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 with a set of micro-benchmarks based on common fundamental data structures, and with a wide range of workloads.

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 [12]. 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 synchonization, 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 [8, 10, 15]. 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, 6, 8, 10, 15]. 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., [3, 4, 11, 13]. 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 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. Like 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 of the fast path (using HTM, after probing

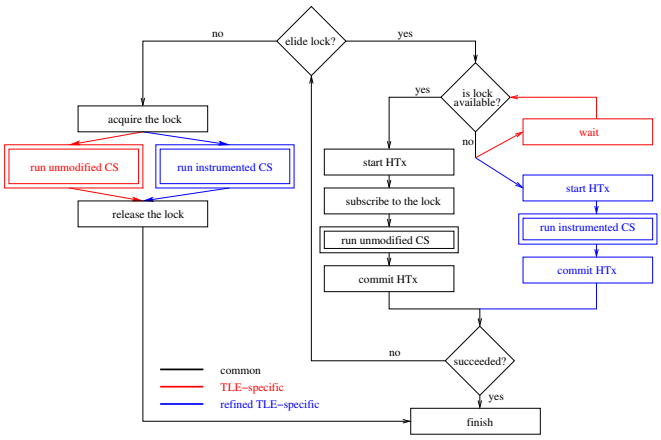


Figure 1: Design scheme for TLE and refined TLE

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 (light-weight) 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 through its builtin libitm library. In fact, our experiments reported in this paper are based on extensions to the libitm library. The GCC compiler allows to produce both unmodified and instrumented paths, while the libitm library allows to specify custom 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. 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 (or might even be destructive) when most transactions succeed on HTM (because the slow path would not be utilized) or when most of them fail to 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 fail to lock, which we believe are the workloads of interest in realistic applications. The actual characteristics of these workloads depend 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 and with a relatively small 4-core machine featuring HTM, we are able to present significant performance advantages of refined TLE over TLE in workloads that include several important data structures, such as AVL trees and skip-lists. In the future work, we plan to reduce the cost of instrumentation via inlining and evaluate the effectiveness of our scheme on a machine featuring HTM with larger thread counts.

2. Related Work

The original idea of TLE was presented by Rajwar and Goodman back in 2001 [12]. However, it 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 [6, 7]. 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 [8, 10, 15]. However, when some operations fail to complete on HTM, the scalability is severely hampered [1, 6, 8, 10, 15]. 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.

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., [3, 4, 11, 13]. 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 [11], 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 [11] builds on Hybrid NOREC [3] and presents Reduced NOREC that aims to reduce this overhead by introducing a small (aka reduced) hardware transaction into the software spec-

[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.

```
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

ulation 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. Apart from significant algorithmic simplicity of the refined TLE compared to Hybrid or Reduced NOREC, we believe the former will outperform the hybrid TM alternatives in cases where transactions fail to complete on hardware, yet the lock remains uncontended (and thus hybrid TMs will not benefit from allowing multiple threads to speculate using software). Based on our experience, we believe those are realistic and interesting cases to consider. Yet, the actual comparison of refined TLE with Hybrid and/or Reduced NOREC is left as a part of future work.

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.

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, including those that always have writes, is due to the *prefetching effect*. That is, even if a thread cannot complete the execution of the critical section using a hardware transaction on the slow path (e.g., because the execution requires a write), the partial execution attempt is often sufficient to warm up the cache for the next execution attempt on the fast path, making the latter faster and more likely to succeed. We demonstrate this phenomenon in Section 6.

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`) [4, 9] that captures information on the addresses that are accessed 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.

[2] More precisely, the update of the global clock is required for transactions that do 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.

```
/*********************************************
local_seq_number is the snapshot of the
epoch counter taken by each thread executing
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 [14].
*********************************************/

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             r_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

- 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.

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 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 a 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 hard-

ware 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.

We note, however, that one might still use refined TLE and cope with these issues by applying the *lazy subscription* optimization [3] 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, 2, 5, 10]. Note that while applying this technique to standard TLE is subject to numerous pitfalls [5], applying it to RW-TLE and FG-TLE is always safe due to the instrumentation of the slow path.

6. Performance Evaluation

We have implemented the refined TLE approaches in the libitm library. This library is a part of the open-source GCC distribution[3], and is intended to support transactional programs by providing several synchronization mechanisms, including standard TLE. We evaluated our implementation using a set of micro-benchmarks based on common fundamental data structures. In this paper, we show results from experiments with AVL trees and skip-lists implementing a set interface (supporting Insert, Remove and Find operations), and with skip-lists implementing a priority queue interface (supporting Insert and RemoveMin operations). These results are elaborated in Section 6.1 and Section 6.2, respectively.

We ran our experiments on a Haswell (Core i7-4770) 4-core hyper-threaded machine (8 hardware threads in total) running at 3.40GHz and powered by Oracle Linux 7. To reduce noise from the power management system, the machine was 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. Furthermore, before starting measurements, all threads were set to spin for a few seconds to allow the system to warm up.

```
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.

In our experiments, we varied the number of threads between 1 and 8. All threads were synchronized to start at the same time (after a warm-up period), and performed work for 5 seconds unless specified otherwise. During that time, each thread performed operations chosen uniformly and at random according to a given probability (e.g., 60% Find, 20% Insert and 20% Remove). The data structure used for a particular experiment was initialized with keys selected uniformly and at random as described in corresponding sections below. The key for each operation performed by each thread (e.g., Find in set) was also chosen uniformly and at random. At the end, each thread reported the total number of operations it had performed, and the total throughput was calculated. 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.

We compared our implementations of RW-TLE and FG-TLE with standard TLE and a lock-based synchronization. In the following figures, FG TLE (X) denotes the version that uses `r_orecs` and `w_orecs` of size X each. The other two synchronization techniques, standard TLE (denoted in figures and in the following simply as TLE) and a lock-based synchronization, are provided by libitm up to a few modifications described next.

For fair comparison, we made the following two material modifications to the TLE implementation of libitm. When a hardware transaction in Haswell HTM aborts, hardware provides a hint bit whether one should retry on HTM. The original implementation of TLE in libitm uses a policy that decides to retry on HTM only when the hint bit is set. We found out that in many our experiments, the transaction failure code returned by the hardware was 0, providing no meaningful information on the cause of the failure. In these cases, relying on this bit (i.e., avoiding retries on HTM) was not efficient. Therefore, we modified the TLE implementation of libitm to discard the hint bit, which resulted in overall better performance. We followed the same approach with RW-TLE and FG-TLE as well. In addition, we bumped up the number of retries on HTM before resorting to lock from two to five. We found in our experiments that a slightly higher number of attempts had a significant positive impact on TLE results. We used the same constant

[3] For our evaluation, we used GCC 4.9.0.

number of attempts for RW-TLE and FG-TLE. Thus, in TLE and in both refined TLE variants, the failed hardware transaction is retried up to five times before resorting to the lock, regardless of the hint bit. Note that for RW-TLE and FG-TLE, only attempts on the fast path are counted.

6.1 Experiments with sets

In this section, we present the results of our evaluation of two implementations of the set interface, one is based on AVL trees while another is based on skip-lists. We used sequential versions of trees and skip-lists based on publicly available reference implementations, converting each operation into an atomic block. In both versions, the memory for nodes inserted to (removed from) the set was allocated (deallocated, respectively) inside the atomic block. At the beginning of each experiment, the set was initialized with keys selected uniformly and at random from a given key range; the set was initialized to contain the number of keys equal to half of the range. Thus, by varying the key range we effectively controlled the initial size of the set.

Figure 5 presents results for AVL tree-based sets for various key ranges and mixes of operations as specified in captions. For instance, "8K, 60-20-20" means that the experiment was performed with the key range of 8K and the workload consisting of 60% Find, 20% Insert and 20% Remove operations. The results are normalized with the throughput of a lock-based single-threaded execution, representing relative speedup achieved by every approach.

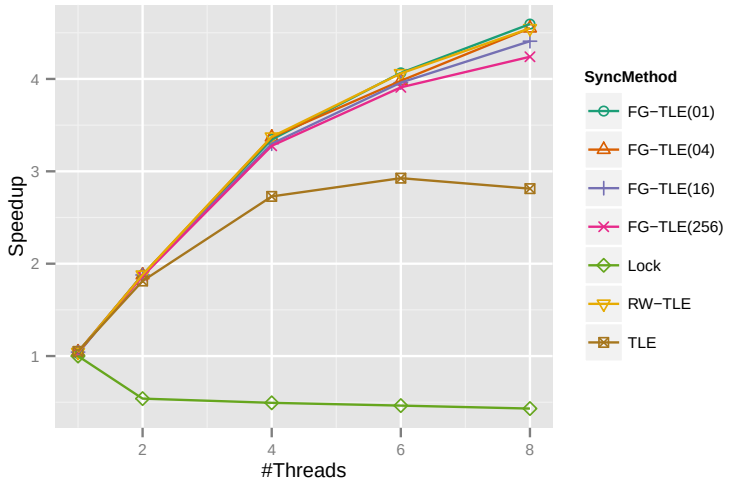
As expected, the benefits of the refined TLE show up when the workload includes update operations. This is because in read-only cases, the vast majority of all hardware transactions succeed at the first attempt and the lock-based path is not used. With update operations, some transactions fail to the lock due to conflicts on data they access (i.e., tree nodes that get modified). The number of conflicts and the benefit of refined TLE correlates positively with the number of update operations and correlates negatively with the size of the set. (The latter is because the smaller the set is, more conflicts are created on average by each update operation.) We note that even when a workload includes only Insert and Remove operations, only half of them, on average, actually update the corresponding set, while the other half skip the update since they do not find the key (in case of Remove) or since the key is already in the set (in case of Insert).

In addition to the speedup comparison between TLE and refined TLE, several observations can be drawn from the results in Figure 5. First, FG-TLE performed slightly better with a smaller size for `orecs` arrays, while the best performance was achieved by FG-TLE(1). We used statistics collected on the number of successful and failed attempts on fast and slow paths on HTM and lock, as well as timing information about executions under the lock[4], to shed some light on this phenomenon. Figure 6 depicts some of those statistics using as an example the experiment with the AVL tree-based set, the key range of 8K and the workload of 60% Find operations. Figure 6(a) shows the number of successful speculative executions on the slow path relatively to the number of executions under lock (that use the slow path as well) for refined TLE variants. Intuitively, the larger size of `orecs` arrays should allow more threads to succeed while speculating concurrently on the slow path. Indeed, there is a very rough correlation between the size of `orecs` arrays and the statistic shown in Figure 6(a).

However, we note that the larger size of `orecs` also means more overhead for executions under lock (including more memory fences). Figure 6(b) shows the time spent by executions under lock using various syntonization algorithms, normalized by the time spent under lock using lock-based synchronization and the same number of threads. The results for single thread runs are not shown, as TLE variants almost never fell to lock in these cases. Here we can see a clear correlation between the size of `orecs` arrays and the incurred overhead for various FG-TLE variants. As expected, RW-TLE that does not use `orecs` incurs the smallest overhead. However, even though the barriers in RW-TLE are particularly light-weight, an execution under the lock spends almost 3x more time in RW-TLE compared to that in TLE. Further investigation showed that this overhead comes mostly from the fact that GCC does not seem to support inlining of barriers used by libitm. In the future work, we plan to investigate this issue and drastically reduce the overhead of RW-TLE as well as of FG-TLE.

Figure 6(c) shows the relation between the two previous statistics, depicting the total number of successful executions on the slow path on HTM divided by the time spent by executions under the lock. In the following, we refer to this statistics as the *utility* measure. Figure 6(c) shows a rough negative correlation between utility and the size of `orecs`. In particular, even though FG-TLE(1) has lower number of successful executions on the slow HTM path per each execution under lock comparing to, e.g., FG-TLE(256) (as shown in Figure 6(a)), its lower overhead of executions under the lock (as shown in Figure 6(b)) results in a higher utility value. This explains, in part, the higher throughput results achieved by FG-TLE(1) over other refined TLE variants, as shown in Figure 5. Furthermore, we note that the utility measure in Figure 6(c) for most refined TLE variants increases with the number of threads as more threads can run in parallel on the slow path while the lock is held. This is exactly the design goal of the refined TLE, and we expect the utility and performance advantages of RW-TLE and FG-TLE to grow even further on larger machines.

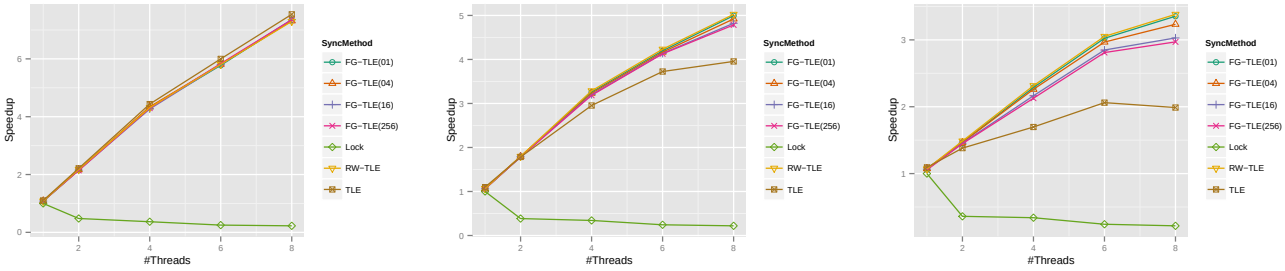
Our experiments with skip-lists-based sets show similar patterns to AVL trees-based sets, however the benefit of refined TLE over TLE is even higher. As an example, Figure 7 presents the results achieved with skip-lists-based sets with the range of 64K and no Find operations. We believe the higher benefit is because skip lists generally use larger nodes (as they might have multiple `next` pointers on different levels). This means a transaction might read a larger



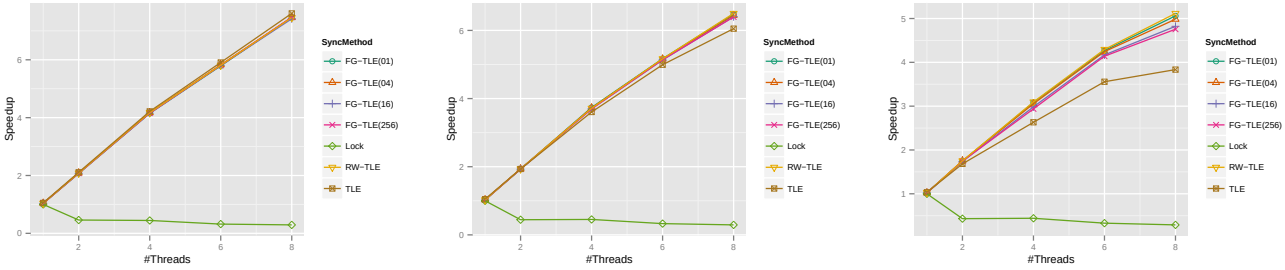
(a) 64K, 0-50-50

Figure 7: Skip-lists-based set throughput normalized with the throughput of a lock-based single-threaded execution.

[4] To reduce a probing effect, timing information was sampled randomly for less than 1% of executions under lock, on average.

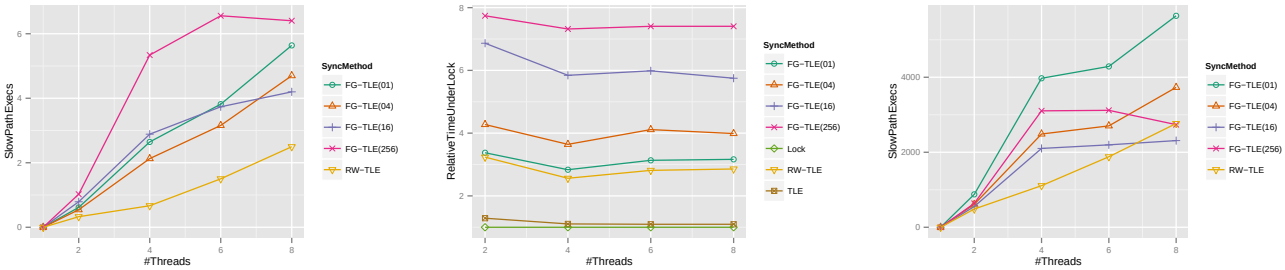


(a) 8K, 100-0-0 (b) 8K, 60-20-20 (c) 8K, 0-50-50



(d) 64K, 100-0-0 (e) 64K, 60-20-20 (f) 64K, 0-50-50

Figure 5: AVL tree-based set throughput normalized with the throughput of a lock-based single-threaded execution.



(a) Number of successful speculative executions on the slow path per one execution under lock

(b) Execution time under lock normalized to the time measured for the lock-based execution with the same number of threads

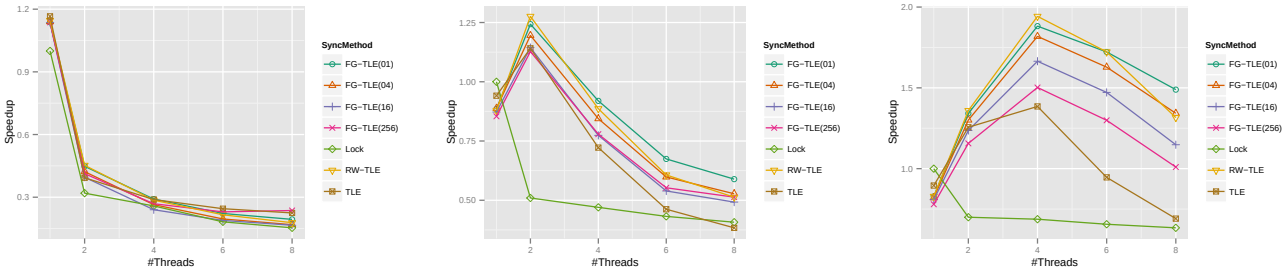
(c) Total number of successful speculative executions per time unit during which the lock was held by a thread running on the slow path

Figure 6: Performance statistics for the AVL tree-based set, 8K key range and 60-20-20 workload.

number of cache lines, therefore having a higher chance to experience aborts due to capacity limits of HTM. Having more capacity aborts helps refined TLE to exploit the benefits of prefetching as failed attempts on the slow HTM path warm the cache for subsequent attempts on the fast HTM path and increase the chance for the latter to succeed. (We discuss prefetching in more detail in the next section, as it plays even more crucial role in experiments with priority queues.) Figures 8 (a) and 8 (b) compare the distribution of HTM trials (including failed ones) made by TLE in experiments with the AVL tree-based set and the skip-list-based set, respectively, and the same workload as used in Figure 6. Supporting our intuition, they clearly show that the ratio of capacity aborts is much higher for skip lists.

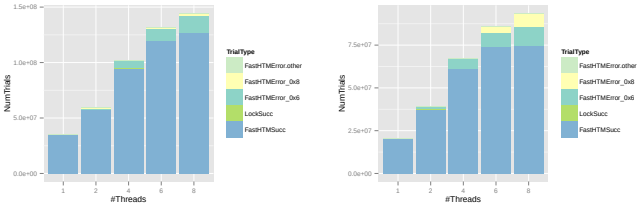
6.2 Experiments with priority queues

In our experiments with priority queues based on skip-lists, we used the full range of 32bits to select random keys for queue initialization and for subsequent operations performed by the varying number of threads. Figures 9(a) and (b) present results for experiments in which the queue is initialized with 100K and 1M keys, respectively, and then threads perform Insert and RemoveMin operations with probability 50% for each. As a result, the size of the queue is kept roughly the same throughout the whole experiment. We note that we allow the same key to be stored more than once in the priority queue, thus Insert operation is always successful in updating the queue.



(a) (b) (c)

Figure 9: Priority queue results for 50% Insert–50% RemoveMin experiments (a) and (b), and Insert-only experiments (c). For 50-50 experiments, the queue is initialized with 100K nodes (a) and 1M nodes (b). For Insert-only experiments (c), the queue is initially empty.



(a) 8K, 60-20-20 (b) 8K, 60-20-20

Figure 8: Distribution of execution attempts for TLE with AVL tree-based set (a) and skip-list-based set (b). FastHTMError_0x6 specifies speculative trials on the fast path that ended up with data conflict abort, while FastHTMError_0x8 specifies capacity aborts.

Figures 9(a) and (b) show that in general, TLE and refined TLE variants scale negatively in the evaluated data structure. This is not surprising as the RemoveMin operation is a bottleneck that causes many transactions to abort due to data conflicts. Yet, for larger queues (Figure 9(b)), TLE and refined TLE variants manage to scale for two threads and in general, speedups are substantially higher than for smaller queues (cf. Figure 9(a)). This is because Insert operations keep threads busy longer (as they need to search for the insertion point in a larger list) and thus reduce contention in RemoveMin. Besides, larger lists increase a chance that the insertion point of an Insert operation will be further from the head of the list, reducing the chance for contention between Insert and RemoveMin operations.

A few interesting phenomena are presented when comparing the performance of refined TLE with TLE in Figures 9(a) and (b). First, for smaller queues (Figure 9(a)), FG-TLE(256) performs better than other refined TLE alternatives. In fact, this is the only variant that manages to beat TLE at 8 threads. This suggests that in this workload, a lesser number of conflicts between threads speculating on the slow path and a thread holding the lock is more beneficial than the overhead created by a larger number of `orecs` used. We note that the distribution of trials on HTM (not presented due to lack of space) clearly shows that the number of self-aborts on the slow path due to `orecs` entries being updated by the thread under lock is significantly lower for FG-TLE(256) comparing to other FG-TLE variants.

On the other hand, the picture is quite different for larger queues (Figure 9(b)). There, all refined TLE variants beat TLE with more than two threads, while the performance of FG-TLE is better with a smaller size of `orecs`. The performance of RW-TLE is particularly remarkable given that none of the transactions succeed on the slow path (as they always perform a write). We believe this is an effect of prefetching made by futile transactions on the slow path, helping subsequent transactions on the fast path to succeed. This effect is apparent in larger queues where Insert operations need to access more cache lines, on average, and thus have a higher chance to experience a cache miss or abort due to capacity reasons. Refined TLE variants exploit the fact that, unlike TLE, when some thread is holding the lock, speculation can continue on the slow path, keeping the cache warm. This is particularly useful for RW-TLE, whose memory footprint overhead is negligible.

To estimate the effect of prefetching, we performed another set of experiments with only Insert operations. There, we start with an empty queue and then threads perform 1M Insert operations with randomly chosen keys; the operations are divided equally between all threads. We measure the time for the last thread to complete, and calculate throughput by dividing the total number of operations performed (i.e., 1M) by this measured time.

Results in Figure 9(c) show that TLE and all refined TLE variants scale up to 4 threads and then degrade as the number of conflicts between concurrent Insert operations increases. All refined TLE variants, including RW-TLE, beat TLE substantially with 4 threads or more, echoing the results in Figure 9(b). This gives evidence that performance benefits of refined TLE in Figure 9(b) are derived from faster Insert operations that enjoy prefetching made by speculative attempts on the slow path. In the future work, we aim to collect additional performance statistics to investigate further prefetching benefits of refined TLE variants.

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 in the slow path 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.

Initial experiments conducted using our implementation of RW-TLE and FG-TLE in the libitm library of GCC show that both

approaches significantly improve the performance of TLE on a range of micro-benchmarks and workloads. Interestingly, we found that RW-TLE improved performance even in cases where it did not complete *any* execution in HTM while the lock was held; the improvement is due to the ability of RW-TLE (and FG-TLE) to keep retrying the critical section and keep the cache warm.

In future work, we plan to experiment with an adaptive version of our algorithms based on the description in Section 4.2.1. Also, we note that we were able to show performance advantages of RW-TLE and FG-TLE despite the fact that our implementation could not benefit from inlining of read and write barriers in the slow path. The lack of inlining is a limitation of the current implementation of the GCC extension for transactional programs, and poses a major drawback for algorithms with lightweight barriers like RW-TLE and FG-TLE. In future work, we hope to demonstrate the effect of inlining of the barriers in the slow path, and compare this improved implementation of refined TLE with hybrid approaches that use STM as the fallback path.

Acknowledgment: We are grateful to Virendra Marathe for useful discussions of some of the ideas reflected in this paper.

References

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

[2] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Proceedings of 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[3] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 39–52, 2011.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

[5] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *Proceedings of 6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[6] 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.

[7] 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.

[8] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 3–14, 2014.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, 2003.

[10] A. Matveev and N. Shavit. Reduced hardware lock elision. In *Proceedings of 6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[11] A. Matveev and N. Shavit. Reduced hardware NOREC: An opaque obstruction-free and privatizing HyTM. In *Proceedings of 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2014.

[12] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, 2001.

[13] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 53–64, 2011.

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

[15] 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.