Practical Experience with Transactional Lock Elision*

Tingzhe Zhou, PanteA Zardoshti and Michael Spear
Department of Computer Science and Engineering
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. The most straightforward use of TM in real-world programs is in the form of Transactional Lock Elision (TLE). In TLE, critical sections are attempted as transactions, with a fall-back to a 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 for TLE, if not for truly transactional styles of programming.

I. INTRODUCTION

Transactional Memory (TM) [1] was first proposed more than two decades ago, as a hardware mechanism for simplifying the creation of concurrent data structures. Subsequent research has considered expanding the role of TM to a full-fledged programming model, in which programmers use coarse-grained transactions as the primary means of synchronizing threads [2], and to using transactional techniques to invisibly improve the scalability of lock-based programs [3]. This latter idea, often called Transactional Lock Elision (TLE), makes use of hardware and/or software instrumentation to replace lock-based critical sections with transactions.

Despite significant interest and effort from the research community, TM has not yet seen widespread use in industry. The lack of adoption is particularly surprising given that (a) vendors have supported hardware TM (HTM) in commercially-available processors since 2012 [4], (b) compiler support for software TM (STM) has been present in the GCC compiler since 2012 [5], and (c) a Technical Specification for using TM in C++ programs (the TMTS) was announced in 2015 [6].

In this paper, we focus on TLE. TLE plays an important role in the long-term adoption of TM: until it is shown to be useful, there is little incentive for HTM and STM implementations to provide support for the advanced features needed by transactional programming models, such as self-abort [7], [8], deferred actions [9], OS support for transactional system calls [10], escape actions and open nesting [11]–[13], and nuanced contention management [14].

The primary vehicles for our study are two open-source applications: the PBZip2 parallel file compression/decompression toolkit [15], and the x265 media encoder/decoder [16]. Both programs are large, robust, and mature, with many locks, non-trivial input and output operations, and subtle protocols for sharing memory. Neither was designed with TM in mind, and both have been heavily optimized. In both cases, we explore whether it is possible for TLE to improve program performance. Our TLE versions of these benchmarks were produced via a hand instrumentation that applied the C++ TMTS. In this manner, we believe our findings have the most potential to generalize, both to other TLE endeavors, and to efforts to improve the TMTS. Since it conforms to the TMTS, it is also relatively easy to test these benchmarks with a variety of HTM and STM implementations.

While there are many small discoveries reported in this paper, two experiences stand out. In the case of PBZip2, we found that both HTM and STM were able to improve performance relative to locks, but STM could only do so when the C++ TMTS was extended with a mechanism for relaxing the ordering guarantees on certain transactions. We propose a dynamic mechanism for achieving this relaxation, which is reminiscent of the memory_order_relaxed feature of the C++ memory model [17]. In x265, we found that the most significant critical section in the program did not obey two-phase locking, and was incompatible with TLE. A simple refactoring solved the problem, but raises an important question: is two-phase locking a necessary or sufficient condition when using TLE? As with PBZip2, both HTM and STM were able to improve the performance of x265, relative to the original lock-based program. Our peak improvement was 9%.

The remainder of this paper is organized as follows. We begin with a discussion of TM, TLE, and the TMTS in Section II. In Section III, we describe the high-level behavior of PBZip2 and x265. Section IV discusses the quiescence mechanism used by GCC’s STM to ensure lock-based semantics, and presents situations in which quiescence overheads are avoidable. Section V discusses problematic lock-based code in x265, which is not immediately transactionalizable. Section VI briefly discusses additional challenges these appli-

*A preliminary version of this paper appeared at TRANSACT 2017, a workshop without archival proceedings.
lications present, and describes our workarounds. Section VII presents performance results for the two applications, and shows that with modest effort, TM is able to outperform the original lock-based code. Section VIII concludes.

II. BACKGROUND

In this section, we briefly review details of TM, the C++ TMTS, and TLE.

A. Transactional Memory Implementations

TM can be implemented in hardware (HTM), software (STM), or a combination of the two. When TM is implemented in hardware, two hardware instructions are used to indicate the boundaries of a transaction [18]. The begin instruction creates a register checkpoint and informs the cache controller of the need to monitor memory accesses for conflicts with other threads. The end instruction discards the checkpoint and informs the cache controller that tracking is no longer needed. During the transaction’s execution, a set of sufficient conditions on the behavior of the cache dictate whether the transaction remains valid: if the transaction reads a location, then the corresponding cache line must remain in the cache until commit; if the transaction writes a location, then the corresponding cache line must not be evicted or shared with other cores before commit. If a condition is violated, the transaction aborts, and the checkpoint is restored.

When TM is implemented in software, the compiler inserts function calls at the points where a transaction begins and ends. It also inserts a function call in place of each memory access within the transaction [8]. These function calls connect to a library that performs checkpointing, tracks the memory accesses of each transaction, and manages shared metadata for detecting transaction conflicts. For simplicity, one may think of the shared metadata as a table of readers/writer locks, which are acquired as a transaction progresses, and released at the commit point of the transaction. STM algorithms vary with regard to shared metadata implementation, protocols for detecting conflicts, and mechanism for buffering writes.

HTM has less latency than STM: there are no per-access function calls or explicit management of metadata. However, HTM is less flexible: there is no opportunity to introduce any nuance in how conflicts are resolved. In contrast, STM can defer conflict resolution until at least one transaction is guaranteed to succeed. Furthermore, HTM is tightly coupled with the cache controller, and thus hardware transactions cannot access more data than fits in the cache, or survive interrupts and system call. For transactions that deterministically fail to complete in hardware, a common approach is to temporarily serialize all transactions, execute the failing transaction in isolation and without hardware protection, and then re-enable concurrency.

B. The C++ TM Technical Specification

The C++ TMTS defines implementation-agnostic language support for TM in C++. The primary feature of the TMTS is the notion of an atomic block. A program that uses atomic blocks will produce output that is equivalent to an execution in which the atomic blocks executed in some serial order, without overlap. However, under the hood, TM is expected to be used to execute these blocks concurrently.

An atomic block is allowed to “cancel” itself by throwing a special exception. If the exception crosses the transaction boundary, the effects of the transaction are undone. To support cancellation, the compiler must prove that an atomic transaction does not perform any operations that are externally visible before the transaction commits (e.g., I/O operations). In the case of separate compilation, the transaction_safe function type indicates that a function’s effects can be undone. An atomic transaction can only call functions that are statically verified as being transaction_safe.

Sometimes a transaction must perform an “irrevocable” operation [19], [20], e.g., I/O, system calls, and accesses to atomic() variables. The synchronized block construct serves this role. There is an equivalent global total order over all atomic and synchronized blocks. synchronized blocks cannot cancel their effects, but they may perform unsafe operations. When they do, the underlying TM system serializes all transactions, runs the synchronized block to completion, and then re-enables concurrent transactional execution. The GCC implementation of the TMTS also uses serialization to ensure progress: if an atomic transaction aborts repeatedly, it will disable concurrency, run in isolation, and re-enable concurrent transactional execution upon its completion.

The C++ TMTS requires both flavors of transaction to interact with the language-level memory model in an intuitive way. In particular, when transitioning data from transactional to non-transactional state, or vice versa, the TM implementation must not cause races. In particular, when a transaction “privatizes” data, it may need to “quiesce” until all concurrent transactions (which might be accessing that data) commit or abort and complete any pending undo operations [21], [22]. Only then can the TM implementation be certain that a subsequent nontransactional access by the privatizing thread will not race with accesses by transactions that are doomed to abort. Quiescence overheads grow as the number of concurrent transactions increases.

C. Using TM for TLE in C++

A naïve approach to TLE in C++ would replace each lock-based critical section with a synchronized block. This creates several problems. The first is that any serialization of any transaction (whether due to irrevocability or contention) causes unrelated transactions to be suspended. Unfortunately, any use of the TMTS for TLE will have this problem. If a programmer identified critical sections that could be protected by disjoint sets of locks, and then used TM to elide those locks, they cease to be treated as disjoint from the perspective of the TM system. The second problem is that atomic blocks are statically checked for unsafe operations, which silently cause serialization in synchronized blocks. Thus it is
preferable to use atomic blocks, even though they require more effort [23].

An additional challenge when using the TMTS for TLE is that the TMTS does not expose many knobs for low-level tuning of the TM system’s behavior. Karnagel et al. showed that careful tuning of the retry policy on individual transactions was a key factor in improving performance [24]. Similarly, Yoo et al. found that coarsening transactions to encompass multiple critical sections of the original lock-based program could result in less overhead (by amortizing the boundary cost of transactions) [18]. These decisions, however, depend on the underlying TM implementation. In the TMTS, programmers must employ a one-size-fits-all approach.

III. PBZip2 AND x265

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

a) PBZip2: PBZip2 is a 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 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 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 are small and do not make system calls. The input, output, compression, and decompression operations are performed outside of critical sections. The main source of contention is for the locks protecting the inter-stage queues.

b) x265: x265 encodes and decodes video streams and images to and from the HEVC/H265 compression format. The encoding and decoding algorithms divide each frame into sequences of macroblocks called “slices”, which are passed to worker 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”, which can be independently processed. Additional parallelism is achieved via a wavefront algorithm used in individual frames. 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 objects.

IV. QUIESCENCE AND LOCK ELISION

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 [25]–[28], the STM algorithm in GCC does not, and requires a committing transaction to execute code similar in spirit to a user-space RCU Epoch [29].

The C++ TMTS uses the notion of transactional sequential consistency [30] to describe a 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 [31].

In this memory model, quiescence ensures that when a transaction transitions data between shared and thread-private states, concurrent transactions accessing that data do not race with legal nontransactional accesses. In HTM, such accesses are not possible. In STM, they can result from delayed undo

There are three main lock objects in x265:

• The lookahead lock prevents concurrent access to shared input and output queues of frames; in essence, it mediates inter-frame parallelism.
• The CTURows lock is used by the wavefront processing algorithm to mediate communication from a completed CTU to the CTUs that depend on it.
• The EncoderRow 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 performance metadata and metrics.

Fig. 1: HEVC wavefront parallel processing [16].
or write-back operations. In C++, publication safety (i.e., ensuring the absence of races when transitioning data to a state in which it can be accessed by nontransactional code) is guaranteed for race-free programs, and thus quiescence 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.

A. Problems with TLE and Quiescence

When the C++ TMTS is used to achieve lock elision, it entails a form of lock erasure: whereas the original program may contain many locks that 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.

In addition, 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, 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 STM transaction quiesces after committing in GCC.

B. 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.\footnote{The issue is that existing STM algorithms would require writing transactions to quiesce before releasing ownership of locations.}

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.\cite{32} 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 offers no value when few transactions privatize.

We propose a new TM API function: \texttt{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\cite{33}).

Listing 2 demonstrates the use of \texttt{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

\begin{lstlisting}[language=C++]
// Vector update thread // Private thread // Proxy thread
for k ∈ 0...size do
    atomic vec[k] ← msg;
    vec[k] ← null
end
if msg = null then
    retry
end
use(msg)
\end{lstlisting}
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.

C. Pitfalls

TM.NoQuiesce has the potential to significantly increase scalability: quiescence can introduce cache misses linear in the number of threads, to determine when each thread is 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 [34] is able to identify all races that arise when a TM library fails to provide privatization safety. Extending T-Rex to understand implicitly privatization-safe STM with selective disabling of privatization appears to be straightforward.

V. 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, TLE ought to be easy: 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 [23], 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 [35], 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. The solution in that work was to model these accesses 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 mirrored past work with memcached: lock-based critical sections were replaced with transactions, and as appropriate, functions were annotated for transaction safety. 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 could not complete.

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 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 workload configurations and thread counts, we confirmed that this modification did not affect performance. With the change in place, each of the critical sections could be separately transactionalized.

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

- Can it be proven that naïve transactionalization is safe for critical sections that obey two-phase locking?
**Listing 4:** A ready flag avoids lock nesting, facilitating transactionalization.

```c
// Lock no longer held during
// produce stage
1 out_queue.lock()
2 element ← new queue_node()
3 out_queue.enqueue(element)
4 element.ready ← false
5 out_queue.unlock()
6 produce(element)
7 out_queue.lock()
8 element.ready ← true
9 out_queue.unlock()

// Lock acquired during final stage
// of pipeline
10 e ← nil
11 out_queue.lock()
12 if out_queue.peek().ready then
13   e ← out_queue.dequeue()
14 out_queue.unlock()
15 out_queue.lock
16 queue.dequeue
17 element.ready
```

- Under what conditions will naive transactionalization of non-two-phase locking code remain safe? To the best of our knowledge, no prior work has dealt with these problems, or explored transactionalization of programs that did not obey two-phase locking.

VI. 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.

c) Logging Overheads: Both applications can be configured to produce diagnostic output to logs while locks are held. Such output cannot be rolled back, and hence ought to serialize transactions. These outputs are similar to log output in the transactional versions of memcached [23] and Atomic Quake [36]. In those applications, the programs did not require any ordering among logging operations: log messages are timestamped, the order can be determined post-mortem, and the return values of any syscalls during logging are ignored. Consequently, those log operations could either be executed unsafely (and possibly more than once) by STM, or deferred until the end of the transaction. In our applications, we chose to defer output [9]: if we marked the output as unsafe, it would still lead HTM to serialize.

d) 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 [37]. However, the library did not support timeouts. We extended the condition variable library to allow timed wait operations via POSIX semaphores. We verified that this change had no impact on the behavior of the original lock-based program. However, in our experiments, condition variables did present a common source of serialization, especially for HTM. We leave exploration of this problem as future work.

e) 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.

VII. 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 [37], 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 3.40GHz. 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 [38]). The HTM results fall back to a serial mode after hardware transactions fail twice. We did not pin threads to specific cores: since our tests are on a single-chip machine, pinning did not offer any significant benefit.

A. PBZip2

PBZip2 offers two independent operations, Compress and Decompress. We tested each, using a 650MB test file. Within PBZip2, we varied the number of worker threads, as well as the size of the blocks that were processed in parallel; all other configuration parameters were left at their defaults. In our experiments, we varied the number of worker threads from 1 to 8, and considered 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 computation.

We compare five algorithms. The baseline is the original code, which uses pthread mutex locks. We then consider three STM algorithms: STM + Spin uses GCC’s ml_wt algorithm, with 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 sections in PBZip2 is to protect queue metadata. Therefore, the average size of critical sections is small. Each thread can access the queue metadata after it finishes compressing or decompressing its block. Conflicts among critical sections are rare: for a 650MB test file, we observed between 950 and 1100 transactions, of which 0.1% aborted at least once in STM. In the HTM experiments, 13% to 18% of transactions aborted twice and fell 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 compute resources, but also increases contention (between caches and between transactions). 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 at higher thread counts STM + CondVar and STM + CondVar + NoQuiesce both outperform the baseline in Figures 2a and 2f. 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.

B. x265

To evaluate x265, we considered 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 varied the number of worker threads, and again considered the five algorithms from above. The impact of spinning was 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 was 9.5% better than pthreads at 4 threads (Figure 3b). Moreover, HTM outperformed pthreads in almost every case. Again, this was with the untuned GCC HTM support: the abort rates in Figure 4 suggest that better performance is possible by tuning the fallback
C. Is Quiescence Overhead Important?

Given our past experience with transactional workloads, we expected eliminating quiescence to have a more significant impact on performance. To gain a deeper understanding of its potential benefits, we conducted targeted microbenchmark experiments on a larger machine, with two Xeon X5650 CPUs, each of which has 6 cores/12 threads and runs at 2.67GHz. This test machine includes 12GB of RAM, and runs a Linux 4.4.0 kernel. We used the GCC 5.3.1 compiler. Results are the average of three 10-second trials. This CPU does not support hardware TM. The STM results use the same GCC STM algorithms and configurations as prior experiments.

Figure 5 presents experiments for three data structure microbenchmarks and two configurations. We limit our focus to high-contention scenarios with small transactions: a list-based set storing 6-bit keys, a hash-based set storing 8-bit keys, and a tree-based set storing 8-bit keys. Transactions execute randomly-selected operations using randomly-selected keys. On the left side of the figure, operations are split evenly between inserts and removes. On the right side, half of the operations are lookups, with the remainder split evenly between inserts and removes.

We consider three STM implementations: the baseline GCC implementation (STM), with quiescence after every transaction; an implementation in which no transactions quiesce (NoQ); and our implementation that uses TM.NoQuiesce to selectively disable quiescence (SelectNoQ). Note that eliminating all quiescence is not correct: whenever a removing transaction frees memory, the GCC TM requires it to quiesce before returning that memory to the system allocator.

In the list experiments (Figure 5a and 5b), selectively disabling quiescence offers the same benefit as the unsafe NoQ option. Note that in this workload, little scaling is expected. Note, too, that the dip in performance at two threads is expected; it results from inter-chip communication on a global counter within the GCC STM implementation. Surprisingly, with 50% lookups, selectively disabling quiescence outperforms globally disabling quiescence. In the experiment, the list is initially 50% full, and hence at any time a transaction has a 12.5% chance of performing a successful remove operation.
and quiescing. Since the benchmark executes transactions in a tight loop, any quiescence represents a period with less contention. In particular, for transactions that must traverse to the end of the list, quiescence by concurrent threads provides a chance to make forward progress. In essence, a small amount of quiescence provides congestion control.

In the hash and tree workloads, conflicts are much less likely, since transactions access disjoint regions of the data structure. In these cases, SelectNoQ performs on par, though slightly below NoQ, and both outperform the baseline STM. Moving from hash to tree, conflicts become more likely. As they do, we see the same trend as with the list: at high contention, occasional quiescence gives expensive transactions a chance to complete.

In summary, we observe that removing quiescence has a benefit that is proportional to the frequency of transactions and their length. Even when transactions are infrequent or tiny, quiescence is worth eliding. When transactions become more common and larger, quiescence becomes a dominant overhead, and removing it correctly and safely is a valuable optimization. These experiments also explain the variability in STM performance in PBZip2 and x265: when an STM implementation does not provide any other form of contention management, quiescence becomes a congestion control tool. As the TMTS expands to include programmer-specified policies for how to handle conflicts, this behavior should become less likely.

VIII. 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 TLE 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 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 speed up STM executions of a program. However, our experiments show that quiescence currently serves as a form of implicit congestion control, and eliminating it can lead to increased abort rates and decreased performance. We believe that the TMTS should allow programmers to specify contention management policies, so that the effect of quiescence can be more predictable.

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.