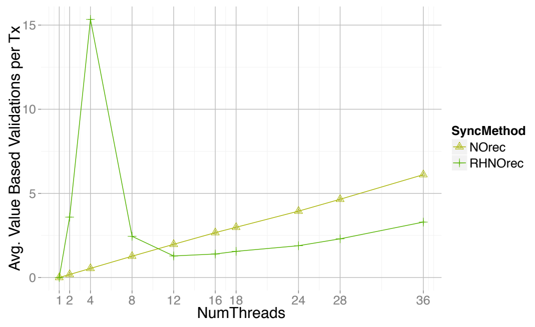

Figure 10: Validation frequency. Key range is 8192, 20% Insert/Remove operations, run on the Xeon machine.

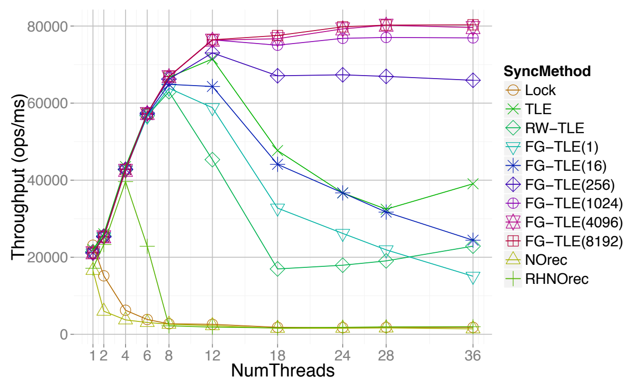

Figure 11: Throughput of the bank accounts micro-benchmark.

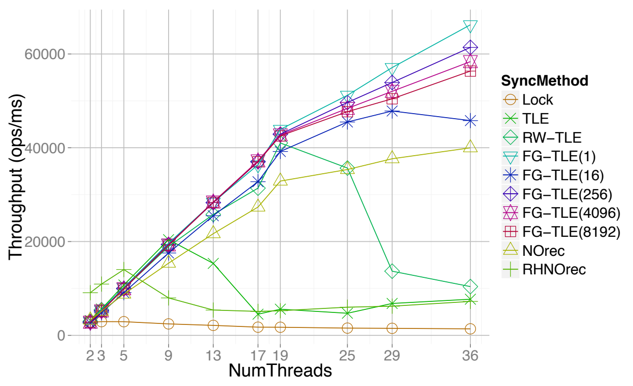

Figure 12: Total throughput in an AVL tree experiment where one of the threads executes an HTM-unfriendly instruction.

are frequent. Moreover, unlike with the refined TLE variants, hardware transactions in RHNOrec *do not yield* for software transactions. Thus, when a hardware transaction manages to commit after getting exclusive ownership of the cache line for the timestamp, it forces all software transactions to revalidate their read set, slowing them down and causing even more reads of the timestamp variable. Figure 10 shows the number of read set validations per software transaction with RHNOrec comparing to that with the NORec algorithm. There, as long as the number of threads is small enough so that hardware transactions still manage to commit successfully on the slow path, the number of validations in RHNOrec skyrockets. Along with that, as software transactions spend more time reading and validating, the harder it gets for hardware transactions to commit, as the chance for a conflict on the timestamp variable increases. Thus, we see here simply an example for the lemming effect [10]: once there are enough software transactions executing reads and validate operations, hardware transactions that try to increment the timestamp fail, and eventually switch to the software path, making it even harder for other hardware transactions to commit.

6.3 Evaluating corner cases

In this section we describe experiments that we design to evaluate refined TLE in some corner cases.

The first case is a micro-benchmark where all accesses to shared data are read-modify-write operations, i.e., operations that read values from a few memory locations, perform some short calculation and write updated values to the same locations. In particular, note that unlike in the case of AVL trees, all critical sections in this micro-benchmark perform at least one write.

The micro-benchmark uses an array of 256 counters, which represent balance in bank accounts. All accounts are initialized to the same balance, and then each thread at each iteration picks two different accounts uniformly at random, and transfers a random amount of money from one account to the other. Only the transfer operation is in the critical section; picking the amount and the two accounts is done beforehand. Since the transactions are very small, they are very likely to succeed in HTM as long as there is no conflict (we padded each account counter so it is in its own cache line). However, with only 256 accounts, the probability of collision goes up with the number of threads.

Figure 11 presents the throughput (number of money transfers per ms) as the function of the number of threads, with various synchronization methods. While TLE scales well up to 12 threads, its performance degrades significantly after that point due to collisions between threads on (at least) one of the bank accounts. On the other hand, despite the instrumentation overhead, refined TLE manages to perform well even at a high number of threads, especially with variants that use a high number of `orecs`. This is not surprising, because these variants allow more parallelism while a thread runs under the lock. In other words, these variants refine the concurrency control of TLE by disallowing only transactions that may be conflicting with the lock-based execution from running in parallel, selectively restricting the parallelism when there is too much contention. Finally, we note that, unsurprisingly, both NOrec and RHNOrec perform very badly here, as all transactions do writes, which are scenarios where NOrec-based algorithms do not perform well.

The other corner case that we explored is where there is one thread that consistently executes an operation that cannot succeed using HTM, while many other threads are running operations that do not conflict with each other. In particular, using the AVL tree discussed earlier, we set up an experiment in which one thread executed Insert and Remove operations (at equal probability) augmented to include an instruction that aborts a HW transaction (we simply divide by zero when a hardware transaction is running). In addition, another group of threads was running Find operations. We experimented with two versions: one where the instruction that aborts HW transactions is placed at the beginning of the critical section (before any shared data is accessed), and one where it is placed as its last operation.

The results for the large tree size (64K key range) and the version with an HTM-unfriendly instruction placed at the end of the critical section are shown in Figure 12. The smaller tree size and/or the version with an HTM-unfriendly instruction at the beginning of the critical section produced similar results. As one might expect, TLE performs badly with the increase in the number of threads as once there is a thread holding the lock, all other threads are blocked and wait for the lock release. At the same time, FG-TLE variants scale across all thread counts, utilizing the fact that they allow speculation on HTM to proceed concurrently with a lock holder. RW-TLE scales well up to 19 threads and then crashes. We believe the reason for that is the different strategy it uses for returning to the fast path. While FG-TLE does not abort hardware transactions running on the slow path when the lock is released, RW-TLE does so, trying to switch into the fast path more eagerly. In this experiment, however, given that some executions consistently fall back to the lock, this eager strategy causes a lemming effect in RW-TLE when the number of threads increases. Notably, RHNOrec performs poorly since the thread executing an HTM-unfriendly instruction always ends up running in the software path, creating contention on the global timestamp variable, similarly to the effect described in Section 6.2.2. Note that without any instrumentation, the fast path of RHNOrec cannot avoid incrementing this timestamp,

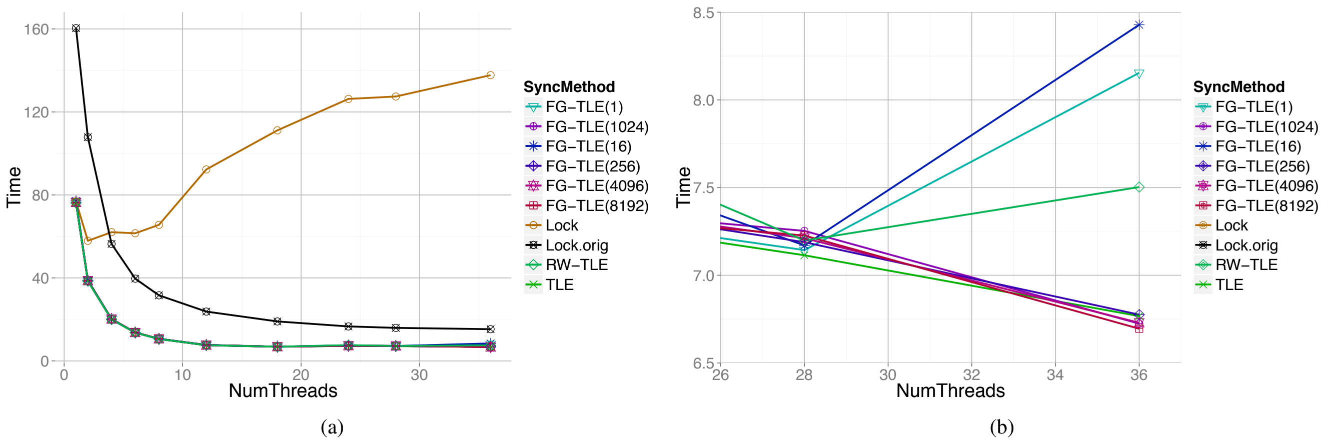

Figure 13: Runtime (in seconds) of the ccTSA application: full thread range (a), and zoom on high thread counts (b).

even for transactions that perform Find and thus do not write. Along with that, NOrec does not have this issue and thus scales up to 36 threads, although still loosing substantially to FG-TLE.

6.4 Experimentation with ccTSA

6.4.1 Overview and transactification of ccTSA

The ccTSA software [3] is an open source de novo gene sequence assembler (`https://code.google.com/p/cctsa`). It gets as an input many short DNA fragments known as *reads*, and tries to find the whole genome by looking for overlap between them. This is done by extracting subsequences of length k from these reads, denoted as *k-mers*, and building the De Bruijn graph from them, representing the extent to which they overlap.

ccTSA is a multithreaded lock-based software that scales well to dozens of threads when applied to real genome data. Its scalability is achieved by splitting the main hash-map, which is used to store k-mers, into thousands of hash-maps (4096, by default), each protected by its own lock, and hashing each read to one of these hash-maps. The use of multiple hash-maps enhances the parallelism not only during the reads storage and removal phases, but also during the processing phase, in which these hash-maps are treated as chunks of work that are claimed and processed by worker threads.

Our transactified variant of ccTSA removes most of this complexity by simply using a single hash-map for all k-mers, and having each thread save the reads that it puts into the hash-map, in a thread-local vector. Using thread-local vectors reduces the need for coordination between worker threads and speeds-up traversals over k-mers during the processing phase. This simplified version is much closer to what a sequential implementation of the algorithm would look like, but has very poor scalability if the lock over the central hash-map is not elided. As we show later, however, the simplicity of our approach significantly reduces the single thread runtime overhead, even when using the lock without ever eliding it.

The only other technical change that we applied to the original code was replacing the STL hash-map with our own transaction-safe hash-map implementation. Our implementation also uses C++ templates, and we instantiated it with exactly the same types and hashing functor as those used by the original ccTSA code.

One of the most interesting points in the transactification process is the extent to which we could exploit the fact that the refined TLE mechanisms provide semantics that are much closer to those of lock-based programs, as opposed to those of transactional programs. In particular, with refined TLE we are guaranteed that an execution that fails to software never aborts. Therefore, using "transaction-pure" annotations, we could avoid compiler instrumentation of any functions called within an atomic block if they perform only thread-local updates, or if they use their own synchronization mechanisms that make them thread-safe. Examples for such functionality in ccTSA are the storage of k-mers in thread-local vectors and occasional calls to (thread-safe) `malloc` for allocating new chunks of memory. Note that these annotations are not safe in RHNOrec (and in general, in any hybrid or software TM system in which software-based executions may abort). Thus, we do not use RHNOrec and NOrec in the evaluation of ccTSA.

6.4.2 Experimental results

Our experiments were run using 36 base pair reads from the E.coli organism that were provided with the original software, and used the default k-mer length of 27 (that is, k=27). The results in Figure 13 show the total time to complete the work as we increase the number of threads, both for the original ccTSA implementation, and for our transactional variant that uses a different synchronization mechanism described above.

Comparing first only the lock-based solutions, we notice that at a single thread, our simplified lock-based solution (Lock) is more than 2x faster than the original lock-based implementation (Lock.orig). However, it also scales negatively beyond 2 threads, while the original implementation scales nicely all the way up to 36 threads. This is yet another example for additional overhead that is often paid in fine grained locking solutions in order to provide good scalability, as previously noted by McSherry et al. [20].

Once we add lock elision to our variant, though, it scales similarly or better than the original lock-based implementation, but without paying the additional overhead of fine-grained locking. In other words, the transactified variant with the elided lock outperforms the original implementation (that uses fine-grained locking) by more than 2x at any number of threads.

As for the difference between TLE and refined TLE, for this experiment, all lock elision variants perform similarly well, where FG-TLE(8192) is doing the best among the refined TLE variants at high number of threads, and even slightly outperforms TLE at that point. Indeed, when we zoom in to the last two data points (28 and 36 threads), shown in Figure 13(b), we can see that the high number of `orecs` is required for the good scalability. In particular, RW-TLE, FG-TLE(1) and FG-TLE(16) scale negatively beyond 28 threads, and the only variants that do better than TLE at 36 threads are those that use at least 1024 `orecs`.

To shed more light on these results, we measured the fraction of all atomic block executions that acquired the lock. All lock elision variants rarely fall to the lock (the maximum failure rate is 0.15%, for TLE at 36 threads), which explains their good scalability. In particular, these results confirm the intuition presented in Introduction regarding the (lack of) expected benefit of refined TLE over TLE in scenarios where most transactions succeed on HTM. Along with that, at high numbers of threads refined TLE variants typically fall to the lock less frequently than TLE. In fact, TLE has a jump in the number of failures between 28 and 36 threads, which we believe is the reason that FG-TLE(8192) slightly outperforms TLE at 36 threads. As we showed before, though, the refined TLE variants also take significantly longer time to execute under the lock, partly due to the lack of inlining of read and write barriers. We believe that this is the reason that in this case the refined TLE variants do not outperform TLE (despite the fewer failures to the lock), and that any reduction in the instrumentation overhead, for example, via inlining and compiler optimizations will significantly improve the performance of the refined TLE solutions.

7. Discussion

In this paper we introduced RW-TLE and FG-TLE, two approaches that refine TLE to improve the potential parallelism it offers. The RW-TLE and FG-TLE algorithms allow hardware transactions to execute a critical section on the instrumented path while a thread is holding the lock, without bearing the cost of hybrid TM systems that use STM as the fall-back method; in particular, RW-TLE only requires trivial instrumentation of write instructions. The lower instrumentation cost of refined TLE is achieved by limiting the number of threads that can run in software-only mode to be only one. This relieves that thread from detecting conflicts with other threads running in software, and guarantees successful completion of its critical section execution in a single attempt.

Experiments conducted with the implementation of RW-TLE and FG-TLE show that both approaches significantly improve the performance of TLE on a range of workloads. In addition, refined TLE variants often outperform an efficient hybrid TM algorithm despite the higher parallelism in software supported by the latter, as they enable threads to succeed more frequently in the fast path.

Acknowledgments

We are grateful to Virendra Marathe for useful discussions of some of the ideas in this paper, and to our anonymous reviewers and our shepherd Milind Chabbi for suggestions to improve presentation.

References

[1] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *Proceedings of ACM PODC*, pages 212–221, 2014.

[2] Y. Afek, A. Matveev, O. R. Moll, and N. Shavit. Amalgamated lock-elision. In *Proceedings of DISC*, pages 309–324, 2015.

[3] J. H. Ahn. ccTSA: A Coverage-Centric Threaded Sequence Assembler. *PLoS ONE*, 7(6), June 2012. doi: 10.1371/journal.pone.0039232. URL `http://dx.doi.org/10.1371/journal.pone.0039232`.

[4] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *ACM Workshop on Transactional Computing (TRANSACT)*, 2014.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of ASPLOS*, pages 199–210, 2012.

[6] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, 2010.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 39–52, 2011.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of ASPLOS*, pages 336–346, 2006.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of ASPLOS*, pages 157–168, 2009.

[10] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical report, Sun Labs, 2009.

[11] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[12] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive integration of hardware and software lock elision techniques. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 188–197, 2014.

[13] N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 209–219, 2014.

[14] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 3–14, 2014.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of ACM OOPSLA*, pages 388–402, 2003.

[16] A. Kleen. TSX anti patterns in lock elision code, Mar. 2014. `https://software.intel.com/en-us/articles/tsx-anti-patterns-in-lock-elision-code`.

[17] A. Matveev and N. Shavit. Reduced hardware lock elision. In *Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[18] A. Matveev and N. Shavit. Reduced hardware NOREC: An opaque obstruction-free and privatizing HyTM. In *ACM Workshop on Transactional Computing (TRANSACT)*, 2014.

[19] A. Matveev and N. Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of ASPLOS*, pages 59–71, 2015.

[20] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[21] T. Nakaike, R. Odaira, M. Gaudet, M. Michael, and H. Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, 2001.

[23] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *ACM SPAA*, pages 53–64, 2011.

[24] W. Ruan and M. Spear. Hybrid transactional memory revisited. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 215–231, 2015.

[25] T. Wang. Integer hash function. `http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm`, 2007. Accessed: 2015-02-13.

[26] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.