Practical Experience with Transactional Lock Elision

Tingzhe Zhou, PanteA Zardoshti, and Michael Spear
Lehigh University
{tiz214, paz215, spear}@cse.lehigh.edu

Abstract

Transactional Memory (TM) promises both to provide a scalable mechanism for synchronization in concurrent programs, and to offer ease-of-use benefits to programmers. To date, TM's biggest successes have been as a mechanism for achieving Transactional Lock Elision (TLE). In TLE, critical sections are attempted as transactions, with a fall-back to the original lock if conflicts manifest. Thus TLE expects to improve scalability, but not ease of programming. Still, until TLE can deliver performance improvements, transactional styles of programming are unlikely to gain popularity.

In this paper, we describe our experiences employing TLE in two real-world programs: the PBZip2 file compression tool, and the x265 video encoder/decoder. We discuss the obstacles we encountered, propose solutions to those obstacles, and introduce open challenges. In experiments using the GCC compiler's hardware and software support for TM, we observe that both are able to outperform the original lock-based code, potentially heralding the readiness of TM to be used more broadly in TLE, if not for truly transactional styles of programming.

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

Keywords Transactional Memory, Lock Elision, Synchronization, Privatization, Semantics, Two-Phase Locking

1. Introduction

There are two equally appealing programming models for Transactional Memory (TM) [9]. In the first, programmers endeavor to create new concurrent programs, or new concurrent modules within existing programs, and expect to use transactions as the primary mechanism for concurrency control. In this "transactions first" approach, every feature of the TM implementation, to include any support for self-abort [8, 17], deferred actions [25], or nuanced contention management [20], is an input to the programmer's design process. It is taken for granted that the program's use of TM will result in good scalability. In the second model, a programmer takes as input a lock-based program with less-than-desirable performance, and replaces locks with TM, hoping that in so doing, unnecessary serialization can be avoided. The programmer in this case thinks of TM as a mechanism for achieving lock elision, and is encouraged to ignore advanced features of the TM implementation.

The current draft of the C++ TM Technical Specification (TMTS) [11] exposes two constructs for lexically-scoped transactions, which correspond to these two uses: `atomic` blocks and `synchronized` blocks.[1] The TMTS allows both types of blocks to execute simultaneously, and specifies that either hardware or a software runtime will track conflicts among both types of transactions to ensure that any transactional execution has an equivalent serializable, sequential history.

Despite five years of support for TM in the GCC compiler and mainstream CPUs, and two years of experience with the C++ TMTS, TM adoption remains limited. This is particularly concerning given the availability of `synchronized` blocks: if individual locks can be elided, one at a time, without requiring a whole-program rewrite, then certainly programmers should be able to find opportunities to employ TM and improve scalability. While the use of `synchronized` and `atomic` blocks seems to be decoupled, there is a natural dependency that programmers may perceive: if lock elision is not profitable, then full-blown transactional programming is unlikely to result in programs that scale better than they would with locks. In effect, any future adoption of TM depends on success stories with transactional lock elision.

Part of the problem faced when demonstrating the effectiveness of lock elision is the significance of the input program. The most compelling examples of successful lock elision will improve the performance of a program that is widely used. However, such programs are likely to be highly optimized, with carefully crafted lock protocols that already avoid contention and ensure good scalability. To elide locks in anything less significant is to create a false comparison, where the baseline is either unimportant or unoptimized. To date, efforts to improve production-quality programs via TM have had limited success [19], or have required low-level reasoning about a specific hardware TM (HTM) implementation [12].

In this paper, we employ the C++ TMTS to transactionalize two programs of significant size and realism: the PBZIP2 file compression benchmark, and the x265 media encoder/decoder. Both programs are large, robust, and mature. Both transactionalizations adhere to the C++ TMTS, resulting in programs that can be connected to any TM implementation. In both cases, we find that the use of TM for lock elision improves performance, with HTM showing gains of up to 9%. Along the way, we identify several gaps in the quality of existing tools and libraries for transactional programming, we observe obstacles unique to transactional lock elision, and we propose solutions that affect HTM, software TM (STM), or both.

The remainder of this paper is organized as follows. In Section 2, we describe the high-level behavior of the two applications upon which this paper focuses. Section 3 discusses the quiescence mechanism used to ensure lock-based semantics, and presents situations in which quiescence overheads are avoidable. Section 4 discusses problematic lock-based code in x265, which is not immediately transactionalizable. Section 5 briefly discusses additional considerations these applications present, and describes our workarounds. Section 6 presents performance results for the two applications, and shows that with modest effort, HTM is able to outperform the original lock-based code. Section 7 concludes.

[1] In prior versions of the technical specification, these were referred to as "atomic" and "relaxed" transactions, respectively.

2. PBZip2 and x265

Our study focuses on the transactional elision of locks in two programs, as described below.

PBZip2 PBZip2 is the parallel version of the bzip2 file compression algorithm. Whereas the original bzip2 algorithm takes as input a complete file stream, and then compresses it, PBZip2 splits a file into multiple streams, and compresses those streams in parallel. The user is able to specify the size of each stream, to balance the amount of work per thread with the number of threads. Internally, the program generally follows a serial-parallel-serial pipeline pattern. A producer thread creates stream descriptors, and passes them to consumer threads. Consumer threads compress or decompress streams, based on the descriptors they pull from the queue. Their output is passed to the final stage, a serial write stage, which produces the output file by assembling its input in the correct order.

The implementation employs six locks and six condition variables. The critical sections are friendly to transactionalization, in that they do not make system calls, and are small. In particular, the compression and decompression operations are performed outside of critical sections. The main source of contention is for the locks protecting the inter-stage queues.

x265 x265 is a video encoder application capable of encoding video streams or images into the HEVC/H265 compression format. The encoding algorithm divides each frame into sequences of macro-blocks called "slices", which are passed to decoder threads. Each slice consists of a sequence of CTUs (Coding Tree Units), which can be encoded by making reference to another unit in the same frame (intra-picture prediction) or in another frame (inter-picture prediction). The output frame is stored in a decoded frame buffer to be used for the prediction of other frames.

x265 takes advantage of as much parallelism as possible to improve performance. With frame-level parallelism, independent frames can be encoded simultaneously. Each video frame is also divided into slides that can be independently processed. In CTURow-level parallelism, a wavefront parallelization algorithm processes an individual frame. Within each CTU, CUs (Coding Units) distribute their analysis work to threads, which provide CU-level parallelism. At the lowest level, vector instructions can be enabled to process adjacent pixels of a frame. To manage parallelism more abstractly, x265 includes wrappers over traditional synchronization objects. These include a thread pool and a condition variable wrapper, as well as a wrapper around mutex locks. A depiction of the wavefront appears in Figure 1.

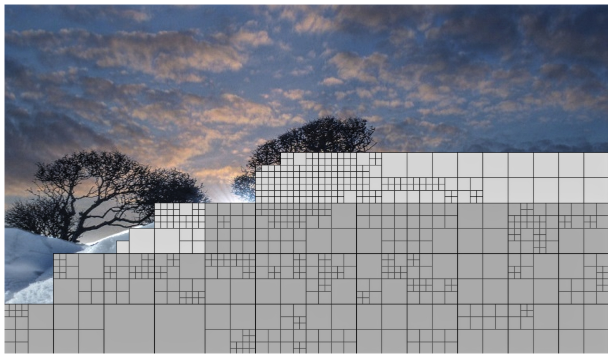


Figure 1: HEVC wavefront parallel processing [18].

There are three main lock objects in x265:

- Lookahead Lock: This lock prevents concurrent access to shared input and output queues of frames; in essence, it mediates inter-frame parallelism.
- CTURows Lock: This lock is used by the wavefront processing algorithm, to mediate communication from completed CTUs to the CTUs that depend on it.
- EncoderRow Lock: This lock protects shared data when multiple threads work on the same row within a slice of a frame.

There are additional locks, to include the "bonded task group" lock, which governs the allocation of jobs to threads; a "parallel motion estimation" lock, which protects searches for reference frames during motion searches; and a "cost lock", which protects metadata and metrics maintained by the threads.

3. Quiescence and Lock Elision

The Java programming language places constraints on the behavior of programs, even when those programs are not correctly synchronized [15]. As a result, formalizations of transactional semantics for Java that were expressed in terms of locks [16] relied on quiescence to ensure ordering between transactions, even when those transactions did not access *any* shared memory. Quiescence ensures that whenever a thread commits a transaction, it waits until all concurrent threads commit or abort *and clean up* before the thread is permitted to execute the code that follows the transaction. While some STM algorithms have quiescence support built-in [2, 4, 14, 22], the STM algorithm in GCC does not, and requires a committing transaction to execute code similar in spirit to a userspace RCU Epoch [3].

In contrast, the C++ language does not define the behavior of racy programs, and the C++ TMTS does not define the semantics of transactions in terms of locks. As a result, we can use the notion of transactional sequential consistency [1] to describe a C++ memory model that includes locks, `atomic` variables, and transactions. The memory model requires a global total order on synchronization operations, program order on synchronization operations within a thread, and a global total order on all transactions. The model does not handle explicit self-abort within transactions, which Shpeisman et al. previously showed to be a source of significant additional complexity [21].

From a practical perspective, the main role of quiescence is to ensure that when a transaction transitions data from a shared state to thread-private, or vice versa, concurrent transactions accessing that data do not race with legal nontransactional accesses. In HTM, such accesses are not possible; in STM, they can result due to delayed undo or write-back operations. Since publication safety (i.e., ensuring the absence of races when transitioning data to a state in which it can be accessed by transactions) is assured by data and control-flow dependencies in the source code, quiescence in C++ is only required for privatization safety (i.e., ensuring the absence of races when transitioning data to a state in which it is no longer accessed by transactions) among STM transactions.

When the C++ TMTS is used to achieve lock elision, two problems emerge. First, lock elision is achieved via a form of lock erasure: whereas the original program may contain many locks, which protect disjoint regions of memory, all elided locks become transactions over a single shared heap. As an example, if a program contained a queue protected by lock L_1, and a stack protected by lock L_2, the transactional version would contain one class of transactions used to protect both the queue and the stack. As a global synchronization operation, quiescence forces a transaction on the stack to delay after it commits, waiting until any concurrent transaction touching the queue *or* the stack has committed or aborted. Since the locks are erased during TMTS-based transactional lock elision, the granularity of quiescence becomes unnecessarily coarse.

The second problem is that quiescence has the potential to result in transaction congestion. Consider two transactions, T_1 and T_2, each of which takes U units of time to complete. Suppose that T_1 begins at time 0, and T_2 begins at time $U/2$. When T_1 completes,

Listing 1: Proxy privatization

```
// Vector update thread
atomic_cancel
    for $k \in 0 \ldots size$ do
        $update(vec[k])$

// Privatizer thread
atomic_cancel
    $msg \leftarrow vec$
    $vec \leftarrow$ null

// Proxy thread
atomic_cancel
    if $msg =$ null then
        retry
$use(msg)$
```

Listing 2: Producer/consumer workload

```
// Producer thread
while true do
    atomic_cancel
        if $\neg c.full()$ then
            $c.insert(produce())$
        $TM.NoQuiesce()$

// Consumer thread
while true do
    atomic_cancel
        if $\neg c.empty()$ then
            $tmp \leftarrow c.get()$
        else
            $tmp \leftarrow$ nil
            $TM.NoQuiesce()$
    if $tmp \neq$ nil then
        $use(tmp)$
```

it must wait for T_2 to commit or abort. This waiting does not increase the likelihood of T_2 aborting, because T_1 has already committed. However, if T_1 and T_2 execute in tight loops, then after one iteration, the interval between when the next T_1 and next T_2 begin is likely to be less than $U/2$. Quiescence in T_1 results in future congestion.

While HTM does not incur quiescence overheads, STM must. Furthermore, these overheads are growing increasingly expensive. Prior to 2016, GCC's STM implementation performed quiescence after every writing transaction. This, however, does not support proxy privatization (see, e.g., Listing 1). Since 2016, *every* transaction quiesces after committing. Clearly, this maximal use of quiescence prevents privatization-induced races.

3.1 Programatically Avoiding Quiescence

Alternatives to maximal quiescence draw from the observation that privatizing in C++ always involves at least one transaction. Two sufficient criteria arise: (a) require quiescence in the transaction that transitions the data to a nontransactional state, or (b) require quiescence in the last transaction executed by a thread before it accesses data nontransactionally. In practice, neither approach is straightforward: When transactions are nested, or have complex memory access patterns, it is not feasible to expect programmers to know which transactions privatize. Indeed, some transactions might only privatize under certain circumstances (consider a consumer who reads from a producer/consumer queue: if the queue is empty, there is no data to privatize). Furthermore, proxy privatization (itself a reasonable idiom, e.g., for a producer/consumer workload with per-consumer queues) cannot support marking privatizing transactions (criteria "a" above) without added overhead.[2]

In the proxy privatization case, a writer privatizes the data, and then a transaction in another thread executes before the data is accessed nontransactionally. In the non-proxy privatization case, a writer is the last transaction to modify the data before nontransactional access. In both cases, it is easier to achieve the second sufficient criteria: we could mark the last transaction before the private access.

With complex control flows, nested transactions, and separate compilation, we do not believe that programmers will be able to correctly identify the minimal set of transactions that require quiescence. However, one simple heuristic can capture a fair portion of the times when quiescence is not needed: if transactions T_1 and T_2 are executed sequentially by the same thread, then T_1 requires quiescence only if the thread's memory accesses between T_1 and T_2 might include data that was accessible by transactions prior to T_1's execution.

Prior work by Yoo et al. [24] suggests that in some workloads, quiescence can be disabled for *all* transactions. Yoo et al. also showed that in such cases, disabling quiescence for those workloads had a significant improvement on performance. Unfortunately, such an approach is not compositional: any change to the program requires whole-program analysis to determine if globally disabling quiescence remains correct. It also fails when *few* transactions privatize.

We propose a new TM API function: `TM.NoQuiesce`. When called within a transaction, this function indicates that the transaction should not quiesce after it commits. The call has no meaning for strongly isolated HTM implementations, or for STM implementations that do not require quiescence. The STM implementation is also free to ignore the API call. Two examples are when the transaction making the call is nested within another transaction, in which case its programmer is unlikely to know the privatization behavior of the parent transaction, and when the transaction `frees` memory (certain TM-aware memory managers require quiescence before returning memory to the operating system [10]). Furthermore, since the call can be made conditionally, it avoids overhead in the above case, where a consumer encounters an empty queue.

Listing 2 demonstrates the use of `TM.NoQuiesce`. The producer need never quiesce, since it never privatizes data, and the consumer need only quiesce if it succeeds in extracting an element from the collection (c). This example offers additional benefits: for single-producer, multi-consumer workloads, the producer is more likely to be the bottleneck, and avoids quiescence. Furthermore, when a consumer finds no work, it does not wait unnecessarily before looking again.

3.2 Pitfalls

`TM.NoQuiesce` has the potential to significantly increase scalability: quiescence can entail cache misses linear in the number of threads, to observe their current state and determine when they are no longer at risk of racing with a subsequent nontransactional access; and long-running transactions can lead to a quiescence operation blocking unrelated threads' committed transactions for the duration of the long-running operation. However, when used incorrectly, `TM.NoQuiesce` transforms an otherwise correct program into a racy program.

The problem is that Transactional Sequential Consistency demands a global total order among transactions, and the transitive closure of transaction order and program order must establish happens-before relations. Quiescence delays committing transactions long enough to be certain of transitivity with program order across threads. In contrast, `TM.NoQuiesce` asserts that data and/or control-flow dependencies within a specific transaction, or among specific dynamic instances of transactions within the thread, are enough to provide happens-before. When the assertion is faulty, the program becomes erroneous because there are accesses to shared memory that may not be compatible with any global total order on transactions. We expect these errors to be easy to identify and fix using transactional race detectors. For example, T-Rex [13] is able to identify all races that arise when a TM library fails to provide privatization safety. Extending T-Rex to understand

[2] The issue is that existing STM algorithms would require writing transactions to quiesce *before* releasing ownership of locations.

Listing 3: Example of non-serializable critical section in x265.

```
// LOCK for output queue(i)
$OutputQueue.lock()$
$element = newQueueNode()$
$OutputQueue.enqueue(element)$
 $process(element)$
$OutputQueue.unlock()$

// LOCK for output queue(ii)
$OutputQueue.lock()$
$OutputQueue.dequeue()$
$OutputQueue.unlock()$
```

```
// Process(element)
process(element)
    $m\_lock.lock()$
    $m\_task = element.size()$
    $m\_lock.unlock()$
    $sub\_working()$
    $Wait()$
    $return$

// Task for working threads
sub_working()
    $...$
    $m\_lock.lock()$
    $m\_task - -$
    $m\_lock.unlock()$
    $...$
```

Listing 4: A `ready` flag avoids lock nesting, facilitating transactionalization.

```
// LOCK for output queue(i)
$OutputQueue.lock()$
$element = newQueueNode()$
$OutputQueue.enqueue(element)$
$element.ready = false$
$OutputQueue.unlock()$
$process(element)$
$OutputQueue.lock()$
$element.ready = true$
$OutputQueue.unlock()$

// LOCK for output queue(ii)
$OutputQueue.lock()$
if OutputQueue.peek().ready
    $element = OutputQueue.dequeue()$
$OutputQueue.unlock()$
```

implicitly privatization-safe STM with selective disabling of privatization appears to be straightforward.

4. Two Phase Locking and x265

Part of the appeal of TM is that it ought to be *easier* than using fine grained locks. By extension, it ought to be easy to employ TM for lock elision: the programmer need only replace each lock-based critical section with a transaction. Past work has revealed this task to be laborious, but not thought-intensive. For example, in transactional memcached [19], the effort was in identifying which transactions caused unnecessary serialization, and then creating transaction-safe variants of the standard library functions that were responsible for the serialization.

In memcached, critical sections obeyed two-phase locking [6], by ensuring that all lock acquires preceded all lock releases within each critical section. The only complication was when a critical section also read a C++ `atomic` variable. Our solution was to model these as mini-transactions, and subsume them within the transaction that replaced the critical section. In memcached, critical section behavior did not depend on an atomic variable changing between accesses, and hence this was safe.

The transactionalization of PBZip2 did not result in any new lessons about how to transactionalize code: lock-based critical sections were replaced with transactions, and as appropriate, functions were annotated to ensure transaction safety. Transactions did not contain complex behavior or control flow. However, during the transactionalization of x265, we found a situation in which the pattern of lock acquisitions and releases was clearly not two-phase locking, and hence the program could not be naively transactionalized: if the outer lock was replaced with a transaction, the program would hang.

Fortunately, the violation of two-phase locking in x265 was fixable. The specific behavior was that a producer thread would acquire a lock on its output queue, then produce elements, then place the elements in the queue and unlock the queue (Listing 3). During element production, several smaller critical sections ran, with inter-thread communication between the critical sections. These critical sections could not be subsumed by the transaction on the output queue. Our solution was to embed a ready flag in each queue node, rather than keep the queue locked for the duration of the program (Listing 4). Across several workloads and thread counts, we confirmed that this modification did not affect performance. With the change in place, each of the critical sections during element production could be separately transactionalized.

Our experience introduces two research questions, which we leave as future work:

- Can it be proven that naive transactionalization is safe for critical sections that obey two-phase locking?
- Under what conditions will naive transactionalization of non-two-phase locking code remain safe?

5. Additional Considerations

Library and compiler support for the TMTS remains inconsistent, and we encountered three categories of code that caused transactions to serialize unnecessarily or behave incorrectly. For completeness, we discuss each problem and its resolution below.

Logging Overheads In TM versions of memcached and Atomic Quake [26], critical sections occasionally perform logging operations, such as error messages and diagnostic writes to per-thread logs. The program does not require any ordering among logging operations: they are timestamped, the order can be determined post-mortem, and the return values of any syscalls during logging are ignored. Consequently, these operations can either be executed unsafely (and possibly more than once) by STM, or deferred until the end of the transaction. Since unsafe execution would still lead HTM to serialize, we chose to defer the logging operations, using the Mimir method [25].

Conditional Synchronization In order to support its soft real-time guarantees, x265 uses timeouts whenever a thread waits on a condition variable. To support this behavior, we first refactored the relevant critical sections to be compatible with Wang's transaction-safe condition variable library [23]. However, the library did not support timeouts. Our extension makes use of timed wait operations in POSIX semaphores, and was verified to have no impact on the behavior of the original lock-based program.

Vector Instructions Lastly, x265 can be configured to make use of vector instructions (e.g., Intel SSE) during rendering. In all, there are over 50 distinct SSE instruction types used by the program, all of which cause STM implementations to serialize. By analyzing each SSE call, we were able to determine that the compiler correctly instrumented SSE memory accesses, at which point the remaining SSE arithmetic operations did not require instrumentation. Our solution was to use the (deprecated) `transaction_pure` annotation to prevent these operations from causing the compiler to insert serializing instructions. However, this is not a satisfactory long-term approach.

6. Evaluation

In this section, we evaluate the use of TMTS-based transactional lock elision in PBZip2 and x265. As much as possible, we preserved the original structure of the source code. The only exceptions are (1) our "ready" flag in x265, which allowed us to transform the code to adhere to two-phase locking, (2) the addition of `TM.NoQuiesce` calls, and (3) minor refactorings of transactions that wait as part of condition synchronization. We use Wang's transaction-friendly condition variables [23], but these require waiting transactions to be enclosed in a loop, and rewritten so that a waiting transaction always performs its wait as its last instruction. Since the TMTS does not officially support these condition variables, we also considered the use of this refactored code *without* conditional waiting, in which case threads repeatedly poll their wait condition within a small transaction.

All experiments were conducted on a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40 GHz. This CPU supports Intel's TSX extensions for HTM, includes 8 GB of RAM, and runs a Linux 4.3 kernel. We used the GCC 5.3.1 compiler, and only modified its TM implementations enough to support transaction-friendly condition variables (the default HTM implementation does not, due to a lack of support for deferred actions). Results are the average of 5 trials. The STM results use ml_wt algorithm (a privatization-safe version of TinySTM [7]. The HTM results fall back to a serial mode after hardware transactions fail twice.

6.1 PBZip2

PBZip2 offers two independent operations, Compress and Decompress. We test both, using a 650MB test file. Within PBZip2, we are able to dynamically control the number of worker threads, as well as the size of the blocks that are processed in parallel; all other configuration parameters are set to their defaults. In our experiments, we vary the number of worker threads from 1 to 8, and consider block sizes of 100K, 300K, and 900K (the range is 100K to 900K, with 900K being the default). Apart from the worker threads, there is a main thread, which runs the benchmark harness but does not participate in the work.

We compare five algorithms. The baseline is the original code, which uses pthread mutex locks. We then consider three STM algorithms: STM + Spin uses the ml_wt algorithm, and uses spin waiting when the baseline would wait on a condition variable. STM + CondVar uses transaction-friendly conditional variables. STM + CondVar + NoQuiesce adds dynamic disabling of quiescence for selected transactions. Lastly, the HTM + CondVar executes the transaction using GCC's HTM support.

The main use of critical section in PBZip2 is to protect queue metadata. Therefore, the average size of critical sections is small. Each thread can access the metadata after it finishes compressing/decompressing its block. Conflicts among critical sections are rare: for a 650MB test file, we observe between 950 and 1100 transactions, of which 0.1% abort at least once in STM. In the HTM experiments, 13% to 18% of transactions abort twice and fall back to serial mode. Since current HTM support does not report the size of the working set on transaction abort, it would be beneficial for programmers to be able to suggest retry policies on a transaction-by-transaction basis: for queues that are expected to be un-contended, more retries before serialization might be appropriate.

Figure 2 shows the performance of the TM algorithms on PBZip2. STM + Spin performs the worst in all conditions except for Figure 2d. This is because spinning not only wastes threading resources, but also increases contention between caches. In Figure 2d HTM + CondVar performs worse than STM + Spin in some cases because nearly 20% of HTM transactions fall back to the serial path. Note, however, that STM + CondVar and STM + CondVar + NoQuiesce both outperform the baseline in Figures 2a and 2f, at least for high thread count. At low threads, conflicts are rare, and STM instrumentation overheads dominate.

Disabling quiescence offers mixed results. There is, necessarily, extra tracking and instrumentation overhead, which much be offset. In Figures 2d and 2e, disabling quiescence offers the best performance at high concurrency levels, which correspond to the scenarios in which the most gain is expected. Note, too, that HTM + CondVar often outperforms the baseline, achieving a peak speedup of 8.5% in Figure 2a. In this case, the fallback rate remains high (15%-18%), suggesting that finely tuning fallback strategies would offer even better performance.

6.2 x265

To evaluate x265, we consider three file sizes: small (38MB), medium (735MB), and a real movie downloaded from Netflix (3810M). The application defaults to a pool of 8 worker threads, 3 frame threads, and a main thread. In our experiments, we vary the number of worker threads, and again consider the five algorithms from above. The impact of spinning is disastrous in this workload, even at low thread counts. To maintain readability in Figure 3, we plot speedup relative to the single-thread pthread execution, instead of execution time.

The peak performance of HTM is 9.5% better than pthreads at 4 threads (Figure 3b). Moreover, HTM outperforms pthreads in almost every case. Again, this is with untuned GCC HTM support: the abort rates in Figure 4 suggest that even better performance is possible by tuning the fallback policy.

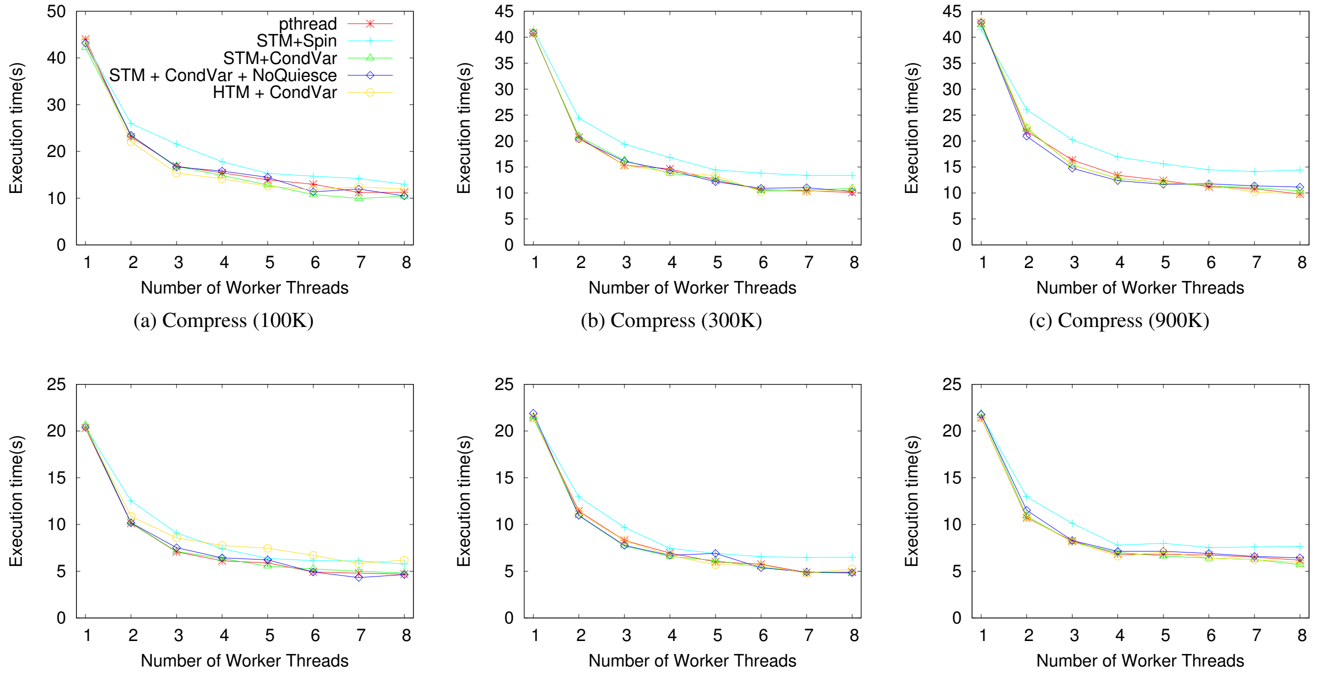
Again, disabling quiescence did not have a significant or consistent impact on performance. In the worst cases, it even decreases performance relative to STM + CondVar. In Figure 4, we find that disabling quiescence results in higher abort rates for the STM execution. As discussed in Section 3, quiescence is leading to transaction start times becoming bursty, where they would otherwise follow a more normal distribution over time.

Across all experiments, we observe many situations in which STM + CondVar and HTM outperform the pthread baseline. This result is in spite of the lock erasure effects of TMTS-based transactional lock elision. However, we required support for conditional synchronization, which is currently lacking in the TMTS. We were particularly surprised by the performance of STM; its overheads at transaction boundaries, and on every access of shared memory, were still less than the gain in performance. A variety of optimizations could take this result even further, such as reducing latency for small transactions [5], making quiescence avoidance conditional on the number of threads (to reduce bookkeeping costs at low thread counts), and moving code out of the critical section via atomic deferral [25].

7. Conclusions and Future Work

In this paper, we applied the C++ TMTS to elide locks in two real-world programs, PBZip2 and x265. In both cases, the programs were already carefully crafted to avoid lock contention and to scale. Nonetheless, transactional lock elision improved performance by up to 9%. To the best of our knowledge, this is the first example of the TMTS, as implemented in the GCC compiler, improving the performance of real-world code. Moreover, the improvement spanned both hardware and software implementations of TM.

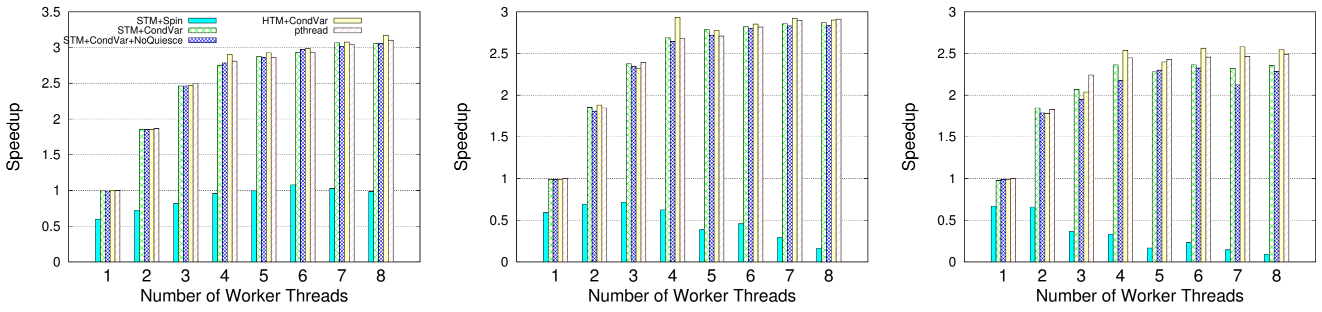
Unfortunately, our experience does not validate the expectation that transactional lock elision will be easy. In x265, the most important critical section was not serializable, and we could not transactionalize it without understanding several thousand lines of code, and changing the way in which threads interacted with one of the central queues in the program. There is exciting future work in this area, exploring the conditions under which an unmodified critical section can and cannot be transactionalized. Our intuition is that



(a) Compress (100K) (b) Compress (300K) (c) Compress (900K)

(d) Decompress (100K) (e) Decompress (300K) (f) Decompress (900K)

Figure 2: Performance of Transactionalized PBZip2



(a) Netflix film (3810M) (b) Medium test file (735M) (c) Small test file (38M)

Figure 3: Performance of Transactionalized x265

two-phase locking is a sufficient condition, but a more formal study is needed.

We also showed that quiescence avoidance need not be thought of as an all-or-nothing proposition. Specifically, TMTS-based lock elision introduces orderings that a fine-grained locking program would not display. Allowing programmers to avoid these overheads, without sacrificing composability, will help STM executions of a program. However, our experiments show that quiescence avoidance is a delicate proposition, and does not unilaterally improve performance.

Lastly, our experience suggests that much more work is needed before programmers can use TM easily. Library support remains inconsistent, and even a fully-implemented specification is insufficient to address third-party libraries, such as the vector math library used by x265. We encourage continued effort in this direction. We have shown that TMTS-based lock elision can produce performance gains, and thus that further investment in transaction-safe libraries will have long-term value.

Acknowledgments

We thank Victor Luchangco and Dave Dice for their advice and guidance during the conduct of this research. This material is based upon work supported by the National Science Foundation under Grant CAREER-1253362, and through financial support from Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

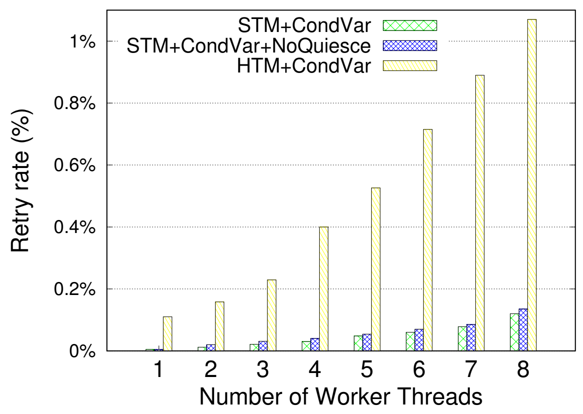


Figure 4: Retry rate for Netflix test (1̃0,000,000 Transaction Commits)

References

[1] L. Dalessandro and M. Scott. Strong Isolation is a Weak Idea. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[2] L. Dalessandro, M. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[3] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[4] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[5] A. Dragojevic and T. Harris. STM in the Small: Trading Generality for Performance in Software Transactional Memory. In *Proceedings of the EuroSys2012 Conference*, Bern, Switzerland, Apr. 2012.

[6] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[7] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[8] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[9] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[10] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[11] ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory, May 2015.

[12] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*, Orlando, FL, Feb. 2014.

[13] G. Kestor, O. Unsal, A. Cristal, and S. Tasiran. T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications. In *Proceedings of the EuroSys2014 Conference*, Amsterdam, The Netherlands, Apr. 2014.

[14] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a Scalable Software Transactional Memory. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[15] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, Long Beach, CA, Jan. 2005.

[16] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[17] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[18] Parabola Research. HEVC Wavefront Animation, Dec. 2013. https://www.parabolaresearch.com/blog/2013-12-01-hevc-wavefront-animation.html.

[19] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[20] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[21] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards Transactional Memory Semantics for C++. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[22] M. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[23] C. Wang, Y. Liu, and M. Spear. Transaction-Friendly Condition Variables. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, June 2014.

[24] R. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[25] T. Zhou and M. Spear. The Mimir Approach to Transactional Output. In *Proceedings of the 11th ACM SIGPLAN Workshop on Transactional Computing*, Barcelona, Spain, Mar. 2016.

[26] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.