TORTIS: Retry-Free Software Transactional Memory for Real-Time Systems

Claire Nord*[1], Shai Caspin†‡[1,2], Catherine E. Nemitz†§[2],
Howard Shrobe*, Hamed Okhravi¶, James H. Anderson†, Nathan Burow¶, and Bryan C. Ward¶
* MIT CSAIL, † The University of North Carolina at Chapel Hill, ‡ Princeton University, § Davidson College
¶ MIT Lincoln Laboratory

***Abstract*—Software transactional memory (STM) is a synchronization paradigm originally proposed for throughput-oriented computing to facilitate producing performant concurrent code that is free of synchronization bugs. With STM, programmers merely annotate code sections requiring synchronization; the underlying STM framework automatically resolves how synchronization is done. Today, the programming issues that motivated STM are becoming a concern in embedded computing, where ever more sophisticated systems are being produced that require highly parallel implementations. These implementations are often produced by engineers and control experts who may not be well versed in concurrency-related issues. In this context, a real-time STM framework would be useful in ensuring that the synchronization aspects of a system pass real-time certification. However, all prior STM approaches fundamentally rely on retries to resolve conflicts, and such retries can yield high worst-case synchronization costs compared to lock-based approaches. This paper presents a new STM class called Retry-Free Real-Time STM (R^{2}STM), which is designed for worst-case real-time performance. The benefit of a retry-free approach for use in a real-time system is demonstrated by a schedulability study, in which it improved overall schedulability across all considered task systems by an average of 95.3% over a retry-based approach. This paper also presents TORTIS, the first R^{2}STM implementation for real-time systems. Throughput-oriented benchmarks are presented to highlight the tradeoffs between throughput and schedulability for TORTIS.**

I. Introduction

The range of sophisticated features implemented in embedded systems is accelerating at a rapid pace today. These features are being fueled by the availability of high-performance multicore platforms that can support computationally intensive workloads within a restricted size, weight, and power (SWaP) envelope. In recent years, these platform characteristics have enabled a wealth of new capabilities across a wide range of application domains, from medical devices and robotic systems with enhanced intelligence, to automobiles, aircraft, and space vehicles that can function autonomously.

```
transaction {
    let item = queue1.pop();
    queue2.push(item);
}
```

Listing 1: Pseudo-code showing example transaction annotation for an operation moving an object from one queue to another.

In many of these domains, the systems of interest are *safety-critical real-time systems* that have timing constraints requiring certification. In such systems, "performance" is tied to formal analysis involving worst-case system behaviors, unlike throughput-oriented systems, where average-case behaviors are the usual focus and rigorous analysis is not pervasive. Designing a safety-critical real-time system requires facing many analysis-related certification issues, which become more complex due to concurrency on a multicore platform.

The case for real-time transactional memory. One concurrency-related issue that looms large in any application domain is the need for efficient synchronization. In work on real-time systems, multiprocessor synchronization has been the subject of significant research over the last 40+ years, as well summarized by Brandenburg's recent systematic review with 225 references [13]. Many nuances and challenges exist in correctly applying this vast body of work. For example, some real-time locking protocols cannot be used together [12, §4.6.5]—would a typical programmer know this? Moreover, while dealing with concurrency can be generally tricky, adding schedulability concerns only further complicates matters. Such difficulties are exemplified by the work of Chen *et al.* [15], who identified misconceptions related to task suspensions that led to flawed analysis for a number of multiprocessor synchronization protocols that had persisted for over 20 years.

In work on throughput-oriented computing, the need for simple yet efficient and correct synchronization has been the driving force behind considerable prior work on a concept called *transactional memory (TM)*. As illustrated in Lst. 1, when a TM framework is employed, programmers must merely annotate code sections that require synchronization.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering. © 2021 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

This work was also supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

[1]Equal contribution by both authors.

[2]Work conducted at the University of North Carolina at Chapel Hill.

The TM framework itself determines how synchronization is actually achieved. The goal here is to enable programmers to more easily produce correct code that harnesses prominent results on efficient synchronization. An especially strong case for TM can be made for safety-critical real-time systems where there is a need for efficient synchronization, performance analysis, and real-time certification in code that may be developed by a domain expert, not a multiprocessor real-time synchronization expert.

TM approaches. TM can be realized in software, hardware, or some combination of the two [29], [39]. The specific focus of this paper is *software* TM (STM).[1] A myriad of approaches to managing conflicts have been studied that are designed around trade-offs such as minimizing overheads, maximizing throughput, and/or ensuring different progress guarantees. These approaches are categorized in Fig. 1. For example, Herlihy and Moss [29] originally conceived TM as being lock-free, but lock-based synchronization is used in Transactional Locking II [19] to reduce overheads, using retries for limited purposes such as resolving deadlock. In contrast, obstruction-freedom [28] has been presented as a non-blocking synchronization approach for STM with weaker progress guarantees than lock-free synchronization, to trade-off progress for throughput.

Regardless of how synchronization is realized, TM is fundamentally an approach to *automatic synchronization*. The specific synchronization mechanisms used within a given TM framework are designed to realize automatic synchronization while enabling application-relevant performance approaching that of hand-tuned concurrent code. As most work on TM has been conducted in the general-purpose computing domain, most prior work on TM uses *optimistic* techniques to synchronization that *abort* and *retry* conflicting transactions. This optimism can enable non-conflicting transactions to execute and commit concurrently, increasing throughput.

Retry-based synchronization, however, is *not* fundamental to TM; it is merely a means to realize automatic synchronization. Retry-free automatic synchronization has been studied [16], [31] for general-purpose systems,[2] but it has never been applied or evaluated in the context of a real-time system, where worst-case behavior and analysis supersedes throughput.

Contributions. This paper presents the first retry-free (*i.e.*, blocking) real-time STM. We call this class of STM approaches and associated blocking and real-time analysis *Retry-Free Real-Time STM* (*R^{2}STM*). R^{2}STM is designed to minimize worst-case synchronization costs and enable tighter analytical bounds on synchronization-related delays, rather than maximizing average-case throughput. Additionally, eliminating retries enables I/O to be supported within transactions.

The R^{2}STM design is motivated by the fact that in real-time systems, *schedulability* is the paramount application-relevant performance metric, not throughput. Retry-based STM in this context requires *bounds* on retries to validate real-time constraints. Such bounds are inherently based on worst-case contention, rather than the lower average-case contention for which retry-based approaches were designed. This fact makes certifying retry-based systems difficult, as worst-case execution must be accounted for. Furthermore, reasonable bounds for retry-based STM can be difficult to obtain on a multiprocessor. These two observations motivate the retry-free design of R^{2}STM based on the goal of enabling automatic synchronization while maximizing schedulability.

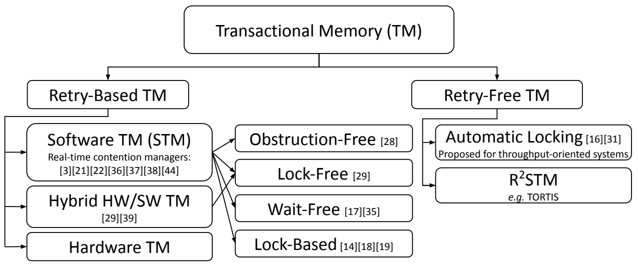


Fig. 1: Hierarchy of TM classes.

Towards this end, we present the first R^{2}STM framework, which we call TORTIS (try-once real-time STM).[3] In order to eliminate retries, TORTIS relies on lock-based synchronization to manage shared-object accesses. TORTIS realizes automatic synchronization via compile-time instantiation of lock and unlock calls. In order to eliminate the possibility of deadlock, TORTIS statically determines the set of objects that *may potentially* be accessed within each transaction, and whether accesses are reads or writes. Knowing the set of objects used in each transaction enables the compiler to compute resource groups [7], and assign each transaction a single group lock, thereby preventing deadlock.

This paper makes four main contributions. First, we conduct a schedulability study that demonstrates an average of 95.3% schedulability improvement when an R^{2}STM approach is used. Second, we present TORTIS, the first demonstration of R^{2}STM. Third, we evaluate concurrency and throughput trade-offs involving retry-based vs. retry-free STM on multiple data structures; we find that the relative performance of TORTIS compared to a competing retry-based STM is highly dependent upon the access patterns, but that overall TORTIS yields competitive throughput while enabling greater schedulability. Fourth, we investigate the locks emitted by TORTIS and their performance in a case study.

Organization. The rest of this paper is organized as follows. After providing needed background in Sec. II, we evaluate retry-free and retry-based STM with respect to schedulability in Sec. III. We then present the design and implementation of TORTIS, in Sec. IV. Next, we evaluate the throughput trade-offs of TORTIS in Sec. V and the impact of our static analysis for resource grouping in Sec. VI. We conclude in Sec. VII.

[1]We use the term "TM" when discussing transactional memory broadly, and "STM" when referring to TM implemented in software.

[2]This work is presented as "pessimistic atomic sections," but we argue any automatic realization of atomic sections is TM.

[3]The predictable and consistent tortoise wins the race.

II. Background and Motivation

Here we provide TM background, with a specific focus on real-time TM. We also describe the analysis assumptions and requirements for schedulability analysis for TM systems.

Transactional memory. The goal of TM is to allow simple transaction annotations that mark atomic sections in source code, as shown in Lst. 1, while having all synchronization handled automatically by the TM system. How this automatic synchronization is realized has been the subject of nearly three decades of work, which has generated several different classes of approaches to handling such synchronization (see [25] for further discussion), as illustrated in Fig. 1. It is commonly assumed that TM approaches must necessarily rely on aborting and retrying transactions to realize such synchronization, but this assumption is not fundamental to TM. It is driven by optimization for the common case in which conflicts are rare. Some STM systems employ locking to realize automatic synchronization, but do so optimistically based on the set of objects that are accessed at runtime within a transaction [19]. Even these lock-based approaches suffer from the possibility of runtime deadlock, and must fall back on abort/retry to enable progress. Yet other STM approaches only guarantee obstruction-freedom [28], which only guarantees progress in the contention-absent case.

The Transactional Locking II (TL2) algorithm [14], [18], [19], which relies on retries, is perhaps the most widely cited locked-based STM in existence, so we use it as a basis of comparison to our new approach. As such, we describe how it functions in more detail. Under TL2, transactions are pre-executed to determine which objects they access. This simulation adds overhead to the transaction execution, but 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 reads is updated (or locked) after its simulation, thus invalidating the simulation's results. In this manner, transactions may conflict at runtime if they access common objects.

Autolocking [16], [31] is an approach to STM from the programming-languages community that frames the problem in terms of automatically inferring the correct set of locks to protect atomic sections. This line of work makes the case for "pessimistic concurrency," *i.e.*, retry-free, for general-purpose applications. To our knowledge, we are the first to investigate retry-free transactional memory for real-time systems. Real-time systems must use fundamentally different locking protocols. In particular, suitable progress mechanisms must be used to limit priority-inversion durations [8]. (We use non-preemptive execution as a progress mechanism in this work; in contrast, in non-real-time systems, user-level processes are typically not allowed to disable preemptions.) As real-time systems are the subject of this paper, we hereafter use retry-free and R^{2}STM interchangeably.

Real-time TM. Prior work on real-time TM has mainly focused on the development of more predictable *contention managers*, which are mechanisms applied to ensure progress when transactions conflict in retry-based implementations. These contention managers often focus on distinguishing read from write accesses to enable reader parallelism. Several previous STM systems (*e.g.*, [3], [44]) are designed for real-time systems, but in fact lack any schedulability analysis. Instead, such systems have been empirically evaluated based on average-case responsiveness, deadline-miss ratios, and/or synchronization overheads.

Sarni *et al.* [36] presented the first real-time contention manager with associated schedulability analysis. Other real-time contention managers were presented by El-Shambakey [21]. Schoeberl *et al.* [37], [38] presented real-time TM for Java chip multiprocessors. Belwal and Cheng [4] investigated the effect of eager vs. lazy conflict detection on real-time schedulability. Note that all prior work on real-time TM and contention management focused exclusively on retry-based conflict resolution.

El-Shambakey's [21] run-time experiments measured average deadline-miss ratios using several contention managers plus the OMLP [10] and RNLP [43] multiprocessor real-time locking protocols. He found that "*more jobs meet their deadlines under [the] OMLP and [the] RNLP than any contention manager by* 12.4% *and* 13.7% *on average, respectively*." This conclusion is based on *observations* and not *analysis*. Including an analysis-oriented comparison would likely have made the observed discrepancy more pronounced. Indeed, retry bounds in multiprocessor real-time systems can be extremely pessimistic. Additionally, retry-based synchronization is prone to large overheads due to retry management (*e.g.*, copying data to roll back aborted transactions).

Quillet *et al.* [35] derived upper bounds on the number of aborts and thus execution time for a retry-based STM system under the Polka contention manager, which was presented in the general-purpose literature. Cotard *et al.* [17] demonstrated a wait-free STM system for multicore architectures and derived upper bounds on retries. In contrast to these works, we show that it is advantageous from a schedulability perspective to realize STM without retries.

Determining objects in transactions. The set of objects accessed within each transaction must be determined (or at least over-approximated) in order to evaluate synchronization-related costs for schedulability analysis in any real-time TM framework. Many prior STM analyses assume this information as part of the task model [21], [22], [36], similarly to how most work on real-time locking protocols assume locks are known *a priori* [8], [42]. Furthermore, many of these analyses assume that each transaction accesses a single object [21], [22]. This assumption is quite limiting—if a transaction accesses only one object, using explicit synchronization such as a lock is simple and efficient. Furthermore, coalescing objects into one for the purpose of analysis is not always safe as interactions between tasks accessing different subsets of objects can in some cases cause transitive retries, and potentially even livelock if not handled properly.

To our knowledge, the only prior work on real-time TM that has presented analysis to determine the set of objects used within each transaction in order to inform retry-bound analysis is that of Schoeberl *et al.* [37], [38]. This analysis is based on static data-flow analysis of the source code of the application. Such data-flow analysis proceeds by tracking the set of objects that *may reach*, *i.e.*, could be used by, each transaction. The precision of such data-flow analysis depends upon many factors, including *context sensitivity*, or interprocedural analysis considering the calling context of a given function invocation, and *flow sensitivity*, or the order of operations in a program. *Aliasing*, or two variables pointing to the same object, can also affect the precision of data-flow analysis. Regardless of the data-flow analysis employed, the more precise the data-flow analysis, the fewer conflicts must be considered in worst-case synchronization analysis.

A key observation in this work is that the analysis required for tight retry-bound analysis, *i.e.*, the objects that a transaction uses, can also be used to implement R^{2}STM. If the set of objects potentially accessed within a transaction is known *a priori*, deadlock can be statically prevented using techniques such as group locking [7] or nesting lock invocations using a partial ordering to prevent deadlock [20], [27], [42], [43]. This enables transactions to be executed without ever needing to retry, opening the possibility of performing I/O within transactions. Furthermore, as we show in our schedulability evaluations, worst-case blocking bounds for modern real-time locking protocols are often much smaller than corresponding retry bounds when using the most recent analyses.

III. Schedulability Evaluation

In this section, we present a comparison of R^{2}STM and retry-based STM approaches on the basis of schedulability. We begin by describing the scope of our evaluation before discussing how schedulability is determined for retry-free and retry-based STM. Finally, we present the results of our schedulability study.

Experimental design. We used SchedCAT [2] to analyze the schedulability of task systems scheduled with the Partitioned Earliest-Deadline-First (P-EDF) scheduling algorithm on an eight-processor platform.

We generated implicit-deadline sporadic task systems for scenarios categorized by a number of parameters, which we varied to represent systems similar to those studied in prior work [6]. A *scenario* is defined by a particular selection of each parameter. Task periods were selected from a log-uniform distribution in $[10ms, 100ms]$ or $[1ms, 1000ms]$. 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 contains some number of transactions for that object. We consider two object-access scenarios: (i) the number of transactions is chosen on a per-task, per-object basis among $\{1, \ldots, 5\}$, and (ii) all tasks have exactly one transaction for each accessed data object. In both scenarios each transaction accesses only one data object. We do not make any assumptions about the order of transactions, or their arrival pattern within jobs. Each access is set to be a write transaction (as opposed to a read-only transaction) with a probability chosen from $\{0.1, 0.25, 0.5, 0.75, 0.9, 1.0\}$. We used transaction lengths selected uniformly from either $[1\mu s, 25\mu s]$ (*short*) or $[25\mu s, 100\mu s]$ (*medium*). We did not include overheads when determining schedulability.

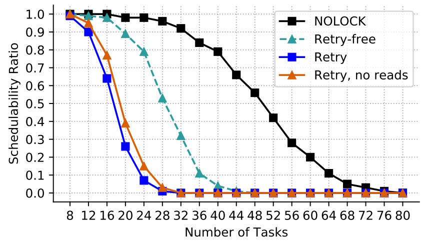
In many STM implementations, reading and writing an object can be handled separately. While reads may be common at runtime, even occasional writes must be accounted for in schedulability analysis. This is true regardless of the STM approach applied. In our evaluation, we assumed that read transactions always read and write transactions always write.

Additionally, when analyzing an individual transaction, we must consider which resources may be accessed. Consider a data buffer as an example. While at runtime separate transactions may access different buffer elements concurrently, when determining schedulability, we must account for any possible conflict. Thus, we assume that each access to a given data object (*e.g.*, buffer) may conflict with any other access to it.

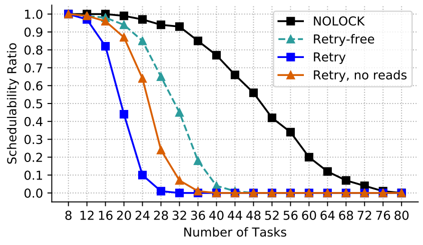
Analysis approaches for synchronization. Regardless of how STM is realized, it can introduce delays (*e.g.*, time spent blocking or retrying) that must be accounted for in schedulability analysis. Our analysis for both R^{2}STM and retry-based STM uses an inflation-free analysis framework [6] to determine schedulability under P-EDF. This framework determines schedulability by incorporating the maximum synchronization interference. *Synchronization interference* on a given CPU partition is the duration of processor time spent blocking or retrying transactions during a given interval.

For an R^{2}STM approach, synchronization interference is entirely due to blocking while waiting to acquire a lock. In the evaluation below, we assume the use of phase-fair reader/writer locks (PF-RW) [11] as the locking protocol supporting the R^{2}STM approach. We apply an extension [34] of the inflation-free framework that provides inflation-free analysis of PF-RW. In our results, this is labeled "Retry-free".

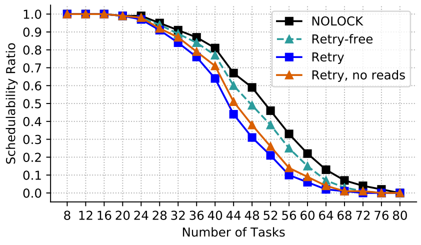
For a traditional retry-based STM, the delays resulting from retrying a transaction contribute to the synchronization interference. For this approach, we apply the non-preemptive lock-free analysis presented with the inflation-free framework [6]. While the analysis of PF-RW incorporates the distinction of read and write access, this lock-free analysis does not. As such, we apply two versions of this analysis: the baseline lock-free analysis (denoted "Retry") and an application of this analysis in which all read transactions are simply ignored (denoted "Retry, no reads"). This second version is intended to capture the best-case scenario, in which read transactions complete with few retries and cause minimal delays to write transactions. Of course, in an actual retry-based implementation, there may be delays incurred by and caused by read transactions, and these may in fact be significant. We discuss this and the implications of other study-design decisions next.



(a) 4 objects, access probability of 0.5, medium transaction length, number of transactions chosen from $\{1 \ldots 5\}$, write probability of 0.9, task period chosen from $[10ms, 100ms]$.



(b) 16 objects, access probability of 0.5, short transaction length, number of transactions chosen from $\{1 \ldots 5\}$, write probability of 0.75, task period chosen from $[10ms, 100ms]$.



(c) 4 objects, access probability of 0.25, short transaction length, 1 transaction, write probability of 0.9, task period chosen from $[1ms, 1000ms]$.

Fig. 2: Schedulability results for scenarios with a range of TSA improvement ratios.

Favoring the competition. Retry-based STM implementations can choose different methods for prioritizing the completion of certain transaction types. As such, to determine the most accurate schedulability results would require per-STM retry-based analysis. For example, TL2 prioritizes the completion of write transactions; transactions that read an object may be forced to retry repeatedly until the read (and transaction maintenance, such as checking a read-version clock) do not overlap with any write to that object. As such, the retry-based curve that ignores read transactions is very optimistic for TL2.

Additionally, for retry-based STM approaches that employ locking, both blocking and retrying may contribute to the synchronization delay. For example, TL2 requires a lock to be acquired in order to perform a write, but even after the lock is acquired, the transaction may fail. In this case, the lock would need to be reacquired before attempting the transaction again, possibly causing blocking in addition to the retry cost. This again points to the need for per-STM analysis and highlights that only accounting for retry delay in the schedulability analysis may be generous.

Finally, as described in the experimental design, we used the same transaction length for both R^{2}STM and the retry-based approaches. By doing so, we are favoring retry-based STM. A transaction handled by R^{2}STM simply reads or writes its set of elements. A retry-based STM implementation, on the other hand, must maintain metadata in order to detect overlapping transactions, and abort and retry transactions. For example, TL2 simulates a transaction's execution, acquires write locks, and checks the simulated values before writing the values and releasing write locks. In practice, this extra bookkeeping could add significant time to the execution of each transaction (and any required retries of that transaction). Accounting for this added work would increase the retry cost of each transaction under retry-based STM approaches. We chose to ignore this work to make our evaluation independent of such implementation choices, and instead focus on retry-free vs. retry-based analysis trade-offs.

Schedulability results. Considering all possible combinations of task-set parameters results in 432 scenarios. For each one, we generated task systems with a number of tasks between $\{8, 12, .., 80\}$. As tasks were generated, they were added to each partition in turn. For each number of tasks, we generated 1,000 independent task systems. We plot the *schedulability ratio* of these, which is computed by taking the ratio of tasks systems that were schedulable by each approach out of the 1,000 systems generated with the given parameters. To summarize the relative performance of approaches under a given scenario, we compute for each approach its *task schedulable area* (TSA), which is the area under its schedulability curve as computed with a midpoint sum. We compare curves on the basis of TSA, and we report that two approaches performed equally if their TSA was within 0.5 of each other.

Each scenario resulted in one graph. In Fig. 2, we present three of these 432 scenarios that are representative of our key observations. While each plot is with respect to the number of tasks, higher task counts generally yield an increase in system utilization and resource contention. With sufficiently high utilization, task systems are not schedulable. The NOLOCK line in each plot represents P-EDF schedulability of independent tasks, including no synchronization delays. A task set that is not schedulable under NOLOCK cannot be schedulable after synchronization interference is accounted for, and is plotted to show the utilization loss associated with synchronization.

From all of our results, we make the following observations.

Obs. 1 *The retry-free approach resulted in a higher TSA than the baseline retry-based approach in 91.9% of scenarios.*

In the remaining scenarios, these two approaches were tied. This observation is illustrated in Fig. 2. Recall that we did not charge overheads for retries (other than accounting for actually re-running the transaction); accounting for these would only serve to further reduce the TSA of the retry-based approach. Fig. 2a depicts a scenario with significant TSA improvement under the retry-free approach compared to the retry-based approaches (117.3% over the baseline retry-based approach and 84.9% over the approach that ignores read transactions). On average, the retry-free approach resulted in a 95.3% TSA improvement over the baseline retry-based approach. Fig. 2b depicts a scenario in which the retry-free approach has a 96.8% TSA improvement.

Obs. 2 *In scenarios with higher write probability, the retry-free approach results in higher schedulability than the retry-based approach that ignores read-only transactions.*

In 73.1% of scenarios with write probability of 0.75 or higher, the lock-based STM resulted in a higher TSA than the retry-based approach that ignores reads, and in 22.2% of those scenarios, the two approaches were tied.

Obs. 3 *Fully accounting for read-only transactions may improve schedulability under retry-based approaches, but it is unlikely that such improvements can cause such approaches to be preferable to retry-free approaches.*

In Fig. 2, analysis that appropriately incorporates read-only transactions would likely result in a curve between the two retry curves. Even entirely ignoring read-only transactions results in a TSA that is usually at most that of the retry-free approach. On average the retry-free approach results in a 7.8% TSA improvement over the retry-based approach that ignores read transactions. Fig. 2c illustrates a scenario in which that improvement is 7.5%.

These observations motivate the need for retry-free STM for real-time applications in which schedulability is the principal design consideration. Next we demonstrate how the same information necessary to compute retry bounds (*i.e.*, the conflict set of each transaction) can also be used at compile time to insert locks that prevent the need for ever retrying transactions.

IV. TORTIS

Here we discuss the design and implementation of an R^{2}STM in Rust, TORTIS. We discuss both correctness, *i.e.*, no deadlock, and performance considerations. R^{2}STM inherently trades off throughput for the schedulability gains demonstrated in the previous section. We show that it is possible to mitigate throughput losses compared to optimistic STM systems when designing an R^{2}STM system.

The key challenge in designing an R^{2}STM system is to reliably prevent any circumstance that would require a transaction to be aborted. While previous lock-based STM approaches [19], [23], [26] prevent many such circumstances by requiring that locks be acquired before results are committed, they require aborts to resolve deadlock (among other reasons). Deadlock freedom is therefore the quintessential requirement for an R^{2}STM system.

For this first R^{2}STM, TORTIS, all transaction synchronization is performed using coarse-grained *group locks* [7]—one lock is acquired before, and released after, a transaction, rather than on a finer-grained basis based on the objects actually accessed at runtime. Thus, the one lock must guard all objects potentially accessed within the transaction, regardless of the code paths exercised, and must also be locked by any conflicting transaction. The most naïve such approach is to use the *same* lock for all transactions, as in the infamous Linux Big Kernel Lock of years past. In contrast, TORTIS performs static analysis to identify and lock separate, non-conflicting resource groups, which helps mitigate throughput losses. We leave exploration of finer-grained locking to future work.

```
fn main() {
  let radar = /* shared object */;
  let lidar = /* shared object */;
  let 3D_model = /* shared object */;
  let plan = /* shared object */;

  for i in 1..sensing_threads{
    let t1 = new_thread(move || {
      let my_sensor = null;
      if (...){
        my_sensor = radar;
      } else {
        my_sensor = lidar;
      }
      transaction /* A */ {
        /* read only */
        read_sensor(my_sensor);
      }
      ...
      transaction /* B */{
        /* write */
        update_model(3D_model);
      }
    });
  }
  for i in 1..planning_threads{
    let t1 = new_thread(move || {
      transaction /* C */{
          /* read only */
          get_position(3D_model);
          get_plan_component(plan);
      }
      /* compute component of plan */
      ...
      transaction /* D */ {
        /* write */
        update_plan(plan);
      }
    });
  }
}
```

Listing 2: Transaction example using vehicle path planning.

A. Static Analysis for Group Locks

Example scenario. To illustrate the principles of our static analysis, we provide a running example in Lst. 2 drawn from our motivating use case in Sec. I of an autonomous car performing motion planning for parking. Note that this is *not* a real system, but simplified and abstracted to highlight important design considerations for TORTIS and illustrate our analysis techniques. The simplified system has two sensors, represented by the `radar` and `lidar` shared objects. It uses these to build a model of the world around it, stored in

3D_model, which is shared between the sensing and planning threads. Based on this model, the car computes a plan for how to park itself, operating on the plan data structure. There are two sets of threads that perform computation in parallel: the sensing_threads, which use a transaction to read the radar and lidar shared objects, and another transaction to write to the 3D_model, updating it. The planning_threads use a read transaction to query the 3D_model. They then perform partial computation for the parking plan, and update the shared plan with their results in a write transaction.

Conflict-set analysis. Which shared object(s) a transaction uses, *i.e.*, its conflict-set, determines which resource group that transaction is in. As resource groups control the emitted locks and potential runtime concurrency, this analysis is critical for TORTIS's runtime performance.

In Lst. 2, transaction A may use either the radar and lidar shared objects, while transaction B uses the 3D_model shared object, transaction C uses the 3D_model and plan shared objects, and transaction D uses the plan shared object. TORTIS must recover this transaction-to-shared-object mapping through static analysis. We start by assigning both transactions and shared-objects identifiers, based on the line of code they begin at or site at which they are allocated, respectively. Using allocation sites to identify objects is the standard approach [32], [37], [40], [41], which we extend to identifying transactions. In Lst. 2 the transactions begin at Lines 15, 20, 28, and 35, and the shared objects are allocated at Lines 2-5.

Having identified the shared objects, we next must determine which transactions access each resource. Statically, a shared object may be used in a transaction if, after it is allocated, there is a sequence of instructions in the program that could lead to it being read or written inside the transaction. We rely on a context-sensitive, flow-insensitive data-flow analysis, similarly to Schoeberl et. al. [37], to map shared objects to transactions, determining transactions' conflict sets.

Conflict sets are inherently conservative, representing the union of all shared objects that *may* be used by a transaction on any path through the program. For instance, in Lst. 2, only one of radar and lidar will be used in transaction A at runtime. Despite this, because both *may* be used in transaction A, TORTIS must include both radar and lidar in the conflict set for the transaction. Programmers can help mitigate this conservatism through the placement of transactions, as discussed later. We leave exploring the trade-offs of dynamic techniques that enable greater parallelism by only blocking on the shared objects used at runtime as future work. Such approaches may require more invasive runtime instrumentation, trading per-thread overhead for increased concurrency.

Creating resource groups. Once TORTIS determines the set of shared objects that may be used by each transaction, it creates resource groups and assigns each transaction to a group.[4] TORTIS assigns two transactions to the same resource group if either: (i) they directly conflict, *i.e.*, the intersection of their conflicts sets is not empty, or (ii) they transitively conflict. To see why the transitive closure over conflicting transactions is required, consider the conflicts between transactions in Lst. 2. Transactions B and C directly conflict, as do transactions C and D. Consequently, when transaction C executes, neither B nor D can execute concurrently. As TORTIS uses one lock per transaction, all three transactions must be assigned to the same lock, and thus resource group. This does limit concurrency as B and D could otherwise execute in parallel, but as we show in Sec. III, the predictability of retry-free execution far outweighs the non-determinism of retry-based approaches, even if they enable greater average-case parallelism.

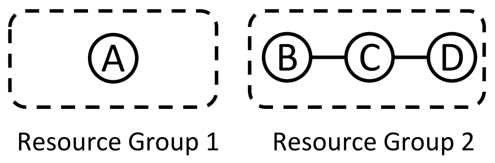


Fig. 3: Conflict-set graph for transactions in Lst. 2. Transactions are nodes, and are connected by an edge if their conflict sets intersect.

Determining resource groups is equivalent to computing connected components in a graph, where transactions are nodes, and an edge exists between two transactions if the intersection of their conflict sets is not empty. We call this the *conflict-set graph*. Connected components in the conflict-set graph represent resource groups, as elements are connected if and only if they transitively access a common object. For the program in Lst. 2, Fig. 3 shows the resulting conflict-set graph. Transactions B and C both access the 3D_model object, and so are connected. Transactions C and D are connected via the plan shared object. Only transaction A accesses the radar or lidar objects, so it is not connected to any other transactions. Consequently, transaction A is placed alone in Resource Group 1, while transactions B, C, and D form a connected component and are placed in Resource Group 2.

Reader/writer locks. As a further optimization, TORTIS supports reader/writer locks. A transaction that only reads shared objects is deemed a read transaction, and all other transactions are deemed write transactions. Transactions that perform a mix of reads and writes are deemed to be write transactions. Our read-write analysis does not change resource grouping, and only requires that read-only transactions are modified to use a read lock, while all other transactions use a write lock. In our example in Lst. 2, transactions A and C are read only, while B and D perform writes to shared objects. More sophisticated reader/writer analysis based on the exact set of shared objects read vs. written would require multiple locks per transaction, and so is out of scope.

[4]Because we assume each transaction is protected by one lock, the resources in a group correspond directly to the transactions that use the same group lock. As such, we refer to the resource groups as representing the resources themselves and the transactions that use those resources interchangeably.

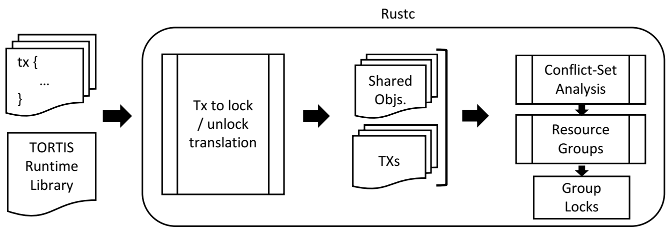


Fig. 4: Overview of the TORTIS compiler extensions within Rustc.

B. Implementation

Fig. 4 illustrates the TORTIS implementation. Our implementation targets Rust programs, allowing us to leverage Rust's type system, and bypass the question of alias-analysis precision. We believe similar results can be achieved with MISRA-C [33] and AUTOSAR [24], but do not implement or evaluate R^{2}STM for those C dialects. To achieve our design goals, we first introduce a runtime library with new types that integrate our R^{2}STM system with Rust's type system. We also introduce syntax for transactions, in effect our R^{2}STM API. Transactions are initially converted by the compiler to lock and unlock calls using a single global lock. TORTIS then optimizes these locks to resource-group locks, using the analysis described above in subsequent compiler passes. In brief, our implementation consists of the following, which we next describe in more detail:

1) **Import TORTIS runtime library.** Programmers must import the TORTIS runtime library, which consists of: (i) wrapper types for shared objects to allow TORTIS to provide synchronization for shared objects while complying with the Rust type system, and (ii) contains our lock implementation.
2) **Recognize transactions.** Transactions are translated to matched lock and unlock calls to a single lock by the compiler, *i.e.*, they are syntax sugar. At this stage we have a naïve but correct R^{2}STM system.
3) **Optimize locks.** TORTIS next optimizes the transaction locks to resource-group locks, thereby providing retry-free, concurrent transactions as described above.

Import TORTIS runtime library. While most of the TORTIS implementation is based on extensions to the Rust compiler, we also developed a TORTIS library. In particular, this library provides the wrapper types `TxCell` and `TxPointer` for the Rust type system to ensure that shared objects can be shared and mutated across threads, with synchronization provided by TORTIS. In particular, our wrapper types implement *traits* that attest to the Rust type system that correct synchronization mechanism exists. Further, these types allow the compiler's STM system to: (i) provide synchronization within transactions, and (ii) verify that shared objects are not used outside of transactions. Other retry-based Rust STM libraries [5], [30] use similar wrapper types to satisfy the Rust type system while providing a custom synchronization mechanism.

We note that only data wrapped in the `TxCell` or `TxPointer` type is protected by TORTIS. This does not imply a loss in generality as these wrappers can wrap arbitrary data types. Further, the programmer can avail themselves of any synchronization method for shared objects not wrapped in `TxPointer`, albeit without the guarantees of the TORTIS STM system. For example, if a transaction acquires a Rust `Mutex`, this may lead to deadlock due to dependencies on non-transactional code. Therefore, non-TORTIS synchronization is not advisable to include in transactions, but can be used alongside transactions at the programmer's discretion.

In addition to the wrapper types, the TORTIS library contains our lock implementation. Putting the lock implementation inside the library rather than within the compiler itself allows the library implementer to change the lock implementation without rebuilding the compiler, thereby providing greater flexibility to experiment with lock implementations. The locks are accessed via `lock()` and `unlock()` function calls that take a single argument—an integer specifying which lock to manipulate. This design opens up the possibility of a future API to specify the desired lock implementation for an application, *e.g.*, to allow the programmer to specify if a spin- or suspension-based lock should be used. The current TORTIS prototype uses a phase-fair reader/writer lock [9].

Recognize transactions. TORTIS adds the `transaction` keyword to Rust, as shown in Lst. 2, *e.g.*, at Line 14. Each transaction is converted into a pair of lock/unlock calls into our runtime library. The lock call is placed at the beginning of the transaction's scope, and the unlock call at the end. The lock calls initially reference a single lock as resource groups have not yet been determined. As such, immediately after replacing the `transaction` keyword with lock/unlock calls, a naïve retry-free STM system exists. However, this naïve system provides no concurrency.

Optimize locks. To enable concurrency in TORTIS, we optimize the initial lock calls to refer to resource-group locks instead of a single global lock. As described above, doing so first requires performing conflict-set analysis on the transactions. To determine what shared objects reach a transaction, we use a context-sensitive, flow-insensitive analysis that TORTIS adds to the Rust compiler.

Once it has computed the per-transaction conflict-sets, TORTIS creates a matrix representing the conflict-set graph described above, where nodes are transactions and edges represents conflicts that require serialization. Given this graph, TORTIS computes the connected components and designates each such component as a resource group. Each resource group in turn is assigned a unique lock.

Once the resource groups are determined, the last step is to update the naïve global `lock()` and `unlock()` calls in individual transactions to reference the resource-group lock instead. As per our runtime library design, the lock is specified as an argument. Thus, changing the lock used by a transaction only requires updating the argument to the API call. At this stage, lock calls are promoted to read-only if possible.

C. Discussion

R^{2}STM is motivated by worst-case real-time performance, not throughput, in contrast to prior STM implementations.

Emphasizing worst-case behavior and schedulability leads to trade-offs around the granularity of synchronization, and elevates the importance of where programmers place transactions. We also highlight how programmers can use the Rust type system to provide hints to the TORTIS analysis.

Granularity of synchronization. TORTIS uniquely identifies objects based upon their allocation site. In a list, for example, TORTIS often cannot differentiate list elements from one another, as they are all allocated in a common function. Thus, TORTIS provides coarse-grained synchronization for such objects, and will effectively treat the entire list (or other multi-object aggregate structures) as one object. This limits opportunities for concurrency and thus average-case throughput. However, we note that for the purpose of schedulability analysis, retry-based STM systems often cannot guarantee that two transactions operating on a common structure will not conflict. For example, even if two transactions do not conflict in a list, if this cannot be determined statically, then schedulability analysis must assume they conflict. Consequently, TORTIS trades off potential parallelism for schedulability.

Transaction placement. TORTIS's context-sensitive analysis favors some software design patterns over others. Consider a helper function `Foo` that takes argument(s). If a transaction is placed inside of the `Foo` implementation, any shared object `Foo` *may use* will be forced into a single resource group. If instead, `Foo` is called within multiple transactions, each transaction could lock a different resource group. Thus, deeply embedding transactions could inadvertently cause non-conflicting transactions to be grouped together.

Leveraging the type system. The type system can also be employed to help statically rule out conflicts. If the `Foo` function is generic, and takes arguments of multiple types, then during compilation Rust will generate one version of `Foo` per type, and TORTIS will separately analyze each instance. Consequently, each implementation may be assigned to a unique resource group, despite there being one shared definition in source code. Therefore, type aliases (*e.g.*, `typedefs`) can be used as hints to TORTIS to minimize conflicts.

Transactions with I/O. R^2STM enables I/O in transactions because transactions never abort. This is supported by TORTIS. Performing I/O in a transaction requires wrapping the I/O object, *e.g.*, File, in a `TxPointer`. The object is then accessed via the wrapper as with any other shared object.

V. Throughput Comparison

The primary objective of TORTIS is to increase schedulability. However, to enable these schedulability gains, some potential concurrency is sacrificed by using coarse-grained locking instead of more optimistic synchronization. In this section, we evaluate these tradeoffs with respect to concurrency and throughput as compared to TL2 [19], a lock-based STM system that leverages retries. We compared TORTIS against TL2 for five main reasons: (i) we are not aware of any open-source *software* TM implementations designed for real-time workloads to test against; (ii) like TORTIS, TL2 is lock-based, though it does allow for retries; (iii) TL2 is a well-established algorithm (over 1,000 citations); (iv) measurement-based studies of some prior real-time (retry-based) STM approaches have been shown inferior to lock-based synchronization [21]; and (v) it has an open-source Rust-based implementation [30], allowing us to conduct performance comparisons without the confounding variable of language choice. We examined throughput on both branching and linear data structures, and generally found that when workloads offer the potential for high concurrency, TL2 enables greater throughput, but when more transactions conflict at runtime, the lower overheads and no retries in TORTIS enable greater throughput.

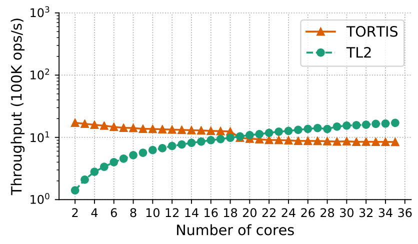
Experimental design. STM is most commonly evaluated based on *observed average-case throughput*. We performed standard experiments from the STM literature to compare performance between TORTIS and TL2, which was optimized for throughput. We also compared to a modified TL2 implementation that locks objects in order, an optimization that increases predictability at the expense of average-case throughput. We found this led to a decrease in throughput in all scenarios by up to 70%, and thus focus the rest of our discussion on the less predictable original version. TORTIS used a low-overhead implementation of a phase-fair reader/writer lock [8].

All experiments were run on a two-socket, 18-cores-per-socket x86 machine running the Linux 4.9.30 LITMUS^{RT} kernel [1] with two Intel Xeon E5-2699 v3 CPUs @ 2.30GHz with 128GB RAM and 32KB L1, 256KB L2, and 46080KB L3 caches. We performed the evaluations on up to 36 cores, with the first 18 cores on the same socket. We used P-EDF scheduling as provided by LITMUS^{RT}, and guaranteed non-preemption by allowing only one thread to execute on each core with no other concurrent workloads. Note that all throughput figures are presented on logarithmic scales to better highlight trends. Experiments in this section consider a single concurrent data structure as is common in STM evaluations. The impact of resource grouping among multiple objects is highlighted in the next section, which considers a case study with multiple shared data structures.

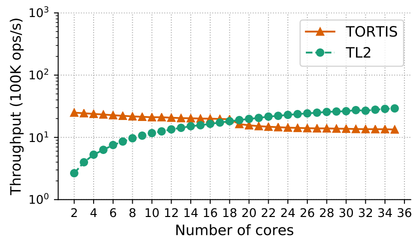
The implementation used with TL2 allows for concurrent non-conflicting writes on the tested data structures while TORTIS does not. Our motivation behind the following experiments was to evaluate the throughput implications of this tradeoff as a function of the amount of potential concurrency in the workload. Towards that end, we varied read/write ratios, as well as the fraction of the data structure accessed within the transaction, for both a branching and a linear data structure.

A. Branching Benchmarks

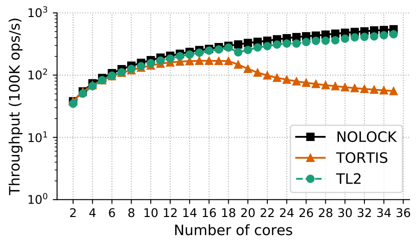
For a branching data structure, we evaluated TORTIS and TL2 on a large red-black tree and measured how inserts/lookups per second scale with thread/core counts. We used a standard Rust-based red-black-tree implementation for TORTIS, and declare the entire tree, not individual nodes, as one shared object. For TL2, we used a red-black-tree implementation optimized for high throughput provided as an extension of the Rust-based TL2 implementation [30].



(a) 100k inserts.



(b) 50k lookups, 50k inserts.



(c) 100k lookups.

Fig. 5: Average throughput for varying access patterns in a red-black tree.

For inserts, we created an array of 100k integers, shuffled the array, and then inserted each value as a node, measuring the time it took for all inserts to succeed. For lookups, we first created a random tree, and then timed how long it took to find every element in the tree in random order. We also evaluated 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) was used to determine whether a transaction is a lookup or an insert. A third of the tree was pre-built to prevent abnormally fast lookups at the beginning of the experiments. On this partially pre-built tree, we timed 50k lookups and 50k inserts.

All work across the experiments was partitioned evenly over the number of cores, with each thread performing an equal fraction of the work. For each experiment, we averaged the throughput for ten unique random trees with unique lookup and insert orders. The results are shown in Fig. 5.

Obs. 4 *For workloads with more inserts,* TORTIS *yields higher throughput on one socket and* TL2 *yields higher throughput on two sockets.*

Insets (a) and (b) of Fig. 5 have 100% and 50% writes, respectively. These write-heavy workloads have less potential concurrency. In these cases, all operations were serialized in TORTIS, while TL2 allowed non-conflicting updates to occur in parallel. However, the extra overhead of maintaining transactional state to enable retries significantly reduced throughput for smaller core counts, enabling the lock-based in-place approach in TORTIS to provide increased throughput on one socket.

Obs. 5 TORTIS *lookup throughput is limited by memory contention due to the locking-protocol overhead in scenarios with high read parallelism.*

Fig. 5(c) shows a peculiar trend: for only lookups, TORTIS does not scale to two sockets (more than 18 cores), despite using a phase-fair reader/writer lock for a read-only workload. We were surprised that the lock did not scale better given that there is *no blocking*. We suspect this is due to high memory contention incurred by the lock implementation itself. To test this theory, instead of running 100k lookups within individual transactions, we ran an additional experiment with only one read-only transaction per thread to eliminate any lock-related memory contention, and observed the NOLOCK line behavior depicted in Fig. 5(c), which scaled as expected.

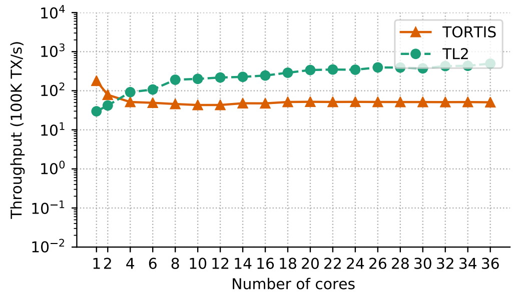


Fig. 6: Varying core count. 64 elements, 2% accessed, 5% writes

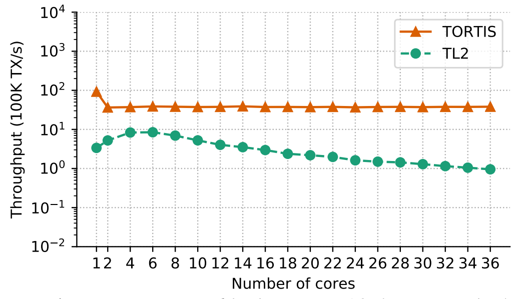


Fig. 7: Varying core count. 64 elements, 10% accessed, 5% writes

The difference between NOLOCK and TORTIS is due to overheads introduced by lock-related memory contention. In contrast, TL2 merely checks that no writes have occurred, and such state may be cached requiring no memory references. In the future we plan to use a new phase-fair reader/writer lock that is optimized for read performance to further improve TORTIS throughput [34].

B. Linear Benchmarks

The other major class of data structures commonly evaluated by STM systems are linear data-structures, *e.g.*, buffers. To explore the comparative throughput performance of TORTIS and TL2 on buffers we constructed a set of synthetic benchmarks. These benchmarks allowed us to perform fine-grained experiments on different access patterns and buffer lengths to observe their effect on throughput, along with variations in read-write workloads, and core count for scalability.

Experimental design. We considered transactions operating on fixed-sized buffers with 64 elements and varied three parameters: (i) core and thread counts, (ii) percent of elements in the buffer accessed, and (iii) percent of element accesses that were writes.

Multiple threads read and write a single shared buffer. Each transaction accesses a given percentage of randomly chosen buffer elements per loop iteration. Some percentage of these accesses are writes, while the rest are reads. For all benchmarks, the accessed elements are randomly determined before the transactions execute to exclude the overhead of determining this in the transaction measurement time.

Core counts. In our first experiments, we varied the number of cores to evaluate how both STM systems scale. We evaluated a small 64-element buffer, kept the percent of writes constant at 5%, and tested multiple access patterns.

Obs. 6 *For workloads with few runtime conflicts, TL2 outperforms TORTIS by up to 9.8× and exhibits throughput scaling with increasing core counts. Conversely, for lower core counts, TORTIS outperformed TL2 by up to 6× but did not scale.*

Fig. 6 shows a case in which only one or two elements of the buffer is accessed, and then only written once every 20 accesses. In this case, the likelihood of transactions conflicting at runtime is low, and there are greater opportunities for runtime concurrency. TL2 enables such runtime concurrency, while the coarse-grained locking in TORTIS prevents it. As such, TORTIS's throughput strictly decreases as the number of cores increases, and lock-related overheads increase. For lower core counts, TORTIS outperformed TL2 due to lower overheads for acquiring locks compared with the additional blocking required to support retries.

Obs. 7 *For workloads with less potential runtime concurrency due to more likely conflicts from more elements being accessed, TORTIS outperforms TL2 by up to 40× due to lower overheads for resolving conflicts.*

As highlighted by Fig. 7, with an increase to 10% of elements accessed versus 2% in Fig. 6, TORTIS outperformed TL2 for all core counts while maintaining consistent throughput. This is due to the additional retries TL2 experiences as writes are more likely to conflict, increasing overheads.

Obs. 8 *TORTIS throughput is less sensitive to changes in access patterns than TL2 throughput.*

While TORTIS throughput remained fairly constant between Fig. 6 and Fig. 7, TL2 throughput changed drastically. The change in the percentage of elements accessed from 2% to 10% yielded an average 97% decrease in throughput for TL2, while TORTIS decreased by an average of only 26%.Motivated by Obs. 7 and 8, we next evaluate the effects of the number of elements accessed on throughput.

Obs. 9 *TL2 has higher throughput for transactions accessing up to 5% of elements, but has decreasing throughput as element access increases. In contrast, TORTIS remains relatively constant as the number of elements accessed increases.*

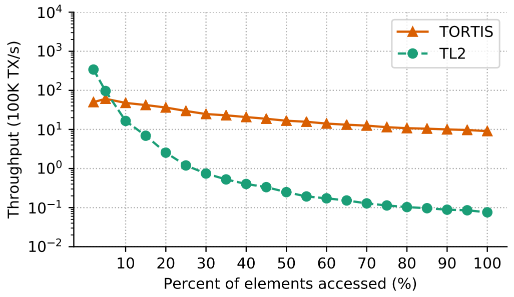


Fig. 8: Varying elements accessed. 64 elements, 2% writes, 8 cores

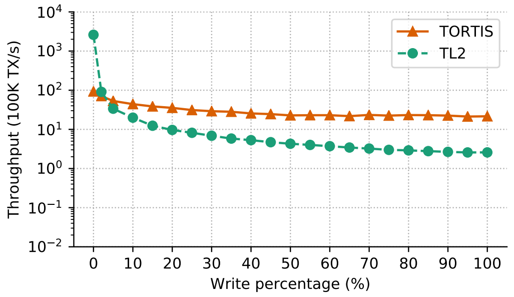


Fig. 9: Varying writes. 64 element buffer, 5% accesses, 8 cores

Our access pattern experiment uses the same buffer size with 2% writes, and the results are in Fig. 8. Increasing the number of elements accessed decreases the potential runtime concurrency for TL2 to exploit and so degrades its throughput, but has little effect on TORTIS.

Obs. 10 *TORTIS outperforms TL2 on workloads with >5% writes, which have less potential concurrency.*

To explore the effect of read/write concurrency, we varied write percentages on a 64-element buffer with 5% of elements accessed per transaction on 8 cores. The results are shown in Fig. 9. TL2's throughput advantage over TORTIS decreases as the write percentage increases. Increasing the number of writes at runtime increases potential conflicts, thereby decreasing potential concurrency. The decrease in potential concurrency reduces TL2's throughput but has little effect on TORTIS.

Combining Obs. 9 and 10, we see that increasing either the number of elements accessed or the number of elements written decreases the potential concurrency available by making conflicts between transactions more likely. The reduction in potential concurrency in turn degrades TL2's performance relative to TORTIS. This relationship is seen in both Fig. 8 and Fig. 9, where TL2 initially outperforms TORTIS by over an order of magnitude for all reads. However, TORTIS has higher throughput for write percentages over 5% when 5% of the data structure is accessed.

Summary. These observations demonstrate fundamental tradeoffs for retry-free STM. For workloads with high concurrency potential due to infrequent writes or only accessing a limited portion of a data structure, optimistic retry-based STM offers greater average-case throughput by as much as 9.8×. Notably, however, the potential for retries significantly reduces schedulability as shown in Sec. III. Conversely, TORTIS

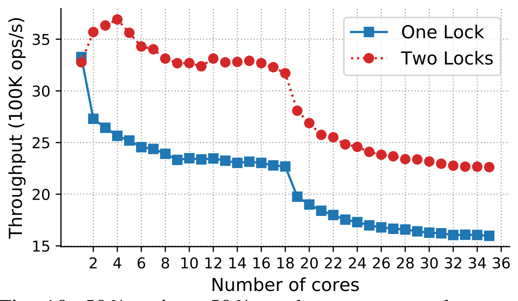


Fig. 10: 50% writes, 50% reads on a tree and a queue

performs comparably or better for workloads with moderately less concurrency potential, due to lower core counts, which are common in embedded systems, higher write percentages (*e.g.*, >10%), or access patterns more likely to conflict (*e.g.*, >5%). We believe these results demonstrate TORTIS offers reasonable throughput, while enabling significantly improved schedulability in most cases over retry-based STM.

VI. Resource-Grouping Case Study

To demonstrate that TORTIS's resource-grouping implementation can recognize different resource groups, and that doing so increases throughput, we present a case study with two classes of transactions: one that operates on a red-black tree, and one that operates on a queue. We consider two program structures, one with two transactions in the source code, with a switch statement choosing which transaction to execute, and another in which the switch statement is part of a single transaction. The former cases yields two resource groups while the later only one.

Experimental design. We spawn one thread per core. each thread contains a loop over a set of transactions, and randomly selects with equal probability whether it reads or writes, and on the red-black tree or the queue.

Obs. 11 *Separate resource groups increase throughput by 38% in our case study.*

Fig. 10 shows the results of this case study. For all but one core, separate resource groups (Two Locks) outperforms a single group (One Lock). We note that the throughput of both versions declines after 18 cores due to the additional cross-socket memory-contention overhead. This demonstrates that our analysis was able to distinguish the red-black tree from the queue, and use different locks for non-conflicting transactions, even when contained in the same thread.

VII. Conclusion

We have presented a new class of STM, R^{2}STM, and TORTIS, the first R^{2}STM implementation. R^{2}STM is designed for real-time applications, emphasizing both automatic synchronization and schedulability. It offers the potential to be a certified STM framework, which could greatly ease the development of safety-critical real-time systems. We showed that R^{2}STM has an average schedulability improvement of 95.3% compared to retry-based STM. Our throughput evaluations demonstrate that while TORTIS may not enable as much concurrency as optimistic retry-based STM, it achieves reasonable average-case throughput in many cases, while enabling greater schedulability.

References

[1] "LITMUS^{RT} home page," http://www.litmus-rt.org/.

[2] "SchedCAT: Schedulability test collection and toolkit," https://github.com/brandenburg/schedcat, 2019, accessed: 2020-06-21.

[3] A. Barros, L. Pinho, and P. Yomsi, "Non-preemptive and SRP-based fully-preemptive scheduling of real-time software transactional memory," *Journal of Systems Architecture*, vol. 61, no. 10, pp. 553–566, 2015.

[4] C. Belwal and A. Cheng, "Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective," *Embedded Systems Letters*, vol. 3, no. 1, pp. 37–41, March 2011.

[5] G. Bergmann, "rust-stm," https://github.com/Marthog/rust-stm/, 2020, commit 74e959d.

[6] 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.

[7] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, August 2007, pp. 71–80.

[8] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina, Chapel Hill, NC, 2011.

[9] B. Brandenburg and J. Anderson, "Reader-writer synchronization for shared-memory multiprocessor real-time systems," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.

[10] ——, "Optimality results for multiprocessor real-time locking," in *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010, pp. 49–60.

[11] ——, "Spin-based reader-writer synchronization for multiprocessor real-time systems," *Real-Time Systems*, vol. 46, no. 1, 2010.

[12] ——, "The OMLP family of optimal multiprocessor real-time locking protocols," *Design Automation for Embedded Systems*, vol. 17, no. 2, pp. 277–342, 2014.

[13] B. B. Brandenburg, *Multiprocessor Real-Time Locking Protocols*. Singapore: Springer Singapore, 2020, pp. 1–99.

[14] V. Chaudhary, S. Kulkarni, S. Kumari, and S. Peri, "Starvation freedom in multi-version transactional memory systems," 09 2017.

[15] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley *et al.*, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," *Real-Time Systems*, vol. 55, no. 1, pp. 144–207, 2019.

[16] S. Cherem, T. Chilimbi, and S. Gulwani, "Inferring locks for atomic sections," ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 304–315.

[17] S. Cotard, A. Queudet, J.-L. Béchennec, S. Faucou, and Y. Trinquet, "Stm-hrt: A robust and wait-free stm for hard real-time multicore embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 4, pp. 1–25, 2015.

[18] G. V. T. B. d. Cunha, "Consistent state software transactional memory," Ph.D. dissertation, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2007.

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

[20] E. Dijkstra, "Two starvation free solutions to a general exclusion problem," EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands.

[21] M. El-Shambakey, "Real-time software transactional memory: Contention managers, time bounds, and implementations," Ph.D. dissertation, Virginia Polytechnic Institute, Blacksburg, VA, 2013.

[22] M. El-Shambakey and B. Ravindran, "STM concurrency control for embedded real-time software with tighter time bounds," in *Design Automation Conference 2012*. IEEE, 2012, pp. 437–446.

[23] R. Ennals, "Software transactional memory should not be obstruction-free," Intel Research Cambridge Tech Report, Tech. Rep. IRC-TR-06-052, January 2006.

[24] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "AUTOSAR–A worldwide standard is on the road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, 2009, p. 5.

[25] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero, "Transactional memory: An overview," *IEEE micro*, vol. 27, no. 3, pp. 8–29, 2007.

[26] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05, 2005, p. 48–60.

[27] J. Havender, "Avoiding deadlock in multitasking systems," *IBM systems journal*, vol. 7, no. 2, 1968.

[28] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*. ACM, July 2003, pp. 92–101.

[29] M. Herlihy and J. 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.

[30] T. Kopf, "swym," https://github.com/mtak-/swym, 2019, commit f7b635d.

[31] B. McCloskey, F. Zhou, D. Gay, and E. Brewer, "Autolocker: Synchronization inference for atomic sections," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 346–358.

[32] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[33] C. MISRA, "Misra c," 2003.

[34] C. Nemitz, S. Caspin, J. Anderson, and B. Ward, "Light reading: Optimizing reader/writer locking for read-dominant real-time workloads," in *Proceedings of the 32nd Euromicro Conference on Real-Time Systems*, 2021.

[35] A. Quillet, A. Queudet, and D. Lime, "Analysis of polka contention manager for use in multicore hard real-time systems," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, 2020, pp. 11–21.

[36] 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.

[37] 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.

[38] 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.

[39] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 1995, pp. 204–213.

[40] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[41] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 387–400, 2006.

[42] B. Ward, "Sharing non-processor resources in multiprocessor real-time systems," Ph.D. dissertation, University of North Carolina, Chapel Hill, NC, 2016.

[43] B. Ward and J. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2012, pp. 223–232.

[44] 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.