Evaluating and Optimizing Real-Time Software Transactional Memory

by Shai Caspin

Honors Thesis

Department of Computer Science

University of North Carolina at Chapel Hill

April 2021

Approved by:

Thesis advisor: James H. Anderson _____

Second reader: F. Donelson Smith _____

Coordinator: Donald E. Porter _____

Abstract

Software transactional memory (STM) is a proposed solution to the challenge of developing correct concurrent code. STM allows programmers to annotate sections of their code that need to be synchronized, and the STM implementation resolves synchronization issues behind the scenes. One application domain where STM can be particularly useful is real-time systems, where schedulability is a crucial metric is system certification. However, STM usually relies on retries to resolve contention, which makes theoretical worst-case behavior, and thus schedulability, highly pessimistic and therefore impractical in the real-time application domain. Previous work on real-time STM has failed to give both theoretical and practical solutions to this problem. Work in this thesis is part of a large effort to present real-time STM that is both schedulable and high-performing, achieved using a retry-free, and thus entirely lock-based, STM implementation. This work details multiple metrics for evaluating retry-free STM compared to a known retry-based lock-based solution, as well as presents optimizations to a locking protocol for increased performance in this context. Evaluations show a retry-free STM implementation with optimized locking protocols is significantly more schedulable and higher performing on single-socket machines than other lock-based STM.

Acknowledgements

Thank you to all my friends, collaborators, and mentors on this project and during my time as an undergraduate. Your guidance and advice has been invaluable. Thank you to my advisor Jim Anderson, mentors Catherine Nemitz and Bryan Ward, and collaborators Claire Nord and Nathan Burow, for making this project a joy to work on while allowing me to grow and challenge myself as a researcher. Thank you to Don Smith and Don Porter for helping me get into research and advising me along the way. Additional thanks to my parents, sisters, and friends for their never ending support.

Contents

1 Introduction **5**
1.1 Organization 6
1.2 Contributions 6

2 Background **7**
2.1 Transactional Locking II 7
2.2 Implementation 10
2.3 Phase-Fair Reader/Writer Locks 11

3 Schedulability **13**

4 PF R/W Lock with Light Reading **18**
4.1 Design 19
4.2 Evaluation 19

5 Throughput **23**
5.1 Branching Benchmarks 23
5.2 Object Grouping Exploration 26

6 Conclusion **29**

References **31**

1 Introduction

Multiprocessor shared-memory systems are now commonly used, allowing greater opportunities for concurrency. Concurrent code requires synchronization mechanisms to coordinate access to shared memory. Proper synchronization can be tricky to achieve, and synchronization bugs can lead to system failures. Software Transactional Memory (STM) is a high-level concept for handling concurrent code where synchronization mechanisms are hidden from the programmer. Code that has to be synchronized is annotated as a *transaction* by the programmer, and the STM implementation handles synchronization internally, usually at a library or compiler level. STM has been well-studied for throughput-oriented systems, with multiple designs and implementations across different programming languages.

STM traditionally relies on speculative transaction execution with aborts and retries to resolve contention. This also introduces greater chance for concurrency. However, retries introduce additional uncertainty in reporting timing guarantees, which are crucial for certifying real-time systems. One such aspect for certification is *schedulability*, which details whether a system can be scheduled without missing any deadlines. Retries can cause increased overheads and worse overall schedulability. No previous exploration of real-time STM has detailed the benefits of a retry-free approach to STM. Retry-free STM can be achieved using entirely lock-based synchronization providing well-studied timing guarantees. Questions then arise of how to precisely evaluate STM performance outside of just traditional throughput, and how to optimize a retry-free implementation to match retry-based performance in the common case.

1.1 Organization

This thesis explores multiple metrics for evaluating lock-based STM using schedulability analysis and throughput benchmarks with a focus on predicting worst-case behavior. First, I will provide necessary details on the theory behind retries and locks, as well as the actual STM designs compared in this work. Then, I will detail the schedulability gains of an entirely retry-free STM, discuss and evaluate how a locking protocol can be tailored for performance, and finally compare two lock-based STM implementations, one of which is entirely retry-free. These efforts are all towards showing that a more predictable retry-free STM implementation can achieve comparable performance to more traditionally designed retry-dependant STM.

1.2 Contributions

The work detailed in this thesis is part of a large effort to create and promote real-time STM that is both predictable (and thus certifiable) and performant. The retry-free STM implementation evaluated in this thesis was designed by Claire Nord, and detailed in her Master's Thesis [16]. Much of the theoretical work was done in collaboration with Catherine Nemitz, and the locking-protocol design was done in collaboration with both Catherine Nemitz and Bryan Ward. My contributions are mainly in designing all the tests, throughput benchmarks, and overhead experiments, implementing various locking mechanisms and state structures in both Rust and C, and aiding with the design of the new locking protocol presented.

2 Background

Transactional Memory (TM) was first proposed by Knight [12] and then further popularized by Herlihy and Moss [11] as a hardware-supported mechanism for synchronization. Shavit and Touitou then proposed a software-only solution [22, 23], branded *Software Transactional Memory*, using non-blocking synchronization. Dice, Shalev, and Shavit presented *Transactional Locking II* (TL2) as a lock-based solution that relied on retries to prevent deadlock [10]. In retry-based STM, synchronization is handled by contention managers, which are mechanisms applied to ensure progress when transactions conflict.

Prior work on real-time TM focused on developing more predictable contention managers. Several previous STM systems (e.g. [1, 24]) are designed for real-time systems, but lack any formal analysis for worst-case behavior. Instead, such systems have been empirically evaluated based on average-case responsiveness, deadline-miss ratios, and/or synchronization overheads. Sarni *et al.* [17] presented the first real-time contention manager with associated schedulability analysis. Other real-time contention managers were presented by El-Shambakey [21], who found real-time locking protocols yield overall better average deadline-miss rations than any retry-based contention manager. Schoeberl *et al.* [19, 20] presented real-time TM with static program-analysis to determine retry bounds. Belwal and Cheng [2] investigated the effect of eager vs. lazy conflict detection on real-time schedulability.

2.1 Transactional Locking II

To show gains and losses of a completely retry-free approach, the retry-free implementation is compared against the Transactional Locking II (TL2) algorithm [10, 7, 9] presented by Dice, Shalev, and Shavit. TL2 was chosen for four main reasons: (i) like our retry-free approach, TL2 is mostly lock-based, though

it does allow for retries; (ii) TL2 is a well-established algorithm (over 1,000 citations); (iii) it has an open-source Rust-based implementation [13], allowing us to conduct performance comparisons without the confounding variable of language choice as our implementation is in Rust, a choice that will be explained later; and (iv) measurement-based studies of some prior real-time STM approaches have been shown inferior to lock-based synchronization [21].

In TL2 transactions are pre-executed to determine which objects they access. This simulation requires additional overhead yet enables greater runtime concurrency. After the simulation phase, locks are acquired for each of the written objects so that results can be committed. In the interest of average-case performance, such locks are acquired in an arbitrary order, potentially leading to deadlock, which causes a transaction to abort and retry. A transaction is forced to retry if an object it accesses is updated (or locked) after its simulation, thus invalidating the simulation's results. Transactions may conflict and force retries at runtime if they access common objects.

The TL2 algorithm differentiates between read and write transactions, with certain optimizations for high performance. A write transaction will perform the following steps:

1. Sample global version-clock
2. Perform speculative execution
3. Lock write-set
4. Increment global version-clock
5. Validate read-set
6. Commit changes and release locks

A read transaction will perform the following steps:

1. Sample global version-clock
2. Perform speculative execution

A read-only transaction is designed to incur lower overheads. Dice, Shalev, and Shavit purposely presented TL2 with *low-cost read-only transactions* claiming read-dominant workloads dominate usage patterns in applications [10]. They proceeded to show certain design aspects of TL2 outperformed previous STM implementations on a red-black tree, in particular due to "improved locality in accessing the locks and the data" provided by one of their implementations [10]. Both read and write transactions incur significant overhead in preserving transaction state so that the speculative execution can be reversed if a transaction is forced to abort and retry. Retrying and aborting is also costly in terms of time spent performing unused computations or rolling them back. A transaction can be forced to abort and retry at both steps 2 and 5 of write transactions, and step 2 of read transactions.

TL2's approach is deadlock-free since in the case of deadlock all transactions will retry. Additional measures to prevent deadlock can be added on top of the basic TL2 algorithm to ensure that two transactions do not continuously cause deadlock. Dice, Shalev, and Shavit note that locks are not ordered when acquired in step 3 of a write transaction, which increased the frequency of deadlock but is overall more efficient in the read-dominant workload case [10].

2.2 Implementation

A retry-free STM implementation must provide the same basic correctness guarantee of progress. This can be achieved using real-time locking protocols with the same guarantees. Nord's TORTIS [16] is a compiler-level Rust implementation of lock-based retry-free STM, which motivated the work in this thesis. TORTIS stands for try-once real-time STM. TORTIS leverages Rust's inherent thread-safety properties and strong type system to identify resources requiring synchronization and assign locks properly within the Rust compiler.

TORTIS's compiler extension includes wrappers for data objects that must be synchronized. Rust's type system allows the compiler to then identify all wrapped object and include them in the contention-management scheme. TORTIS generates a graph where each data object represents a node. If two objects are accessed within a single transaction, an edge connects them. TORTIS then groups each connected portion of the graph to be locked by a single lock. The locking mechanism used can be interchanged based on user-space definitions, and replaces the *transaction* keyword during compilation. The compiler identifies objects that must be synchronized, assigns a lock to each group of objects with inter-dependencies, and substitutes a user-space-defined lock as the lock for said group.

TORTIS also distinguishes between read and write transactions. Rust's thread-safety primitives dictate how shared objects are accessed within threads, and allows TORTIS to distinguish between a read (borrow) and a write (borrow mutable). A *borrow* operation does not demand exclusive access while *borrow mutable* does, distinguishing between read and write object accesses. A transaction is only marked a *read transaction* if all borrow accesses are non-mutable, *i.e.*, all accesses are reads. A system with only read transactions can run with no synchronization. Note read-only transactions still need to be labeled to ensure

linearizability of reads and writes, which guarantees a read issued after a write will read the new value written.

2.3 Phase-Fair Reader/Writer Locks

Synchronization mechanisms that differentiate between read and write operations have been studied widely, one variant of which is reader/writer (RW) locking protocols. Some approaches to RW locking can starve one type of request over the other by giving one type preference [8, 15]. Phase-fair (PF) reader/writer locks were proposed for lower overall *blocking bounds* in real-time systems and for eliminating preferences [6]. Blocking bounds are theoretical bounds for worst-case blocking, *i.e.*, how much time a process attempting to lock a resource can spend waiting for other processes to finish. PF locks alternate read and write phases to ensure the order in which read or write requests are made is preserved, *i.e.*, no read that arrived after a write will execute before that write and vice versa.

Brandenburg and Anderson presented several PF implementations, including a ticket-lock-based implementation (PF-T), a more compact version for embedded systems (PF-C), and a queue-based implementation (PF-Q) [5]. The PF-T and PF-C exhibit $O(n)$ *remote memory reference* (RMR) time complexity and PF-Q exhibits $O(1)$ RMR time complexity. The RMR measure accounts for the number of interconnect traversals to memory for any operation, and is the metric for evaluating the time complexity of spin-based concurrent algorithms. Constant RMR is generally associated with lower *lock overheads* since memory references are costly. Brandenburg showed PF-T exhibits lower overheads in comparison to PF-Q for systems with high contention [4].

Listing 1 Original PF-T Implementation

```
 1: type res_state: record
 2:     rin, rout: unsigned integer, initially 0
 3:     win, wout: unsigned integer, initially 0

 4: constant
 5:     RINC 0x100  // reader increment value
 6:     WBITS 0x03  // writer bits in rin
 7:     PRES 0x02   // writer present bit
 8:     PHID 0x01   // writer phase ID bits

 9: procedure READ_LOCK(ℓ: ptr to res_state)
10:     var w: unsigned int
11:     w := fetch&add(ℓ→rin, RINC)& WBITS                    ▷ In read queue
12:     await (w = 0) or (w ≠ (ℓ→rin & WBITS))                ▷ Satisfied

13: procedure READ_UNLOCK(ℓ: ptr to res_state)
14:     atomic_add(ℓ→rout, RINC)

15: procedure WRITE_LOCK(ℓ: ptr to res_state)
16:     var w, wticket, rticket: unsigned int
17:     wticket := fetch&add(ℓ→win, 1)                        ▷ In write queue
18:     await (wticket = ℓ→wout)                              ▷ Head of write queue
19:     w := PRES | (wticket & PHID)
20:     rticket := fetch&add(ℓ→rin, w)                        ▷ Marked all reads to see
21:     await (rticket = ℓ→rout)                              ▷ Satisfied

22: procedure WRITE_UNLOCK(ℓ: ptr to res_state)
23:     fetch&and(ℓ→rin, 0xFFFFFF00)                          ▷ Clear WBITS
24:     ℓ→wout := ℓ→wout + 1
```

Brandenburg and Anderson's phase-fair reader/writer ticket lock is presented in Listing 1. It is important to note the variables in the *res_state*, specifically *rin*, which is the location on which reads spin. In line 12 of Listing 1, the *read_lock* procedure continuously checks the value of *rin*, potentially causing remote memory references if that value is overwritten and needs to be updated in cache. Another cause for remote memory references is atomic operations such as $fetch\&...$, which force remote memory references by updating the value in memory atomically. Overheads are measured to find worst-case execution time, done so by testing systems with high contention and frequent RMR.

3 Schedulability

An important aspect of real-time systems is *schedulability*. Schedulability assessed whether a given protocol, alongside a scheduler, can ensure a collection of tasks is scheduled without any tasks missing their deadlines. Schedulability hinges on whether the *worst possible* execution order still meets all deadlines. Since TL2 and other STM algorithms are retry-based, schedulability must account for any possible interaction between transactions that may force aborts and retries. If a system is *schedulable*, the transactions within the system can be scheduled under any interleaving of events of execution. The purpose of this section is to highlight schedulability differences between a retry-free approach (TORTIS STM with a phase-fair reader/writer lock) and a lock-based approach that is not retry-free (TL2).

Schedulability studies are conducted by generating a scenario of a system with certain parameters, each scenario has many randomly generated task systems which model possible execution behavior. Each task is then analyzed by a schedulability test, that determines whether the random task set is schedulable. For each scenario, the reported schedulability ratio indicates how many of the randomly generated task systems were schedulable under the given protocol.

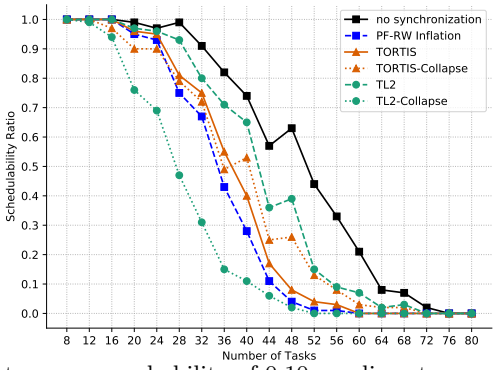
When analyzing an individual transaction, each object that may be accessed must be considered. While at runtime separate transactions could access different data objects concurrently, when determining schedulability, we must account for any possible conflict. Thus, the analysis assumes each access to a data object may conflict with any other access to it. Different object groupings depend on how well the STM implementation can detect resource groups. If a system "collapses" and treats multiple objects as one, the resulting schedulability differs from that of a system in which every object is in its own resource group.

To determine schedulability, we used SchedCAT's inflation-free framework

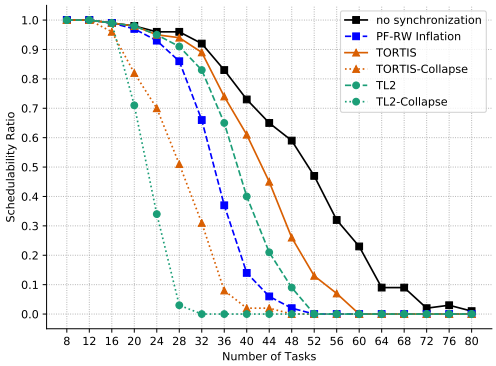
provided by Biondi and Brandenburg [18, 3] . Schedulability is analyzed for of task systems under a partitioned earliest-deadline-first scheduler on an eight-processor platform. (This mirrors the upper end of platform size evaluated with the inflation-free framework.) The inflation free framework utilizes linear programming techniques to bound blocking that can occur for all types of transaction interactions. For example, a write may cause another one to block, a write may block multiple reads, etc., and the linear program must account for all these scenarios for each transaction.

The experimental scope generated includes task systems for scenarios categorized by task periods, task utilizations, number of shared objects, object-access durations, the probability that a task contains any transactions for a given object, the number of transactions that task issues for that object, and the probability that a transaction will be a read or a write transaction. These parameters vary to represent systems similar to those studied in prior work [3] on schedulability. A *scenario* is defined by a particular selection of these parameters. Task periods were selected from a log-uniform distribution in $[10ms, 100ms]$ or $[1ms, 1,000ms]$. Each task's utilization was selected from an exponential distribution with a mean of 0.1. The number of data objects was chosen from $\{4, 8, 16\}$. For each task, the probability that a transaction accesses a given data object was chosen from $\{0.1, 0.25, 0.5\}$. If a task accesses a given data object, it either contains one transaction for that object or it contains a number of transactions for that object selected from $\{1, \ldots, 5\}$. (Each transaction accesses only one data object.) Each access was set to be a write transaction with a probability chosen from $\{0.1, 0.25, 0.5, 1.0\}$. Transaction lengths for both TL2 and TORTIS , as well as in the case of a read or a write, were selected uniformly from either $[1\mu s, 25\mu s]$ (*short*) or $[25\mu s, 100\mu s]$ (*medium*).

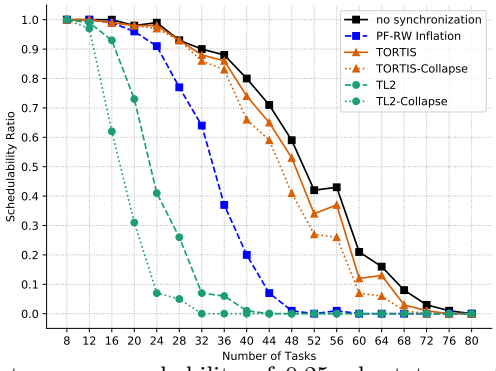
Considering all possible combinations of task-set parameters results in 288



(a) 8 objects, access probability of 0.10, medium transaction length, 1 transaction, write probability of 0.10, task period chosen from $[1ms, 100($



(b) 16 objects, access probability of 0.10, medium transaction length, number of transactions chosen from $\{1 \ldots 5\}$, write probability of 1.0, task perio



(c) 4 objects, access probability of 0.25, short transaction length, 1 transaction, write probability of 1.0, task period chosen from $[1ms, 1000ms]$

Figure 1: Schedulability results for scenarios with near the (a) minimum, (b) median, and (c) maximum improvement ratio.

scenarios. For each one, task systems with a number of tasks between $\{8, 12, .., 80\}$ was generated. For each number of tasks, 1,000 independent task systems were generated. Then, the *schedulability ratio* was computed by taking the ratio of tasks systems that were schedulable by each approach out of the 1,000 systems generated. For the curves labeled as "collapse" in Fig. 1 each system of tasks is modified to direct all resource accesses to a single "collapsed" resource.

To summarize the relative performance of approaches under a given scenario the *task schedulable area* (TSA) was computed, which is the area under its schedulability curve as computed with a midpoint sum. A higher TSA indicates that more task sets can be schedulable. We report the TSA ratio showing the improvement of TORTIS over TL2, and highlight relevant examples.

Schedulability results drive a few key observations that make the case for retry-free STM.

Obs. 1 *TORTIS dominates TL2 with respect to schedulability in most cases.*

This observation is shown in Fig. 1. For 86.4% of scenarios, the TSA of TORTIS was higher than that of TL2. While the TL2 schedulability analysis was our own, we used a recent schedulabilty analysis framework and did not charge overheads for retries (other than accounting for actually re-running the transaction). In roughly half of the scenarios in which TL2 resulted in a higher TSA than TORTIS , schedulability was poor for both approaches (TSA less than 10.0 for both).

Obs. 2 *In higher-contention scenarios, TORTIS results in higher schedulability than TL2.*

This is observed in Fig. 1(b) and (c), with write-access probability of 1.0 in both. (With tasks more likely to access each resource and a higher probability of accesses being writes, transactions are more likely to conflict with each other.)

For example, in 92% of scenarios with write probability of 0.5 or 1.0, TORTIS results in higher TSA than TL2. Recall that in the analysis of a system, a transaction can only be considered to be a read transaction if all resources will only be read—if this guarantee cannot be made, the transaction must be considered a write for schedulability analysis. Thus, while reads may be common at runtime, considering writes may be required for analysis. Additionally, TL2 is designed for the common case where contention is rare, but when contention is high TORTIS provides substantially higher performance (Obs. 2). Schedulability analysis is based on bounding the worst-case contention, so retry-free, deterministic synchronization as used in TORTIS provides improved schedulability.

Obs. 3 *The accuracy of resource groupings affect TORTIS and TL2 similarly, and thus TORTIS dominates in most cases.*

TORTIS leverages a context-sensitive flow-insensitive data-flow analysis to determine resource groups. This is the same type of data-flow analysis that is used to determine retry bounds [19]. Therefore, if conservative data-flow-analysis collapses all resources into a single group, it will also cause significantly more transactions to conflict in TL2. Across all considered scenarios, comparing the original and "collapsed" configurations for both TORTIS and TL2, TORTIS has a higher TSA in 86.4% and 75% scenarios, respectively.

Listing 2 PF-L Implementation

```
 1: type res_state: record // all aligned on different cache lines
 2:     read_status: array of unsigned integer, each initially COMPLETED   ▷ Cache aligned
 3:     win, wout: unsigned integer, initially 0

 4: constant
 5:     WINC 0x100  // writer increment value
 6:     WBITS 0x3  // writer bits in win
 7:     PRES 0x2  // writer present bit
 8:     PHID 0x1  // writer phase ID bits
 9:     PRESENT 0x3  // reader present indicator
10:     COMPLETED 0x4  // reader completed indicator

11: procedure READ_LOCK(ℓ: ptr to res_state, k: core index)
12:     var w: unsigned int
13:     ℓ →read_status[k] := PRESENT
14:     w := ℓ→win & WBITS
15:     ℓ →read_status[k] := w & PHID          ▷ To wait on write phase (w & PHID), if active
16:     await  (w & PRES = 0) or (w ≠ (ℓ→win & WBITS))                          ▷ Satisfied

17: procedure READ_UNLOCK(ℓ: ptr to res_state, k: core index)
18:     ℓ →read_status[k] := COMPLETED

19: procedure WRITE_LOCK(ℓ: ptr to res_state)
20:     var wticket, read_waiting: unsigned int
21:     wticket := fetch&add(ℓ→win, WINC) and ¬WBITS                          ▷ In write queue
22:     await (wticket = ℓ→wout)                                              ▷ Head of write queue
23:     fetch&xor(ℓ→win, 0x3)            ▷ Marked present and new phase for reads to see
24:     read_waiting := ℓ→win & PHID
25:     for k in core numbers do
26:         await (read_status[k] = read_waiting) or (read_status[k] = COMPLETED)

27: procedure WRITE_UNLOCK(ℓ: ptr to res_state)
28:     fetch&and(ℓ→win, 0xFFFFFF01)                          ▷ Clear PRES, but keep PHID
29:     ℓ→wout := ℓ→wout + WINC
```

4 PF R/W Lock with Light Reading

Schedulability experiments indicate retry-free STM provides stronger theoretical timing guarantees than a retry-based approach such as TL2. The schedulability analysis is universal to any phase-fair reader writer lock as bounds are derived from inherent protocol attributes. However, multiple implementations of such a lock can lead to drastically different overheads. We found that the PF-T as presented by Brandenburg and Anderson (and detailed in Listing 1) did not perform well for read-dominant (sparse writes) workloads. A new goal then

arises to present a new variant of phase-fair reader/writer locking that has *light read overheads* to match the inherent design aspect present in TL2. This section provides details for such a locking protocol, named the PF-L that preserves the same schedulability guarantees but provides lower overheads for read locking.

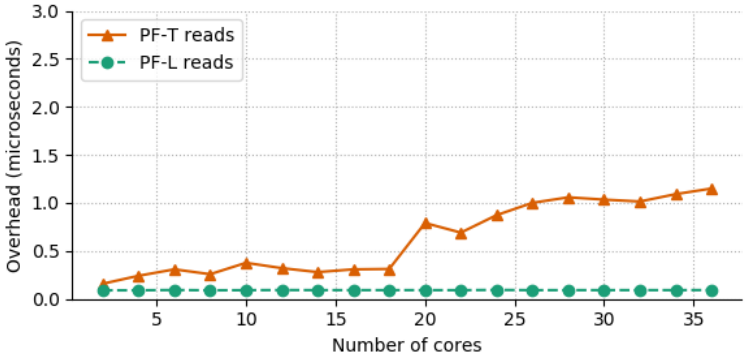
4.1 Design

To provide lower read overheads, the spin locations for reads, *rin* and *rout* in the PF-T (shown in Listing 1), must be modified to allow per-core spin-locations. The solution for this problem requires multiple modifications to the algorithm, the most important of which is allowing each core to have its own *read_status* space in an array that can be written by only readers on that core. This modification also allows correctness of the *read_lock* procedure without any atomic primitives. The pseudocode for the PF-L is presented in Listing 2.

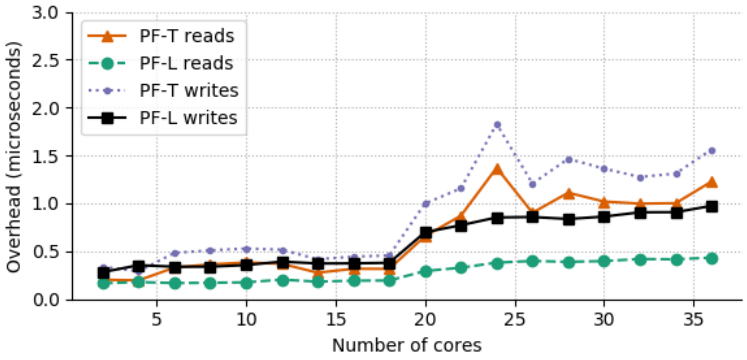
Much of the overheads attributed with lock calls depend on whether the values accessed are cache-hot. In the PF-L implementation all lock-status variables are cached on different lines. This allows each *read_status* variable to exist in a core-local L1 cache, and never be invalidated by readers on other cores. Brandenburg's the PF-T variables are all packed into a single cache line by design to minimize cache-line reloading costs [4]. Both the PF-L and the PF-T have cache-line aligned *res_state* types so that unrelated execution cannot interfere with lock performance.

4.2 Evaluation

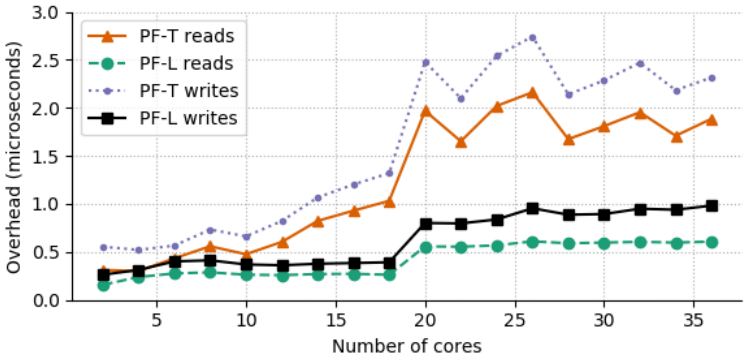
We empirically compared the PF-L to the PF-T implementation [4] on the basis of overheads. In measuring overheads, it is necessary to distinguish time spent in operations inherent to the algorithm (overheads) from those incurred while spinning (blocking). For overhead-measurement purposes only, we instrumented



(a) All reads



(b) 5% writes



(c) 50% writes

Figure 2: Total overheads for lock and unlock operations.

both the PF-T and the PF-L to measure overheads and blocking separately. We recorded blocking and overhead times for 100,000 lock and unlock calls across an increasing number of cores. To simulate high contention and record worst-case overheads, critical sections were empty. All figures present the 99^{th} percentile observed overheads to filter outliers due to interrupts and other jitter due to userspace timing.

We conducted all experiments on a two-socket, 18-cores-per-socket x86 machine running the Linux 4.9.30 LITMUS$^{\mathrm{RT}}$ kernel [14], with two Intel Xeon E5-2699 v3 CPUs @ 2.30 GHz, 128 GB of RAM, and three levels of cache: per-core 32 KB L1 data and instruction caches, 256 KB L2 caches shared by pairs of cores, and 46,080 KB L3 caches shared by all cores on the same socket. We performed each evaluation on $m \in \{2, \ldots, 36\}$ cores and two sockets. For $m \leq 18$, only one socket is used. Fig. 2 shows overhead trends for several different workloads. Overheads were measured separately for reads and writes, each including both lock and unlock costs.

Obs. 4 *The PF-L exhibited constant overheads for an all-read workload.*

Fig. 2a shows that the PF-L exhibits constant lock and unlock overheads of about $0.1\mu s$ across both sockets, while the PF-T overheads are on average $0.4\mu s$ on one socket, and up to $1.3\mu s$ on two sockets. This is attributable to the fact that in the PF-L, read lock and unlock operations only modify a single core-local variable. The PF-T read lock atomically increments a shared variable, which in turn invalidates other caches and bounces the variable across cores and sockets, yielding increased overhead.

As write percentages increase, reads become more costly as *read_status* variables are read by writers on other cores and the write-related variables are constantly updated on all cores with read requests. This behavior also causes an increase in write overheads. The PF-T experiences higher overheads for read-

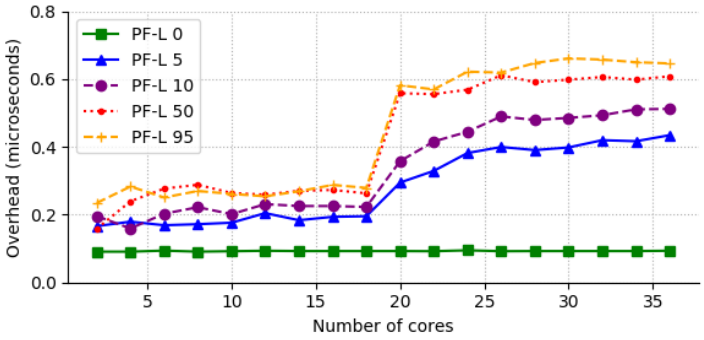


Figure 3: Total read overheads for lock and unlock operations.

dominant (Fig. 2b) and evenly distributed (Fig. 2c) workloads. Since all PF-T variables are on a single cache line, each update invalidates cache-line values for all other cores, resulting in an RMR for every entry and exit section.

Obs. 5 *For all workloads with some writes, overheads increased by up to* $3\times$ *on two sockets.*

All insets in Fig. 2 show higher overheads on two sockets other than the PF-L for an all-read workload. This is attributable to higher cross-socket RMR latencies for both the PF-T and the PF-L for mixed workloads. Fig. 2a highlights the case in which reads in the PF-L generate no RMRs (by design) and does not exhibit increased overheads when executing on two sockets. This claim is further supported by throughput results in Fig. 4, where execution on two sockets consistently yields lower throughput.

Obs. 6 *Reading under the PF-L incurred less overhead than reading under the PF-T.*

For all tested scenarios across varying write percentages and core counts, read operations under the PF-L yielded lower overheads than the PF-T. Fig. 3 shows trends in read overheads with varying write percentages. Beyond 50% writes, overheads are consistent for all read operations and at most $0.7\mu s$. The

PF-L overheads for write-dominant workloads do not appreciably increase beyond 50% writes. With more writes, cache-line invalidations become frequent and cause higher overheads.

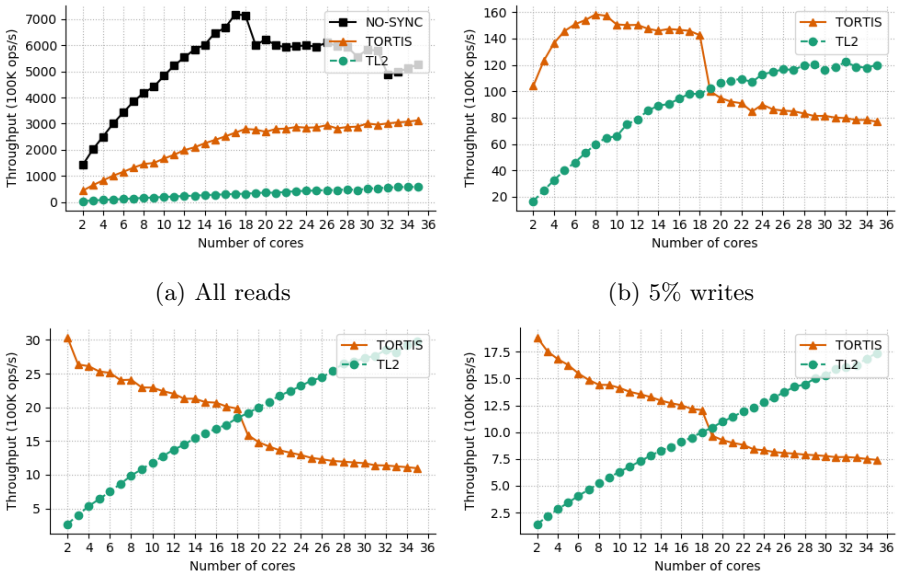
5 Throughput

STM is most commonly evaluated based on *observed average-case throughput*. All experiments are run on the same machine as detailed for the overhead evaluation in Sec. 4. Remember, overheads for the locking protocol increased drastically when execution occurred on more than one socket. The machine architecture has a significant impact on the results presented. TORTIS with phase-fair reader/writer locking and optimized light reads (PF-L) is compared against an open-source implementation of TL2, which includes additional optimizations beyond the basic algorithm [13].

All throughput experiments were conducted in Rust, although the locking protocol is implemented in C and ported. Because of Rust's limitations on sharing data among threads, a Rust implementation of the PF-L required multiple additional atomic operations and overall performed significantly worse than a ported C version of the PF-L.

5.1 Branching Benchmarks

For a branching data structure, TORTIS and TL2 are evaluated on a large red-black tree with time measurements for how operations (some mix of inserts and lookups) per second scale with thread count. For TORTIS , a standard Rust-based red-black-tree implementation is used, with the entire tree, not individual nodes, defined as one shared object. So instead of wrapping individual nodes, the entire tree is wrapped and synchronized as one object. For TL2, we use a red-black-tree implementation given with the Rust TL2 implementation [13].



(a) All reads

(b) 5% writes

(c) 50% writes

(d) 95% writes

Figure 4: Average throughput for varying access patterns in a red-black tree.

Obs. 7 *For an all-read workload, TORTIS outperforms TL2.*

For inserts, we create an array of 100k integers, shuffle the array, and then insert each value as a node, measuring the time it takes for all inserts to succeed. For lookups, we first create a random tree, and then time how long it takes to find every element in the tree in random order. We also evaluate a third case – using half inserts and half lookups. For this 50% writes case, an array of 100k randomly shuffled boolean values (half true, half false) is used to determine whether a transaction is a lookup or an insert. A third of the tree is pre-built so that lookups would be timed correctly from the start. On this partially pre-built tree, we time 50k lookups and 50k inserts.

All work across the insert, lookup, and mixed workloads experiments is partitioned evenly over the number of cores, each thread performing an equal fraction of the work. For each experiment, we averaged the throughput for ten unique random trees and lookup orders to account for possible impact of insert order.

The results are shown in Fig. 4, note the scale for each plot is different.

As highlighted by Sec. 4, for an all-read workload phase-fair reader/writer locks can achieve constant overheads. The NO-SYNC line shown in insert Fig. 4a of Fig. 4 shows the throughput for the same red-black tree set-up but with no attempt at synchronization, and highlights system limitations beyond one socket. Insert Fig. 4a highlights the direct impact of overheads on throughput between retry-free and a retry-based STM locking approaches.

Obs. 8 *TORTIS provides near-constant throughput on the red-black-tree benchmark, outperforming TL2 on one socket.*

The performance of TORTIS is highly influenced by locking overheads. The trends for throughput closely mirror those seen in Sec. 4, where overheads were relatively constant on one socket, and increased greatly on two sockets. This increase in overheads lowers overall throughput of the system. In comparison, TL2 allows linear scaling with increasing core counts regardless of the workloads. However, the extra overhead of maintaining transactional state to enable retries significantly reduces throughput for smaller core counts, enabling the lock-based in-place approach in TORTIS to provide increased throughput.

Obs. 9 *TORTIS does not scale with increasing core counts but TL2 does.*

Fig. 4 highlights the pitfalls for a solely lock-based approach. For any mixed workload, throughput is bounded above by throughput on two cores. This is in part due to the design of the TORTIS red-black tree. Since the entire red-black tree is locked using a single lock, parallelism for writes is impossible to achieve with increasing core counts. This problem does not exist for TL2, where write-parallelization can occur. The benefits of TL2's retry-based approach can be seen in cases of high-contention on a large number of core counts.

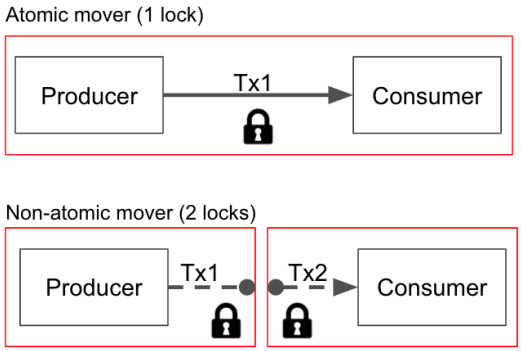


Figure 5: Overview of two mover types

5.2 Object Grouping Exploration

As seen by the schedulability studies presented in Sec. 3, the granularity of synchronization of objects can affect schedulability. Granularity of synchronization is not left entirely to the compiler, but can also be manipulated by the user. For example, transaction keyword placement can prevent any attempt at concurrency. To demonstrate that TORTIS 's object-grouping implementation can recognize different resource groups based on programmer choice, presented here is a producer/consumer case study used in prior work [19]. This case study does not only highlight TORTIS 's ability to identify distinct objects, but shows the impacts of object grouping or "collapse" on throughput. In this study, there is a producer and a consumer queue, with a "mover" task that pops from one queue and pushes to the other, as shown in Fig. 5. Manipulating the placement of sets of transactions over the producer, mover, and consumer allows a study of TORTIS's object grouping and its impact on throughput. Note the same study was not duplicated for TL2 since the data structures associated with the open-source STM implementation do not allow the programmer to choose if multiple objects should be synchronized as one.

There are three sets of transactions in the experiment: a production transaction that pushes to the producer queue, an analogous transaction that pops from

the consumer queue, and a mover transaction that transfers elements between the queues. We investigate two different designs for the mover transaction, as shown in Fig. 5: (i) an *atomic mover* that uses a single transaction for both the push and the pop, and (ii) a *non-atomic mover* that uses a separate transaction for each of the push and the pop. If implemented correctly, the TORTIS analysis should ascertain that the atomic mover causes a transitive conflict between the produce and consumer transactions through the mover transaction. Consequently, the atomic mover design should generate one resource group. In contrast, the non-atomic mover does not generate this transitive conflict, and should lead to two resource groups: one for the transactions on the producer queue, and another for the transactions on the consumer queue.

We divide the workload across multiple threads spanning {3,..,36} cores in multiples of three, allowing a third of threads to be producers, a third to be movers, and a third to be consumers. The same machine as the overheads and throughput experiments is used. Regardless of the number of threads, only one producer and one consumer queue are used. To measure the throughput of the queue system as a whole, we count the number of dequeues performed on the consumer queue. This gives a holistic view to the system's throughput at all levels, as it counts the number of elements that were produced, moved, and then consumed within the given time frame.

The transactions run on a fixed queue of 1,024 elements, and let the system run for three seconds with one second of warm-up time. Note all transactions are classified as *write transactions* by design. The results are presented in Fig. 6.

We demonstrated the correctness of TORTIS 's compiler analysis in two ways: by using a debug version of our library to output the group number for each lock call, and by an analysis of the throughput with the normal runtime library. The debug version of the runtime library verified that the atomic mover

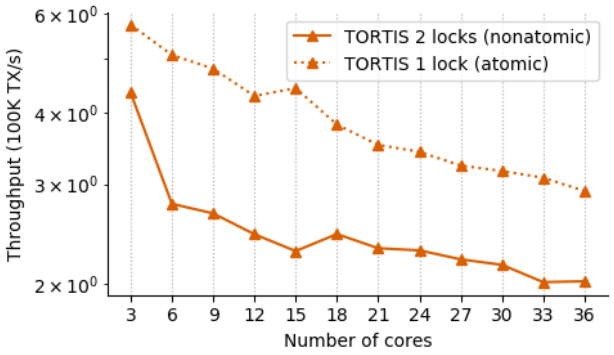


Figure 6: 1024 element queues, 3 seconds, 1 second warm-up

results in one resource group, while the non-atomic mover results in two. Consequently, this case study shows that the TORTIS can indeed distinguish between object groups.

The throughput results for the case study, depicted in Fig. 6, show the performance impact of having two different locks. In particular, the noticeably different behaviour of the atomic and non-atomic cases shows that grouping does impact throughput, and in ways that can be counter intuitive.

Obs. 10 *In a two-queue system, throughput is higher when both queues are protected by only one lock.*

Fig. 6 shows that in a system with one lock protecting both queues, the throughput is significantly higher than in a system with two locks, by an average increase of 1.5× transactions per second. This implies that fine-grained locking may not always yield the best throughput; though using two separate locks allows the producer and consumer to modify their respective queues concurrently, forcing the mover to acquire two locks instead of one doubles the blocking it may incur, and increases the influence of locking overheads.

6 Conclusion

This thesis highlights the schedulability and throughput gains of a retry-free lock-based STM, as well as lock optimizations for increased throughput. Schedulability studies showed that in most cases, phase-fair locks were more schedulable than TL2. Then, a new algorithm for phase-fair reader/writer locking was presented to match *light reading* behavior in TL2. When the new locking protocol was used in TORTIS, it produced higher throughput than TL2 for all systems running on a single socket. The collection of experiments presented show that retry-free STM is not only better from a schedulability perspective, but can also have significant throughput gains. In addition, the object grouping exploration highlights that locking overheads are a crucial part of understanding throughput trends.

Future work can explore multiple aspects mentioned in this thesis. The overhead measurements for the PF-L and the PF-T show that cache-line alignment can impact performance drastically when locks are stressed for worst-case performance. Variants of the PF-T can be implemented with better cache-line alignment schemes to see whether the performance of the the PF-L can be matched for non-read-dominant workloads. Future work can also explore how locking granularity and lock grouping affect performance for larger systems. One of TORTIS 's limitations is that it groups objects pessimistically by design (read Nord's thesis [16] for more details), but perhaps this limitation allows greater performance by minimizing locking overheads. The trade-offs between locking granularity and performance can be further studied to optimize TORTIS. In addition, the new locking protocol for the PF-L was specifically designed with read-dominant workloads in mind; it can be interesting to explore what locking protocols allow greater performance for different workloads, and whether the compiler or lock itself can optimize this behind the scenes.

Retry-free real-time STM can be both practical and high-performing. STM is not widely used out of practical concerns, but perhaps with modifications an entirely lock-based STM can ease concurrent programming efforts on a larger scale.

References

[1] A. Barros, L. Pinho, and P. Yomsi. "Non-preemptive and SRP-based fully-preemptive scheduling of real-time Software Transactional Memory". In: *Journal of Systems Architecture* 61.10 (2015), pp. 553–566.

[2] C. Belwal and A. Cheng. "Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective". In: *Embedded Systems Letters* 3.1 (Mar. 2011), pp. 37–41.

[3] A. Biondi and B. Brandenburg. "Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors". In: *2016 28th Euromicro Conference on Real-Time Systems*. IEEE. 2016, pp. 39–49.

[4] B. Brandenburg. "Scheduling and Locking in Multiprocessor Real-Time Operating Systems". PhD thesis. Chapel Hill, NC: University of North Carolina, 2011.

[5] B. Brandenburg and J. Anderson. "Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks". In: *Proceedings of the ACM International Conference on Embedded Software*. ACM. Oct. 2011, pp. 69–78.

[6] B. Brandenburg and J. Anderson. "Spin-based reader-writer synchronization for multiprocessor real-time systems". In: *Real-Time Systems* 46.1 (2010).

[7] V. Chaudhary et al. "Starvation Freedom in Multi-Version Transactional Memory Systems". In: (Sept. 2017).

[8] P. Courtois, F. Heymans, and D. Parnas. "Concurrent Control with Readers and Writers". In: *Communications of the ACM* 14.10 (1971), pp. 667–668.

[9] G. Cunha. "Consistent state software transactional memory". PhD thesis. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2007.

[10] D. Dice, O. Shalev, and N. Shavit. "Transactional locking II". In: *International Symposium on Distributed Computing*. Springer. 2006, pp. 194–208.

[11] M. Herlihy and J.E.B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM, 1993, pp. 289–300.

[12] T. Knight. "An architecture for mostly functional languages". In: *Proceedings of the 1986 ACM conference on LISP and functional programming*. 1986, pp. 105–112.

[13] T. Kopf. *swym*. `https://github.com/mtak-/swym`. commit f7b635d. 2019.

[14] *LITMUS$^{\mathrm{RT}}$ Home Page*. http://www.litmus-rt.org/.

[15] J. Mellor-Crummey and M. Scott. "Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors". In: *Proceedings of the Third ACM Symposium on Principles and Practice of Parallel Programming*. ACM. Apr. 1991, pp. 106–113.

[16] C. Nord. "Retry-free software transactional memory for rust". MA thesis. Cambridge, Massachusetts: Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science, 2020.

[17] T. Sarni, A. Queudet, and P. Valduriez. "Real-Time Support for Software Transactional Memory". In: *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2009, pp. 477–485.

[18] *SchedCAT: Schedulability test collection and toolkit*. `https://github.com/brandenburg/schedcat`. Accessed: 2020-06-21. 2019.

[19] M. Schoeberl, F. Brander, and J. Vitek. "RTTM: Real-time transactional memory". In: *Proceedings of the 25th ACM Symposium on Applied Computing*. 2010, pp. 326–333.

[20] M. Schoeberl and P. Hilber. "Design and implementation of real-time transactional memory". In: *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*. 2010, pp. 279–284.

[21] M. El-Shambakey. "Real-Time Software Transactional Memory: Contention Managers, Time Bounds, and Implementations". PhD thesis. Blacksburg, VA: Virginia Polytechnic Institute, 2013.

[22] N. Shavit and D. Touitou. "Software Transactional Memory". In: *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM. Aug. 1995, pp. 204–213.

[23] N. Shavit and D. Touitou. "Software Transactional Memory". In: *Distributed Computing* 10.2 (Feb. 1997), pp. 99–116.

[24] R. Yoo and H.-H. Lee. "Adaptive Transaction Scheduling for Transactional Memory Systems". In: *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. 2008, pp. 169–178.