Retry-Free Software Transactional Memory in Rust

by

Claire Nord
B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engineering in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 14, 2020

Certified by .....
Dr. Howard Schrobe
Principal Research Scientist, MIT CSAIL
Thesis Supervisor

Certified by .....
Dr. Hamed Okhravi
Senior Staff Member, MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Dr. Bryan Ward
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

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.

Retry-Free Software Transactional Memory in Rust

by

Claire Nord

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2020, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Software transactional memory (STM) is a synchronization paradigm that has been well-studied in work on throughput-oriented computing. In that context, its main utility lies in aiding programmers in producing performant concurrent programs that are free of synchronization bugs. With STM, programmers merely annotate code sections that require synchronization; the underlying STM framework automatically resolves how synchronization is done. In work on real-time systems, schedulability is more important than throughput. However, all prior work on real-time STM has been limited to approaches that retry transactions when they conflict. Unfortunately, reasonable retry bounds can be difficult or impossible to obtain for multiprocessors. Perhaps because of this, prior work on real-time STM has focused on observed behavior rather than schedulability. This thesis argues that real-time STM should be lock-based and free of retries in order to provide schedulability guarantees, and presents a real-time STM framework called TORTIS, implemented for Rust programs.

The efficacy of TORTIS is evaluated via both benchmarking experiments and a schedulability study. In the benchmarks, when contention for shared resources is high, causing retry-based approaches to repeatedly retry, TORTIS provides up to 75x improved throughput. The high-contention case is also effectively what schedulability analysis aims to bound. In the schedulability study, TORTIS dominated other prominent retry-based STM approaches, and improved overall schedulability across all task systems generated by on average 2.4x and as much as 10.1x.

Thesis Supervisor: Dr. Howard Schrobe
Title: Principal Research Scientist, MIT CSAIL

Thesis Supervisor: Dr. Hamed Okhravi
Title: Senior Staff Member, MIT Lincoln Laboratory

Thesis Supervisor: Dr. Bryan Ward
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

Thank you to Dr. Jim H. Anderson, Catherine Nemitz, and Shai Caspin at The University of North Carolina at Chapel Hill for collaborating on this project and contributing your depth of experience in real-time systems and the schedulability analysis for the paper submission.

Thank you to Dr. Bryan Ward and Dr. Nathan Burow at MIT Lincoln Lab for being such amazing mentors. Thank you for generously sharing your time, giving insightful feedback, and relentlessly encouraging me. You have helped me grow as both a researcher and a communicator.

Thank you to Dr. Hamed Okhravi at MIT Lincoln Lab and Prof. Howard Schrobe at CSAIL for being supportive group leaders and welcoming me into the Resilient Mission Computer group. I'm grateful for the opportunity to represent the group at the RSA Conference and meet other students in security.

Finally, thank you to my parents for your unconditional love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The multicore revolution has redefined the "common case" computing platform to be a multiprocessor. As a result, concurrency-related issues have become ubiquitous, even in settings where they were previously unimportant. This has created a need for design methods for concurrent software that aids programmers in producing working implementations. Without such methods, the "reality" of multicore may fall short of the "promise" suggested by ever increasing core counts.

This thesis is directed at a particular application domain where the mismatch between the promise and reality of multicore is considerable: *safety-critical real-time systems*. Such systems have timing constraints that require certification. Accordingly, performance is tied to formal analysis pertaining to worst-case system behaviors, in contrast to throughput-oriented systems, where performance is usually tied to the average case and rigorous analysis is not pervasive. When seeking to design a multicore real-time application, many difficult analysis-related certification issues arise. Additionally, even experienced developers may face difficulties in writing correct concurrent code, and the safety properties of such code can be challenging to verify formally.

```
/* Dequeue from one shared queue, enqueue on another */
transaction {
    Item = Queue1[Head1];
    Head1 = Head1 + 1 mod Limit1;
    Queue2[Tail2] = Item;
    Tail2 = Tail2 + 1 mod Limit2;
};
```

Figure 1-1: Pseudo-code showing an example transaction annotation.

1.1 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 throughput-oriented computing, this need has been the driving force behind considerable prior work on a concept called *transactional memory (TM)*, which can be realized in software, hardware, or some combination of the two [24, 40]. As illustrated in Fig. 1-1, when a TM framework is employed, programmers must merely annotate code sections that require synchronization. The TM framework itself determines how synchronization is actually achieved. The goal here is to enable programmers to more easily produce correct performant code.

An especially strong case for TM can be made for safety-critical real-time systems. Certification is time-consuming and expensive—in complex systems, certification can dominate software-development costs, which can approach 50% of total cost [27]. Such costs can be eased by reusing previously certified hardware and software components, including the real-time operating system (RTOS) and any middleware employed. Similarly, the certification process regarding synchronization could be greatly facilitated if a *certified* real-time TM framework were available that enables real-time-related synchronization choices to be automatically and *correctly* resolved. Incorrectly resolving such choices can have disastrous consequences, as evidenced by the infamous case of the Mars Pathfinder, where an ill-defined synchronization parameter almost ended a $265 million mission [28].

1.2 Contributions of This Thesis

The specific focus of this thesis is *software* TM (STM).[1] All prior work on TM (software or not) uses *optimistic* techniques that may cause conflicting transactions to be *aborted* and *retried*. This optimism can enable non-conflicting transactions to execute and commit concurrently, increasing throughput. Such designs are motivated by Lamport's argument that "contention for [shared objects] is rare in a well-designed system" [31], and thus such systems perform well when contention is low.

However, in real-time systems, *bounds* on retries are needed to validate real-time constraints. Such bounds are inherently based on quantifying worst-case contention, rather than the low average-case contention that is likely at runtime and for which retry-based approaches are designed. Furthermore, reasonable bounds for retry-based STM can be difficult to obtain on a multiprocessor. These two observations motivate our fundamental thesis that real-time TM should be retry-free.

Towards this goal, we present the first retry-free STM framework, TORTIS (tryonce real-time STM).[2] In order to eliminate retries, TORTIS relies on lock-based synchronization to manage shared-object accesses. Lock-based STM frameworks have been proposed before [22, 20], but these prior frameworks still resort to using retries in dealing with some transaction conflicts and in mitigating deadlocks.

TORTIS is implemented as extensions to the Rust compiler as well as a runtime library. The compiler extensions statically determine the set of objects that *may potentially* be accessed within each transaction. Using the results of this static analysis, locks can be inserted by the compiler to preserve transaction semantics. While determining the set of objects accessed within a transaction is generally undecidable, advances in type systems in modern programming languages enable more precision in data-flow analysis, *i.e.*, determining the set of objects that can reach a line of code. In particular, eliminating aliasing (multiple references to the same object) has significantly increased the precision of data-flow analysis and keeps it from collapsing and

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

[2] In real-time computing, the predictable tortoise wins the race.

determining that all objects in the program may potentially reach a code location. We base TORTIS upon one such language, Rust.

In addition to unclear retry bounds, another limitation of retry-based TM is that input and output (I/O) should not be performed within transactions, as already-performed I/O cannot be "undone." Some STM systems obviate this issue by exploiting the stronger type systems of languages such as Haskell to statically prevent I/O within transactions [23]. Retry-free TM is immune to this issue and can allow I/O in transactions. This is especially useful in real-time applications, where I/O is used in sensing and actuating the physical world.

Building upon these observations, this thesis makes four contributions. First, we present TORTIS, the first retry-free STM system, and the first to support I/O. Second, we evaluate TORTIS by pitting it against a prior well-known lock-based STM implementation to evaluate inherent concurrency and throughput tradeoffs involving retry-based vs. retry-free STM. Third, we present the first ever (to our knowledge) schedulability analysis for an STM system. Fourth we present the results of a schedulability study that demonstrates the improved schedulability enabled by retry-free STM.

This work supports the following thesis:

> *Retry-free STM is not only possible, but significantly more schedulable than retry-based STM. Therefore, retry-free STM should be used in real-time systems for predictable synchronization.*

1.3 Organization

The rest of this thesis is organized as follows. After providing relevant background in Sec. 2, we present the design and implementation of TORTIS in Sec. 3, followed by a discussion of some of the strengths and weaknesses of this design approach in Sec. 4. Then, in Sec. 5, we present the results of both throughput-oriented benchmark experiments, as well as a schedulability study. We conclude in Sec. 7.

Chapter 2

Background

In this section we provide background on shared-memory concurrent programming, real-time systems, TM, the Rust ownership model and type system as they relate to shared-memory concurrent programming in Rust, the effect that the ownership model has on data-flow analysis in Rust vs. C/C++, and the architecture of the Rust compiler pipeline.

2.1 Shared-Memory Concurrent Programming

Shared-memory concurrent algorithms may be *blocking* or *non-blocking*. Non-blocking algorithms use special hardware-implemented atomic instructions and memory fences to safely manage concurrent accesses to shared objects by preserving invariants about the objects. These algorithms are "non-blocking" because they do not spin or suspend program execution. Some common atomic operations are `Compare_and_Swap` and `Fetch_and_Add` (atomic increment). Non-blocking algorithms give the developer instruction-level control over their program execution, but are difficult to write by hand [19]. Herlihy introduced *universal constructions* in 1991 that can be used to construct a non-blocking concurrent implementation of any object [25], but objects made from universal constructions can be inefficient. For example, universal constructions often impose excessive space and time overhead for larger data structures because they copy object state [3].

In blocking algorithms the developer uses primitives like spinlocks and semaphores to enforce some *mutual exclusion* on shared objects. Mutual exclusion ensures that only one critical section may execute at a time. These primitives are "blocking" because they block program execution while waiting for exclusive access to the shared objects. Spinlocks wait in a loop ("spin") until the lock is available, while semaphores suspend a thread until some condition is satisfied. Spinlocks are implemented by libraries using atomic primitives, while semaphores are implemented by the operating system's scheduler. Locks must be *acquired* before reading or writing shared object(s) and *released* afterward. Blocking algorithms using locks are easier for the developer to use and reason about than atomic instructions, but the programmer still needs to obey lock ordering to prevent *deadlock*. Nonetheless, efficient synchronization is difficult to properly implement in either paradigm.

Incorrectly implemented blocking and non-blocking algorithms may also suffer from *starvation*, where one thread is repeatedly "starved" for CPU time, or *priority inversion*, where a high-priority task is continuously preempted by lower-priority task(s) [14]. A TM system uses blocking and/or non-blocking algorithms to synchronize shared objects, lifting the implementation burden from the developer.

2.2 Real-Time Systems

Real-time systems execute a *task system* (some set of *tasks*, or processes) with timing constraints. Tasks repeat with some *task period* between each invocation. Each invocation is called a *job*. Jobs must finish within some fixed time after they're released (before their deadline). A task system is *schedulable* under a given scheduling algorithm or synchronization protocol if all jobs always meet their deadlines. *Schedulability experiments* generate hundreds or thousands of task systems with different parameters to evaluate an algorithm or protocol's schedulability. For further discussion of real-time systems and schedulability analysis see [12].

2.3 Transactional Memory

Prior work on real-time TM has mainly focused on the development of more predictable *contention managers*, which are mechanisms that are applied to ensure progress when transactions conflict in retry-based implementations. Several previous STM systems (*e.g.*, [4, 45]) 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.* [37] 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.* [38, 39] presented real-time TM for Java chip multiprocessors, and presented necessary static program-analysis techniques to determine retry bounds. Belwal and Cheng [6] investigated the effect of eager vs. lazy conflict detection on real-time schedulability. Note that all prior work on real-time STM and contention management focused exclusively on *retry-based* conflict resolution, which necessitates pessimistic schedulability analysis.

El-Shambakey's [21] runtime experiments measured average deadline-miss ratios using several contention managers plus the OMLP [13] 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). To our knowledge, we are the first to present an analysis-based evaluation of representative real-time STM options.

2.4 Rust

Rust [29] is a modern programming language with a strong type system and several unique features that make implementing a retry-free STM system practical. Here we discuss Rust's thread-safety properties and how to write parallel programs with shared objects within Rust's type system.

2.4.1 Thread Safety

The Rust type system provides a weak version of thread safety. It prevents concurrent accesses to individual shared objects, but multi-object data races leading to deadlock or livelock are still possible. To ensure exclusive access, Rust types that are shared among threads must be marked with either the `Send` or `Sync` traits [5]. These traits attest to the compiler that the objects are thread-safe, *i.e.*, that there is a synchronization mechanism for the data type. Traits are similar to *interfaces* in other languages and tell the compiler about the functionality of a particular type. `Send` objects are passed by value between threads, while `Sync` objects are effectively passed by reference. An STM system must provide `Send` and `Sync`, but also seeks to enable parallel programs to access multiple objects, preventing data races, livelock, or deadlock, thus going beyond Rust's guarantees.

2.4.2 Ownership and Sharing

Rust's type system includes an *ownership model*, which enforces that each object can be assigned to one variable at any point in time, referred to as its *owner*. This ownership model is useful to prevent aliasing, as well as to ensure temporal memory safety (*i.e.*, no use-after-free vulnerabilities). There can be additional references to *borrow* a variable, though only one mutable reference to an object is allowed at a time, and the existence of a mutable reference precludes any immutable references. Consequently, parallel programs that compute on shared objects face two challenges from the Rust type system: (i) sharing ownership across threads, and (ii) sharing mutability among owners.

The first issue is addressed by using the `Arc` data type, which provides *multiple ownership* of the contained object. `Arc` atomically counts references to the data to ensure the memory is freed from the heap when the last reference drops out of scope.

The second challenge is solved by bothwrapping data in a type that provides *interior mutability*, *i.e.*, allows multiple mutable references, and the `Sync` trait discussed in Sec. 2.4.1. A type with interior mutability has an immutable interface, but a mutable internal state. Types that provide interior mutability are based on `UnsafeCell`, the "core primitive for interior mutability in Rust" [2], but must include additional code to preserve Rust's ownership invariants. For example, the `Mutex` type provides interior mutability based on `UnsafeCell` but uses a lock to ensure that at most one mutable reference is live at a time.

The Rust ownership model inherently prevents two concurrent programming bugs. First, the developer cannot access a resource without acquiring its lock. The `Mutex<T>` type takes ownership of `T`, so `T` cannot be accessed without acquiring the lock. Second, the developer cannot forget to release the lock when they no longer need it. `Mutex<T>` implements its own `Drop` trait, which runs when the reference drops out of scope. When a reference to a `Mutex<T>` drops out of scope, its lock is automatically released. The developer can also manually drop the reference (thus releasing the lock) before the end of the scope. While these features do prevent some common bugs, they do not prevent deadlock or priority inversion.

Lst. 2.1 puts shared data in a `Mutex` and then wraps it in an `Arc`, allowing mutable ownership among multiple threads. A key requirement for retry-free STM is achieving this same behavior, but with synchronization provided by the STM system automatically instead of manually using a library (*e.g.*, `Mutex`).

2.5 Data-Flow Analysis

In order to enable retry-free transactions, the set of objects that may potentially be accessed within each transaction, which we call the *conflict set*, must be known at compile time, *i.e.*, statically computed. Doing so requires *data-flow analysis*, static

```
let data = Arc::new(Mutex::new(0))
for _ in 0..N {
    //Clone data to pass to new thread
    let data = Arc::clone(&data);
    thread::spawn(move || {
        //synchronize by locking mutex
        let mut data = data.lock();
        ...
    // the lock is unlocked here when
    // ‘data‘ goes out of scope.
    });
}
```

Listing 2.1: Parallel programming in Rust. Reference counting types must be used to give multiple threads ownership of a shared object. Shared objects must implement either the `Send` or the `Sync` trait, or be wrapped in a type, such as `Mutex`, that provides `Send` or `Sync`.

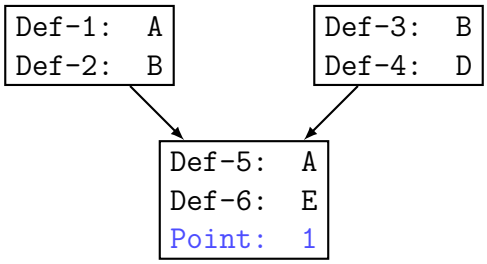


Figure 2-1: Reaching definitions illustrated. Def. 5 of variable A reaches Point 1, as do Defs. 2 and 3 of variable B and the definitions of variables D and E.

analysis of how data objects, or references thereof, flow through a program's control paths.

Data-flow analysis determines all possible shared objects that may *reach* a transaction, *i.e.*, still be assigned to a variable used in the transaction, without an intervening assignment of another object to the variable. Data-flow analysis is inherently conservative as different executions may result in different values reaching a line of code. Fig. 2-1 illustrates this point and shows a simplified version of a program where conditional branches merge back together. Variable B is defined on both branches, and both definitions reach Point 1. Consequently, data-flow analysis must conservatively assume that *either* object can be used at Point 1. Similarly, function arguments that vary based on call site are another source of conservatism for data-flow analysis.

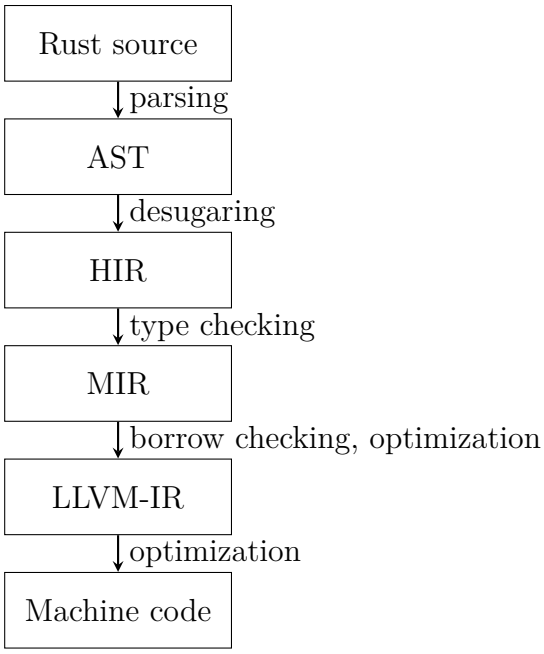


Figure 2-2: The Rust compiler pipeline.

Beyond its inherent conservatism, data-flow analysis runs into several challenges in practice for C/C++. The first concerns mapping out possible control flows to a code location, before even considering data flows along them. Precisely determining the target of indirect control-flow transfers via function pointers or virtual calls in C/C++ is difficult [18]. Thus, data-flow analysis must use an over-approximate set of targets for such calls, potentially introducing spurious data values to the results. Making matters worse, C/C++ allow aliasing, *i.e.*, multiple pointers to the same object. "Points-to" analysis, *i.e.*, determining the object a pointer refers to, is generally undecidable [41, 26], which often makes tracking flows of specific objects impossible. In combination, indirect function calls and aliasing make determining meaningful conflict sets for transactions in C/C++ impossible. In practice, the analysis will collapse with all (or most) transactions marked as conflicting.

Fortunately, modern type- and memory-safe languages like Rust solve most of these problems. Rust's ownership model prevents aliasing, and its strong type system significantly limits possible targets for indirect function calls. While these features were designed largely for security reasons, they also enable more precise data-flow analysis for novel applications such as TORTIS.

2.6 Rust Compiler Pipeline

The Rust compiler, *rustc*, is an LLVM [32] front end that generates multiple intermediate representations (IR) to enable borrow and type checking (borrow checking enforces the ownership model). As illustrated in Fig. 2-2, the Rust compiler turns the original source code into an abstract syntax tree (AST), which is then *lowered* or transformed into the high-level IR (HIR) for type checking, and then mid-level IR (MIR [33]) for borrow checking and other optimizations. MIR is in turn lowered to LLVM-IR, which is then optimized and assembled to the desired architecture by LLVM. The different IRs have different structures, making them more or less suitable for particular analyses and transformations. In particular, the AST and HIR are trees, whereas MIR and LLVM-IR use basic blocks, sequences of non-branching instructions separated by branching instructions. Nodes in the AST/HIR lack a one-to-one mapping with basic blocks, as discussed in Sec. 3.3.

Chapter 3

Design

The key challenge in designing a retry-free TM system is to reliably prevent any circumstance that would require a transaction to be aborted. While previous lock-based STM approaches [20, 23, 22] 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 a lock-based, retry-free STM system.

In this first retry-free STM system, TORTIS, all transaction synchronization is performed using *group locks* [10]: one lock is acquired before, and released after, a transaction, rather than finer-grained model 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 also be locked by any conflicting transaction. While finer-grained locking for retry-free STM may be possible, it is outside the scope of this thesis.

Group locks are based on resource groups, *i.e.*, the set of objects that *may* be accessed within a transaction. Resource groups are determined statically by a data-flow analysis of shared objects to determine which transactions the shared objects reach. With this information, we then perform *conflict analysis* to assign all transactions that may access at least one common shared object to the same group lock. We discern reads from writes in the data-flow analysis to enable read-only transactions and write transactions. Further, conflicts are transitive to guarantee correctness. For

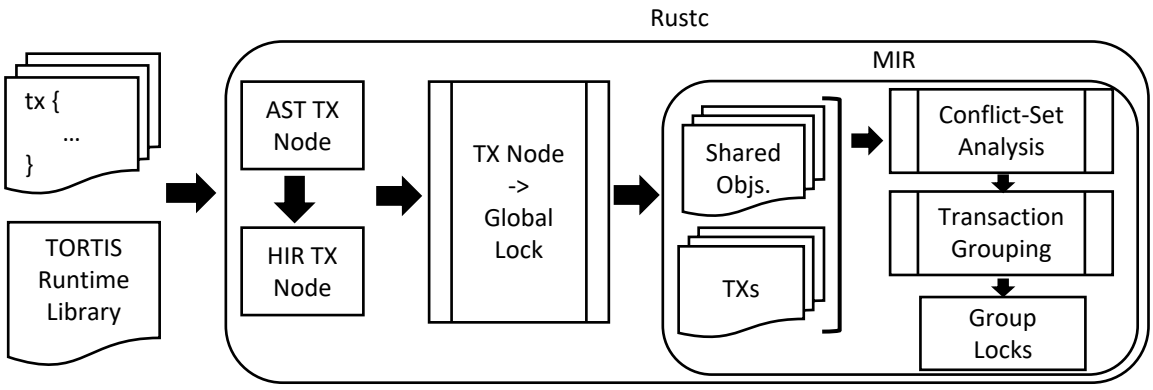


Figure 3-1: Transactions in source code, along with our runtime library are passed to the TORTIS `rustc`. Internally, transactions are added to the AST and HIR nodes before being desugared to global locks when generating MIR. At this stage, a correct STM system exists but provides no concurrency. As an optimization, TORTIS adds a data-flow analysis on shared objects and transactions to determine what shared objects a transaction uses, *i.e.*, its resource group. The transaction conflict sets are then analyzed, and the global lock optimized to the correct resource group lock for each transaction.

example, if transactions 1 and 2 conflict they must use the same group lock. If transactions 2 and 3 also conflict they must also use the same group lock. Since transaction 2 can only use one group lock, transactions 1 and 3 must use the same group lock even though they do not conflict themselves.

Fig. 3-1 illustrates the TORTIS design, and Fig. 3-2 illustrates where in the Rust compiler pipeline each step takes place. To achieve our design goals, we first introduce syntactic sugar for transactions, in effect our API for STM. Transactions are desugared by the compiler to global locks, which nominally achieve retry-free STM albeit with fully synchronous transactions. Our primary contribution is showing that Rust enables these locks to be optimized to group locks that enable a reasonable degree of concurrency and are still retry-free, permitting tight schedulability analysis. We describe the required steps for TORTIS here:

1. **Import TORTIS runtime library.** Programmers must import the TORTIS runtime library to use our wrapper types for shared objects, and allow our STM system to recognize them. The library also contains our lock implementation.

2. **Recognize transactions.** Transactions are syntactic sugar that are parsed

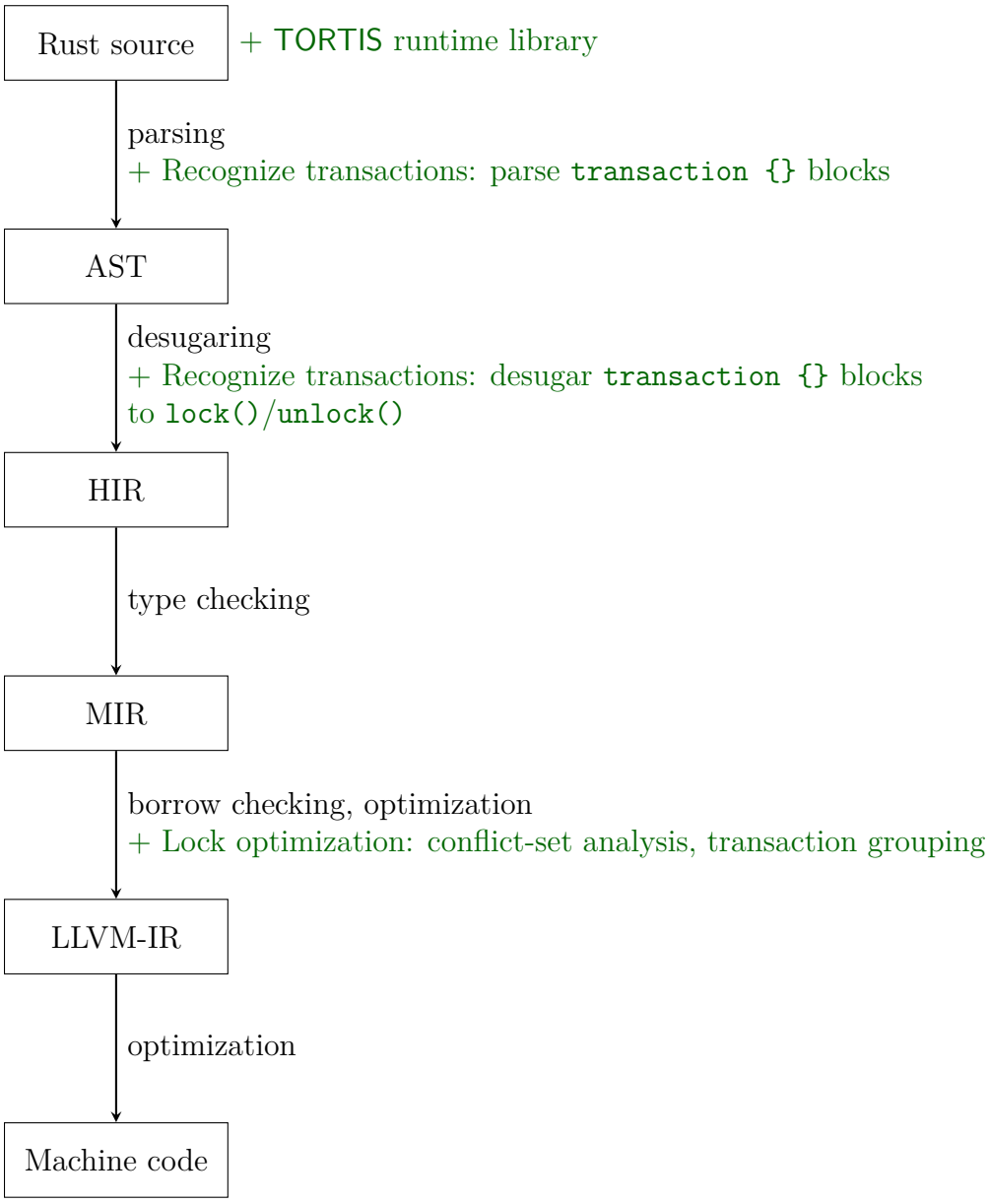


Figure 3-2: The Rust compiler pipeline with TORTIS additions.

and interpreted to a single, global lock by the compiler. At this stage a correct STM system with fully synchronous transactions exists.

3. **Optimize Locks.** TORTIS next optimizes the global transaction lock to resource-group locks to provide retry-free, concurrent transactions.

    - **Analyze conflict sets.** TORTIS analyzes the MIR to determine conflict sets per transaction, using a data-flow analysis from shared-object allocations to transactions.
    - **Group transactions.** Given the conflict sets of each transaction, the compiler groups transactions into resource groups and updates the transaction locks to the correct group lock.

3.1 Running Example

To illustrate many concepts, we present a running example in Lst. 3.1. In this example, transactions are monetary transfers among several different `BankAccount` objects, which are processed in different threads.

3.2 Import TORTIS Runtime Library

While most of the TORTIS implementation is based on extensions to the Rust compiler, we also developed a TORTIS library that must be used by the developer. In particular, this library provides wrapper types that enable interior mutability, as well as implementing the `Send` and `Sync` traits, enabling sharing across threads. These wrapper types, `TxCell` and `TxPtr`, are based on abstractions in the Rust standard library for single-threaded code, namely `Cell` and `UnsafeCell` respectively. `Cell` provides interior mutability by copying values into and out of a mutable memory location via the functions `get()` and `set()`. `UnsafeCell` provides interior mutability via shared references to a memory location.

```
fn main() {
    let alice = Arc::new(
        TxRefCell::new(
            BankAccount::new()
        )
    );
    let bob = ... //same initialization
    let charlie = ...
    let dawn = ...
    let t1 = thread::spawn(move || {
        transaction {
            transfer(alice, bob);
        }
    });
    let t2 = thread::spawn(move || {
        transaction {
            transfer(charlie, dawn);
        }
    });
}
fn transfer(
    src: Arc<TxRefCell<BankAccount>>,
    dst: Arc<TxRefCell<BankAccount>>,
) {
    let srcRef = src.borrow_mut();
    let dstRef = dst.borrow_mut();
    *srcRef -= 10;
    *dstRef += 10;
}
```

Listing 3.1: An example TORTIS program.

Our first attempt at an object with interior mutability without copying used `RefCell` instead of `UnsafeCell`. `RefCell` enforces Rust's ownership and borrowing rules at runtime: it keeps an internal reference count and panics if there is any attempt to borrow a value that is already mutably borrowed. This attempt initially seemed correct when tested with mutexes, but failed when tested with read-write locks because `RefCell`'s reference counting is *not atomic*. Two threads could race to decrement the reference count, leading to an incorrect count and spurious panics. This runtime reference counting is not necessary for TORTIS because the data-flow

analysis ensures that locks enforce Rust's ownership and borrowing rules. It imposes an unnecessary performance overhead as well. Thus, we switched to `TxPtr` using `UnsafeCell`.

`TxCell` and `TxPtr` differ from their single-threaded counterparts in that they are given `Send` and `Sync` traits, which allows them to be shared among multiple threads. However, these traits do not themselves implement a synchronization mechanism, but are simply promises to the type system that such a mechanism does exist for this type. `TxCell` and `TxPtr` are thus wrapper types that can be recognized by the compiler as transactional shared objects and trigger 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 [30, 7] use similar cell- and pointer-based wrappers.

We note that `TxCell` and `TxPtr` (and the data types they wrap) are the only types of shared objects TORTIS protects. This does not imply a loss in generality, as `TxCell` and `TxPtr` can wrap arbitrary data types. Further, the programmer can avail themselves of any synchronization method for shared objects not wrapped in `TxCell` or `TxPtr`, albeit without the guarantees of the TORTIS STM system. However, we make no guarantees about how such other synchronization methods may interplay with TORTIS. 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. However, given the thread-safety properties provided by Rust, they are the only types that do not already provide some underlying form of synchronization for the wrapped object (transactions provide synchronization for multiple objects). For example, to create a shared object outside of TORTIS, you would use a type such as `Mutex`, which provides its own synchronization, albeit subject to programmer error resulting in, *e.g.*, deadlock.

In addition to the wrapper types, the TORTIS library contains our lock implementations. Putting the lock implementations inside the library rather than inside the compiler itself allows the library implementer to change the lock implementation

```
{
    lock(0);
    {
        /* critical section */
    }
    unlock(0);
}
```

Listing 3.2: A desugared transaction in Rust source code.

without rebuilding the compiler, thereby providing greater flexibility to experiment with lock implementations. This is made possible by annotating the functions as *lang items*, or operations that "plug in" to the compiler without being in the compiler source code. We have included implementations of a naive mutex spinlock, a mutex ticket lock, and a "phase-fair" reader-writer ticket lock that alternates between read and write phases [16]. The phase-fair reader-writer spinlock is designed for multiprocessor real-time systems: it has asymptotically optimal blocking bounds with $O(m)$ writer blocking and $O(1)$ reader blocking. This design opens up the possibility of a future API to specify the desired lock implementation for an application, for example, to allow the programmer to specify if a spin- or suspension-based lock should be used. For further discussion of lock scalability, see [17].

Our library also provides an API to access specific group locks, with the lock number used as an argument to the exposed `lock()` and `unlock()` functions. The global lock is number zero. Changing to a specific resource group lock only requires updating the function argument. All other details are abstracted out of the API and handled internally by the library.

3.3 Recognize Transactions

TORTIS adds the `transaction` keyword to Rust as syntactic sugar for transactions, as shown in Line 13 of Lst. 3.1. Each transaction is converted to a node in the AST, like other code blocks. Transaction blocks in the AST become transaction blocks in HIR, which retains a tree-based node structure.

When lowering from the HIR to MIR, the `transaction{}` blocks are desugared to `lock(0)` and `unlock(0)` calls surrounding a regular code block. These calls reference the global lock (zero) by default, but are later optimized to use group locks. This ensures that transactions have their own scope, and thus that any references borrowed within a transaction block will drop out of scope when the transaction completes. In concert with the inserted locks, this ensures that two threads will never hold a mutable reference to the same transactional shared object concurrently. The source equivalent of desugared transactions is shown in Lst. 3.2. These `lock` and `unlock` calls reference the lang items `transaction_lock` and `transaction_unlock` in the TORTIS library, as discussed previously. These calls serve as transaction identifiers in later compiler passes, as anything between the `lock` and `unlock` calls is part of the transaction, and enable our subsequent data-flow analysis for lock optimization.

In our first (failed) implementation effort, we attempted to flag MIR basic blocks as being a transaction or not, which proved to be impossible. AST / HIR nodes correspond precisely to scopes in the code (*i.e.*, regions between curly brackets), and thus one node in the AST / HIR corresponds to a given transaction. Basic blocks in MIR and lower representations break this correspondence. For instance, if a transaction begins with an `if` statement, the conditional branch will be the last statement of the preceding basic block. Consequently, we cannot cleanly label a basic block as being part of a particular transaction. By adding locks/unlocks before lowering from the HIR, we avoid the issue of mapping basic blocks to transactions, and have a clean notion of what is in a transaction: anything between the lock and unlock calls.

By inserting calls to the global lock, the transaction desugaring alone implements a strawman retry-free STM system, albeit one with no concurrency. A similar system could be implemented in C/C++. However, we next show that it is possible to optimize these locks and achieve reasonable concurrency in Rust, whereas this is impossible in C/C++ as discussed in Sec. 2.5. A key insight of this thesis is demonstrating that a strongly typed language with no aliasing such as Rust enables retry-free STM with reasonable concurrency.

3.4 Optimize Locks

To optimize locks, we must first determine which transactions *may* conflict, *i.e.*, which transactions are in the same resource group. Resource groups are defined as the transitive closure of all transactions which may access a common object, *e.g.*, if transactions A and B access a common shared object, as do transactions B and C, then all three transactions are in a resource group. Resource groups are thus defined by determining what shared objects a transaction *may* access. We determine this by working in the opposite direction, mapping each shared object to the transactions that it *may* reach. Here we first cover creating resource groups via *conflict-set analysis*, then show how resource groups are used to group transactions. Our goal is to have many small resource groups as this results in less lock contention, and thus greater concurrency, at runtime.

3.4.1 Conflict-Set Analysis

Conflict set-analysis has two pieces: identifying transactions and shared objects, and creating a per-transaction list of accessed shared objects. For each shared object identified, we must determine the set of transactions that it *may* reach, *i.e.*, perform data-flow analysis on shared objects. `rustc` provides facilities for this at the MIR level, but not AST / HIR, which is why TORTIS implements conflict-set analysis on MIR. Before explaining the technical details, we illustrate the required data-flow analysis through the example in Lst. 3.1. Consider variable `alice` in `fn main`. The variable `alice` is defined on Line 2, which is the unique identifier for that object. A reference to `alice` is passed on Line 10 in the closure to spawn a new thread. Later, the variable is passed to the transfer function on Line 21 inside a transaction. Finally, it is mutably borrowed on Line 25. The goal of our data-flow analysis is to follow all shared objects, like `alice`, to all their uses to determine which transactions use which shared objects. For the purpose of our implementation, the data-flow analysis must be complete, but may not be exact: over-approximating the conflict set may cause additional blocking, thereby decreasing performance and schedulability

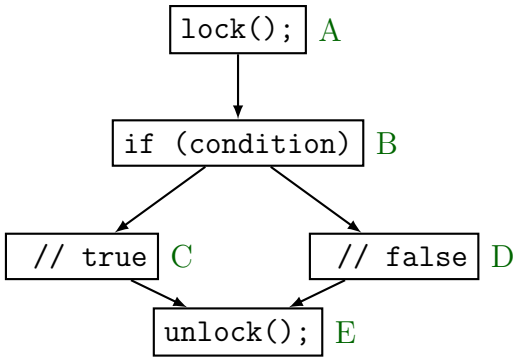


Figure 3-3: Reverse post-order traversal of a graph. In a reverse post-order traversal you visit a node before you visit any nodes reachable from it. This ensures that the data-flow analysis visits every possible basic block in the control flow between the `lock()` and `unlock()` calls. A reverse post-order traversal of this graph is either "A B C D E" or "A B D C E".

but maintaining correctness, while under-approximating it may invalidate transaction semantics.

TORTIS identifies *transactions* within MIR based on the `lock` and `unlock` calls inserted in the prior step (transaction recognition). MIR has a more primitive set of operations than Rust source code and is organized into basic blocks similar to LLVM-IR, with explicit control-flow transitions (such as function calls) from one basic block to another at the end of each basic block. Basic blocks can be ordered arbitrarily within a function, and thus finding the start and end of a transaction and all basic blocks that may execute in between cannot be done by linearly iterating through the basic blocks. Hence, we perform a reverse post-order traversal of basic blocks based on their control flow to identify each transaction's start and end, as well as all the basic blocks in between. Fig. 3-3 illustrates a post-order traversal. This ordering ensures that we see a `lock` call before its corresponding `unlock`, and that all basic blocks that can be executed between are properly associated with the transaction. We use the starting point of the transaction, *i.e.*, its `lock` call, as its unique identifier. With this reverse post-order traversal, we know which function calls are part of which transactions.

Identifying *shared objects* statically requires a simplifying assumption: that all objects allocated at the same line of code are the same object. It may be possible in

some cases to distinguish objects via whole program context and data-flow sensitive analysis, or through dynamic analysis, but we leave achieving additional precision as an open challenge. Our simplifying assumption does introduce conservatism into the conflict analysis. For example, if objects are created within a loop, we treat all objects created by a single line of code in the loop as the same object. While this is conservative, over-approximating the conflict set of a transaction is safe in that it ensures atomicity, but may lead to increased blocking.

The next step of our data-flow analysis is to identify all uses of shared objects within each transaction. The Rust compiler provides facilities for performing *definition-use analysis*, commonly called def-use analysis, in MIR. This analysis returns every use of a given object within the local function. Such a use could be an arithmetic operation, reference, copy, *etc.* With these results combined with the function-call-to-transaction mapping from the reverse postorder traversal, we correlate each use of a TORTIS object with any transaction in the local function. However, transactional shared objects are seldom used solely in the function in which they are declared (they may be passed through a closure to be used in another thread, as in Line 10 of Lst. 3.1). Thus, we must perform interprocedural analysis to identify all potential uses within transactions.

The interprocedural def-use analysis consists of a depth-first search (DFS) that starts at the definition of a TORTIS shared object and traces its uses. When an object is used as an argument to another function call or closure, the search continues by applying the Rust compiler's def-use analysis recursively on that function or closure body. The search stops recursing upon encountering an access of the shared object (`get`, `set`, `borrow`, or `borrow_mut`).

These results are stored in a use-to-allocation mapping, associating the use (read or write) with a set of possible shared objects that line could have referred to. The search finds *all* uses of a shared object within a transaction to determine whether the transaction writes or merely reads the object. Combining the function-call-to-transaction mapping from the post-order traversal with this definition-to-use mapping, we can determine which allocations *may* be used in a transaction, *i.e.*, its conflict set. If a

use is *not* contained in a transaction we emit an error. This ensures that transaction semantics cannot be violated by accessing shared objects outside of transaction blocks.

DFS results are memoized so that deep traversals are not done repeatedly. For example, consider the data-flow analysis of `charlie` in Lst. 3.1 after the analysis of `alice` completes. The `charlie` variable is allocated on Line 8 and also used as the first argument to `fn transfer` on Line 17. Instead of repeating the recursive DFS to find that the object is terminally used on Line 25, it can read the memoized results and report that `charlie` is accessed by the transaction on Line 16.

Data-flow analysis must scan every function in the developer's entire crate (Rust's nomenclature for a library) to determine all the shared objects that could possibly be used in a transaction. This was a challenge to implement in Rust MIR because of the compiler's *query system*. Instead of organizing compilation as a series of passes over the source code that execute sequentially, MIR code generation is organized as a directed acyclic graph (DAG) of *queries*. Each query (other than the "root" query) may depend on other queries, but may not form a cycle. TORTIS adds conflict analysis and data-flow analysis to the query graph, with the conflict-analysis query depending on the data-flow analysis query from every function in the crate. This wide dependency can create cycles among the different queries in the compiler, which can be at different stages of optimization, and also be concurrently processed by different threads in the compiler. TORTIS prevents this cycle by (i) excluding an unstable, experimental function type from the analysis, and (ii) cloning the MIR data structure in one place instead of passing it by reference. Consulting with the `rustc` compiler team revealed that MIR-level, whole-crate analyses are currently lacking in the compiler, yielding poor support for them and requiring cloning the MIR data structure. Cleaner implementations may be possible at the LLVM-IR level, if the enabling assumptions of Rust's type system, specifically that there is no aliasing, are preserved when lowering to LLVM-IR.

Transaction	Conflict Set
Transaction 1	{A, B}
Transaction 2	{C}
Transaction 3	{A, C}
Transaction 4	{D}

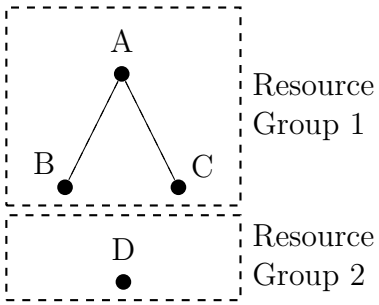


Figure 3-4: Computing the transitive closure over the conflict sets. Transactions 1, 2, and 3 conflict and must share Group Lock 1. Transaction 4 has no conflicts and can have its own group lock.

3.4.2 Transaction Grouping

Transaction grouping takes the conflict sets created by conflict-set analysis and uses them to group transactions that *may* conflict. This formulation of resource groups ensures that transactions are atomic, can never deadlock, and only requires one lock per transaction.

We convert the resource-grouping problem into finding connected components in a graph, where transactions are nodes, and an edge exists between two transactions if they access one (or more) common shared objects. Fig. 3-4 depicts such a graph and the transitive case—though Transactions 1 and 2 have no objects in common, Transaction 3 conflicts with both of them so all three conflict. Connected components in this graph represent resource groups, as elements are connected if and only if they transitively access a common shared object.

Mutex Locks

Once the transaction groups are determined, the last step is to update the `lock()` and `unlock()` calls in individual transactions to reference the lock for the correct resource group instead of the global lock. According to our runtime library design, the lock is specified as an integer argument in our API. Consequently, changing the lock used by a transaction only requires updating the argument to the API call.

Transaction	**Conflict Set**	**Resource Group**	**Lock**
Transaction 1	{read A, **write B**}	Resource Group 1	**Write**
Transaction 2	{read C}	Resource Group 1	Read
Transaction 3	{read A, read C}	Resource Group 1	Read
Transaction 4	{**write D**}	Resource Group 2	**Write**

Figure 3-5: Read-write lock emission after transaction grouping. Each transaction must acquire the read or write lock for its resource group. Transaction 1 writes one of the resources, so it must acquire Resource Group 1's write lock, while Transactions 2 and 3 may acquire Resource Group 1's read lock. Transaction 4 acquires the write lock for Resource Group 2.

Read-Write Locks

If TORTIS is configured to use read-write locks, then the `lock()` and `unlock()` calls must be optimized to `write_lock()` / `write_unlock()` or `read_lock()` / `read_unlock()`. A transaction that writes at least one shared object in its resource group must acquire the group lock exclusively (`write_lock()`), even if other objects in the group are merely read. A transaction may acquire the group lock as a reader (`read_lock()`) if it only reads shared objects and does not write any. Fig. 3-5 illustrates an example of this.

Chapter 4

Discussion

TORTIS's design decisions to obviate retries may sacrifice concurrency, most notably the granularity at which shared objects are identified. Here we discuss some of these tradeoffs, how transaction placement affects concurrency within TORTIS, and opportunities for future work.

4.1 Granularity of Synchronization

TORTIS uniquely identifies objects based upon their allocation site. For example, TORTIS often cannot differentiate list elements from one another, as they are all allocated in a common function. Therefore, 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. 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, schedulability analysis must assume they conflict. Consequently, TORTIS trades off potential performance for schedulability.

```
fn transfer_tx(
    src: Arc<TxRefCell<BankAccount>>,
    dst: Arc<TxRefCell<BankAccount>>,
) {
    transaction {
        let srcRef = src.borrow_mut();
        let dstRef = dst.borrow_mut();
        *srcRef -= 10;
        *dstRef += 10;
    }
}
```

Listing 4.1: Data-flow analysis limitations. This helper function contains one transaction, which acquires and releases one group lock. This lock must guard any possible shared objects passed as arguments to this function, so two transfers with non-conflicting shared objects will still conflict.

4.2 Transaction Placement

The nature of TORTIS's data-flow analysis favors some software design patterns over others. To illustrate this, consider the `transfer` function in Lst 3.1 in comparison to the `transfer_tx` function in Lst 4.1. By enclosing the `transfer` function in a transaction, each transaction has its own individual conflict set, thereby allowing the two transactions in Lst 3.1 to be in different resource groups. If `transfer_tx` was used instead, *i.e.*, the `transaction` block was moved to the helper function, instead of surrounding it, there would only be one transaction to analyze, instead of two. `alice`, `bob`, `charlie`, and `dawn` are all accessed within the lone transaction in `transfer_tx`. Consequently, all four are in the conflict set for the transaction, forcing the serialization of what are two parallel transactions in Lst. 3.1. This shows how pulling transactions out of helper functions and closer to specific objects can facilitate more precise conflict analysis.

Chapter 5

Evaluation

To evaluate both the strengths and weaknesses of TORTIS, we conduct two general classes of evaluations: (i) throughput-oriented experiments, and (ii) schedulability experiments. The throughput-oriented experiments are designed to test how TORTIS scales and performs under different types of contention. Then based on these experiments, we conduct schedulability experiments to quantify how the more deterministic design of TORTIS affects schedulability.

Ideally we would compare our system against an existing open-source real-time STM system with associated schedulability analysis, but we are not aware of any. Instead, for both experiments we chose to compare against the well-known algorithm Transactional Locking II (TL2) [20], performing our own schedulability analysis for both TORTIS and TL2. We chose TL2 as a baseline for comparison for four main reasons: (i) like TORTIS, TL2 is lock-based, though it does allow for retries; (ii) TL2 is a well-established algorithm (over 1,000 citations); (iii) there is an open-source Rust-based implementation [30], which allows us to perform performance comparisons without the confounding variable of language performance issues; and (iv) measurement-based studies of some prior real-time STM approaches have been shown inferior to lock-based synchronization [21].

To better understand our experimental results, we provide a brief overview of TL2. Transactions are effectively simulated at runtime to determine what objects they access. This simulation requires additional overhead not present in TORTIS, 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. We modified the Rust TL2 implementation to lock objects in address order to prevent deadlock and ease schedulability analysis. However, for our throughput evaluation, we compare against both the random (original, TL2) and ordered (modified, TL2-O) lock acquisition. A transaction is forced to retry if an object it accesses 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. We note that TL2 can harness (retry-based) hardware TM features (*e.g.*, Intel TSX) to improve performance, but this was disabled to compare pure software-based TM approaches.

5.1 Throughput Evaluation

STM systems are most commonly evaluated on *observed average-case throughput*. We conducted similar evaluations in the interest of full transparency to demonstrate how TORTIS performs in comparison to prior work. In particular, we considered transactions operating on a buffer and varied the access patterns to identify the limits of concurrency under both approaches.

We ran these experiments on a two-socket, 18-cores-per-socket x86 machine running the Linux 4.9.30 Litmus-RT kernel [12] with two Intel Xeon E5-2699 v3 CPUs @ 2.30GHz with 128GB RAM and 32KB L1, 256KB L2, and 46080KB L3 caches. All experiments were run in the partitioned earliest-deadline-first scheduler (P-EDF) with one thread per core. For each benchmark, we ran transactions in a tight loop for three seconds per core to measure the total transactions per second across all cores that executed.

For the linear benchmarks, multiple threads read and write a single shared fixed-size buffer. Each transaction accesses a given percentage of randomly chosen buffer elements per loop iteration. Some percentage of these accesses are writes, while the

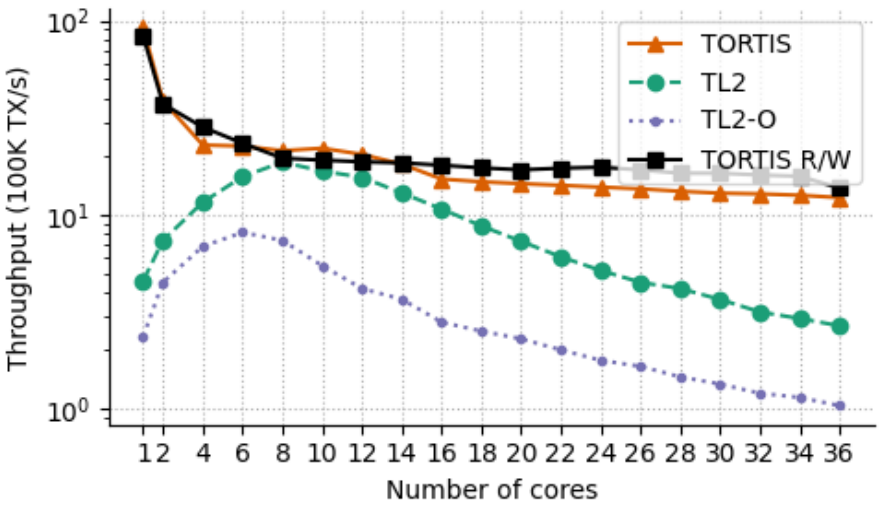


Figure 5-1: Varying the number of cores. 64-element buffer, 10% of elements accessed, 5% writes.

rest are reads. For all benchmarks, the accessed elements are randomly determined before the transactions execute so as not to include overhead in the transaction or measurement time. We vary the number of cores, accessed-elements percentage, and write percentage for our experiments.

In our first experiments, we varied the number of cores to evaluate how both STM systems scale. Each transaction accesses 10% of the elements in a 64-element buffer (6/64 elements), with a 5% probability of writing to the element. The results of this experiment are shown in Fig. 5-1. TORTIS's throughput strictly decreases as the number of cores increases because it guards the buffer with a single group lock, which prevents concurrency. Increasing core counts increase blocking and cache-line bouncing, which decreases overall throughput. As the number of cores increases to 8, TL2 sees increased throughput due to the parallelism afforded by scaling to more cores. However, beyond 8 cores the contention begins to reduce throughput as more transactions are forced to retry or block to commit results.

Next, we sought to evaluate how the number of elements accessed within each transaction affected throughput. In TL2, the more elements accessed, the more state must be tracked in the transaction simulation, and the more locks that must be acquired, each of which incurs additional overhead, not present in TORTIS. The

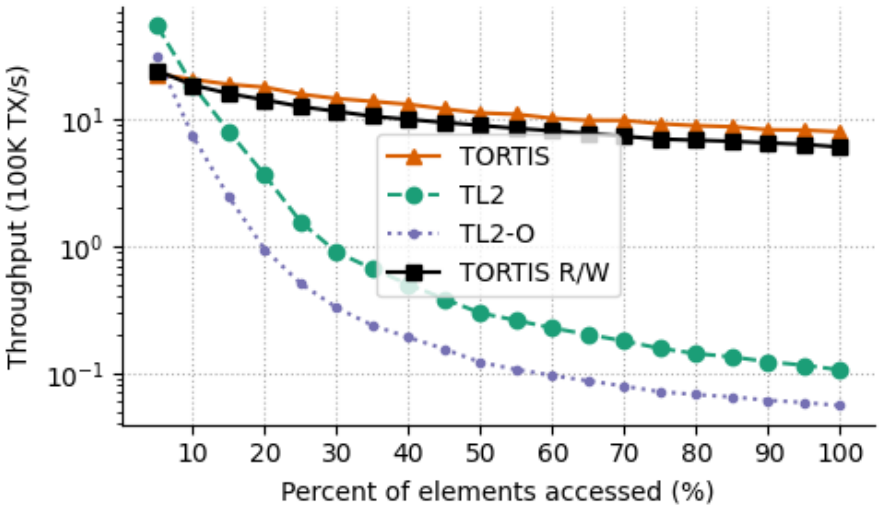


Figure 5-2: Varying the number of elements accessed. 8 cores, 64-element buffer, 5% writes.

results of this experiment are shown in Fig. 5-2. Note that TORTIS throughput decreases gradually as accessing more elements increases critical-section length.

Finally, to explore the effect of read/write concurrency, we varied write percentage, assuming 8 cores and 10% of elements accessed in a 64-element buffer. While TORTIS can be configured to use mutexes or read-write locks, TL2 is optimized for read-only transactions. The results of this experiment are shown in Fig. 5-3. Mutex TORTIS's performance is virtually unchanged as the write percentage decreases and TORTIS with read-write locks shows up to a 2.6x improvement, while TL2 and TL2-O exhibit two orders of magnitude greater throughput.

There are 12 additional graphs based on different experimental parameters available in the appendix. Based upon these experiments, we make the following two high-level observations.

Obs. 1 *TL2 has greater average throughput than TORTIS when transaction contention is low, by a factor up to* $100x$.

This trend is observed in Figs. 5-2, and 5-3. This is to be expected, as TL2 is specifically designed to improve throughput in the common case of relatively rare contention. For example, consider Fig. 5-2: when the percent of elements accessed

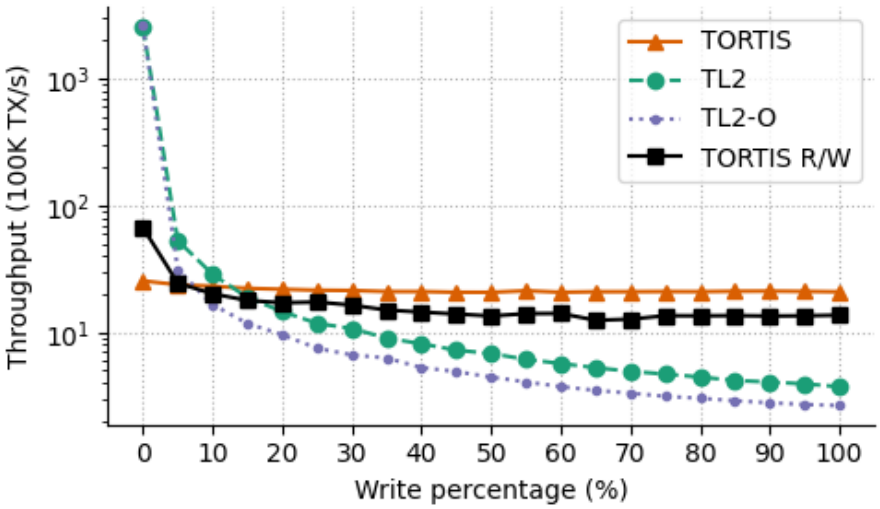


Figure 5-3: Varying the write percentage. 8 cores, 64-element buffer, 10% of elements accessed.

within the transactions is small, TL2 outperforms TORTIS by 2.4x. Similarly, when the overwhelming majority of accesses are read accesses, which TL2 enables to run concurrently, the throughput is nearly 100x that of TORTIS as seen in Fig. 5-3.

Obs. 2 *In more highly contended cases, TORTIS performs substantially better than TL2, by a factor of as much as 75x.*

This observation is supported in all three of our throughput Figs. (5-1, 5-2, and 5-3). When each transaction operates over the entire vector there are fewer opportunities for concurrency, and thus the throughput is lower for both TORTIS and TL2. However, TL2 is subject to additional overheads from retries, both in the actual recomputation of the transaction, as well as the need to maintain additional metadata and perform additional data copies. These overheads dominate the runtime, and substantially reduce performance. In contrast, in TORTIS a single lock is acquired before operating on the buffer in place—a much more efficient operation. When more than 50% of the elements are accessed per transaction, TORTIS outperforms TL2 by an average of 59x, and up to 75x when every element in the buffer is accessed. Similarly, when the write percentage increases, read transactions cannot execute concurrently as often, and a similar trend is observed in that TORTIS enables an average of 4.4x

more throughput when the write percentage is over 50%.

5.2 Schedulability Evaluation

Our principal objective in the TORTIS design was to enable predictable worst-case performance, and thus improve schedulability. In doing so, we willingly sacrifice average-case throughput in more common uncontended scenarios in favor of higher throughput in contended scenarios, and tighter bounds on synchronization-related overhead. To evaluate these points, we conducted a schedulability study. Before describing our study in detail, we briefly overview how we account for transaction interference under TL2 and TORTIS.

5.2.1 Analysis Approaches for Synchronization

Our analysis for both TL2 and TORTIS builds upon the recent inflation-free analysis framework of Biondi *et al.* [8] to determine schedulability under P-EDF. This framework determines schedulability by bounding the maximum deadline busy-period length and includes terms that account for synchronization interference. *Synchronization interference* on a given CPU partition is the duration of processor time spent blocking or retrying during a given interval for tasks to complete.

For TORTIS, synchronization interference is entirely due to blocking, as with other lock-based synchronization methods. We apply the integer-linear-programming-based non-preemptive FIFO lock analysis presented with the inflation-free framework [8] (which includes release blocking) for TORTIS. We note that with the *lang item*-based approach to lock insertion from Sec. 3.3, swapping lock implementations is straightforward, and thus TORTIS schedulability can be assessed by simply assessing any existing locking protocol.

There was no prior schedulability analysis for TL2. As such, we extended the same analysis framework to account for TL2 synchronization costs. In TL2, synchronization interference can be comprised of both retries and blocking; one of its underlying steps is to lock all resources that will be written during a transaction. Though lock-free

analysis can account for repeated retries due to resource contention, existing lock-free analyses do not apply to TL2, as they do not account for the combination of retries and blocking that can occur.

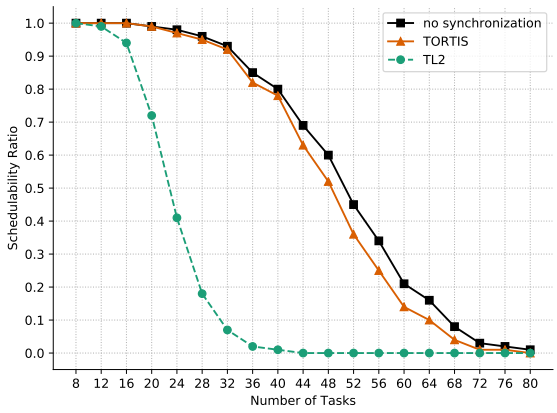
Details of our analysis are included in the appendix, in which we also give examples of the types of transaction interference. There, we illustrate how both local (same CPU) and remote (other CPUs') transactions can force a given transaction to retry. For each local transaction, we factor in that it may cause a retry when the task of interest is preempted during its execution of a transaction; the preempting task may execute a transaction that invalidates the results of the simulated execution, forcing a retry when that transaction resumes execution. The analysis also accounts for interference caused by remote transactions that can be executing concurrently with the task of interest. Our computation of interference is calculated on a per-task basis, allowing us to account for the number of interfering transactions that may be present during a given interval.

When analyzing an individual transaction, we must consider which resources may be accessed. For the buffers examined above, separate transactions could access different buffer elements concurrently. However, as elements are added to and removed from a given buffer, the element accessed by a transaction may change. Thus, we assume that each access to a given data object (*e.g.*, buffer) may conflict with any other access to the same data object. Therefore, the conflict sets for TL2 and TORTIS are the same; we cannot statically determine which elements will be accessed within an object.

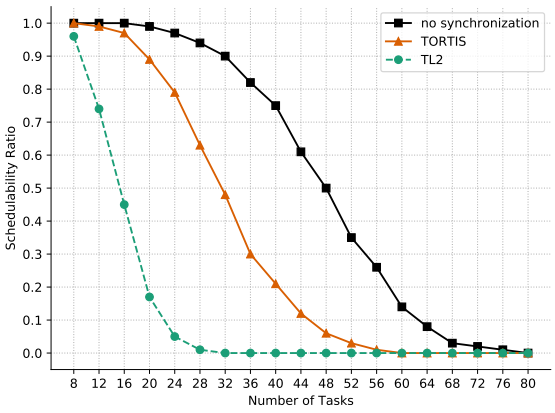
5.2.2 Experimental Design

We used SchedCAT [1] and the inflation-free framework [8] to analyze the schedulability of task systems under P-EDF on an eight-processor platform.

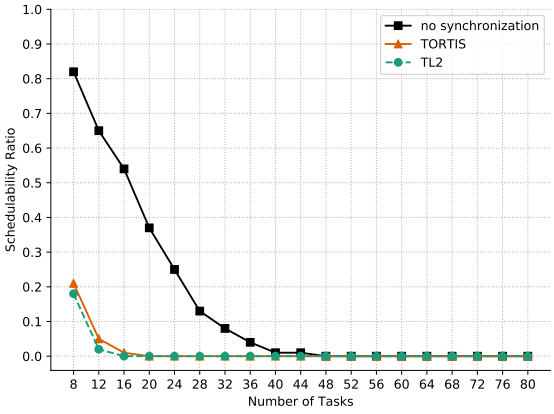
We generated implicit-deadline sporadic task systems for scenarios categorized by task periods, task utilizations, number of shared objects, object-access durations, the probability that a task contains any transactions for a given object, and the number of transactions that task issues for that object. We varied these parameters



(a) 4 objects, access probability of 0.25, short transaction length, 1 transaction.



(b) 16 objects, access probability of 0.10, medium transaction length, 1 transaction.



(c) 8 objects, access probability of 0.50, medium transaction length, number of transactions chosen from $\{1 \ldots 5\}$.

Figure 5-4: For these scenarios, there were four shared objects, and task periods were chosen from $[1ms, 1000ms]$.

to represent systems similar to those studied in prior work [8]. A *scenario* is defined by a particular selection of these parameters. Task periods were selected from a log-uniform distribution in $[10ms, 100ms]$ or $[1ms, 1000ms]$. 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 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.) We used the same transaction length for both TL2 and TORTIS, selected uniformly from either $[1\mu s, 25\mu s]$ (*short*) or $[25\mu s, 100\mu s]$ (*medium*). We did not include overheads.

5.2.3 Favoring the Competition

By using the same transaction length for both TL2 and TORTIS, *we are favoring TL2*. A transaction handled by TORTIS simply reads or writes its set of elements. TL2, on the other hand, simulates a transaction's execution, acquires write locks, and checks the simulated values before writing the values and releasing write locks. As shown in Fig. 5-1, when the number of threads is one, TORTIS outperforms TL2 by a factor of 15x, which is reflective of the extra bookkeeping required for retries. Accounting for this added work would increase the retry cost of each transaction under TL2. We chose to ignore this work to make our evaluation independent of TL2 implementation choices, and instead focus on retry-free vs. retry-based analysis tradeoffs. Furthermore, we assume blocking for locks to commit transaction results is negligible. Hence our analysis applies equally to TL2 and TL2-O. If we did account for blocking, TL2-O would have lower blocking bounds than TL2. Even with these generous simplifications, TORTIS offers significant schedulability improvements over TL2.

5.2.4 Schedulability Results

Considering all possible combinations of task-set parameters results in 72 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 1000 independent task systems. We plot the *schedulability ratio* of these, which is computed by taking the ratio of task systems that were schedulable by each approach out of the 1000 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 report the TSA ratio showing the improvement of TORTIS over TL2.

Each scenario resulted in one graph. All 72 graphs are included as supplementary material, and in Fig. 5-4, we present three of these; we show the graphs corresponding to the scenarios with roughly the maximum, average, and minimum amount of schedulability recovered by TORTIS over TL2. 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 *no synchronization* line in each plot represents P-EDF schedulability of the independent tasks, including no synchronization. A task set that is not schedulable under *no synchronization* cannot be schedulable after synchronization interference is accounted for, and is plotted to show utilization loss associated with synchronization.

From these results, we make the following observations.

Obs. 3 *TORTIS dominates TL2 with respect to schedulability in all observed cases.*

This observation is shown in Fig. 5-4. For all scenarios, the TSA of TORTIS was higher than that of TL2. 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 lower-overhead, retry-free, deterministic synchronization as used in TORTIS provides improved schedulability. 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) nor the blocking on locks to commit per-object results.

Obs. 4 *In some scenarios, the schedulability improvement from TORTIS over TL2 is modest.*

This is observed in Fig. 5-4c, which depicts a system that is difficult to schedule. Here, the TSA ratio is 1.5. With the high resource contention, there is little room for schedulability improvement. At the other end of the spectrum, we observed that the

TL2 performed well in scenarios with low contention, leaving minimal opportunity for improvement.

Chapter 6

Future work

This first investigation of retry-free STM provides a rich platform for further study of real-time STM. We plan to address limitations of TORTIS's current implementation and the implications of retry-free STM in future work.

Multi-crate analysis. Given the Rust compiler's underlying structure, performing analyses that span multiple crates is difficult. Thus, TORTIS does not correctly protect objects that are operated upon within transactions in different crates. In future work, we plan to explore how to support conflict analysis and lock updates during link-time optimization (LTO) to enable whole-program transaction analysis, even across shared libraries.

Nested locking and fine-grained locking. We also plan to leverage the significant body of prior work on nested locking in real-time systems [43, 9, 34, 42] to enable finer-grained locking for transactions, and thus greater runtime concurrency. Fine-grained locking can also be optimized by **moving locks** to delay acquisition or expedite release, shortening the critical section while preserving atomicity.

Transaction priorities and suspend vs. spin. When transactions conflict, prioritizing the more important transaction leads to better schedulability and responsiveness. Handling priority levels in the STM system keeps the developer from running into priority-inversion issues [36, 28, 14]. The system could also behave more predictably with better performance if the contention manager chose to emit not only spinlocks, but also suspension-based locks. Future extensions could use a suspension-

aware locking protocol with priority levels like RNLP [44] or other priority-aware locking protocols like FMLP [11] or FMLP+ [15].

Different lock granularities per transaction. Given the tradeoffs between lock granularity and performance, perhaps fine-grained locks are better for one transaction while coarse-grained locks are better for another transaction within the same program. In addition, it may even be more performant in some cases to emit one lock per field in a shared object. More sophisticated data-flow analysis could use heuristics to determine the best approach for a given data structure or transaction.

Contention-based analysis. The amount of time a transaction blocks in TORTIS depends not only on the contention for the resources it uses, but the contention for all the resources in its resource group. Finer-grained contention analysis could break these "transitive blocking chains" to make the blocking time directly proportional to the number of other transactions attempting to use only the same resources [35].

Runtime lock management. TORTIS's analysis is all performed statically at compile time. Runtime lock management could help determine which subset of potentially accessed shared objects are within scope at the start of a transaction, preventing unnecessary locking.

Study of I/O. While transactions that do not retry can theoretically support I/O, TORTIS does not yet have a clear plan for how to support I/O. Further exploration is needed to determine whether TORTIS requires a new I/O abstraction and what that abstraction might look like.

Case study. We compared TORTIS to TL2 using synthetic benchmarks on common data structures. It would be interesting to apply TORTIS to a larger, more complex real-time system with multiple data structures and evaluate its performance. It would also be interesting to perform a **usability study** to evaluate the difficulty of writing code using TORTIS.

Other optimizations. Two transactions in the same thread cannot block on one another, so should not need exclusive locks. Helper functions containing transactions could be duplicated or inlined to solve the transaction placement issue discussed in

Sec. 4.2. Transactions could more gracefully handle exceptions and release locks before panicking.

Chapter 7

Conclusion

We have presented TORTIS, the first ever retry-free STM framework. TORTIS is designed for application in real-time systems. A certified STM framework in this setting could greatly ease the development of certifiable real-time systems. This real-time focus motivated us to present the first ever schedulability analysis for an existing retry-based STM system. Based on this analysis, we conducted a schedulability study, in which TORTIS dominated prior work by a ratio of over 10x increase in schedulable task sets in some cases. In producing TORTIS, we limited attention to the use of group locks in dealing with conflicting transactions. In future work, we plan to consider finer-grained locking approaches, as well as other issues related to conflict analysis. TORTIS's modular design provides a sound basis for delving into these and other issues pertaining to real-time STM.

Appendix A

Bounding Transaction Interference

Here we describe how we account for the interference between transactions when TL2 is used.

A.1 Inflation-Free Approach

We build on an inflation-free approach [8] that bounds the maximum deadline busy-period length in a system scheduled with P-EDF. For each partition, we use the inflation-free approach by providing analysis of the possible retries or blocking in a busy-period for the TL2. As with the original analysis for lock-free and lock-based resource access that utilized this framework, all synchronization delays incurred by jobs in a busy-period can be accounted for using two terms for each processor.

These two terms are $B^{(PDC)}$ and $B^{(AC)}$, correspond to the processor demand criterion and the arrival curve, respectively. (Refer to the work by Biondi *et al.* [8] for more details on how these terms are used in their inflation-free analysis.) To compute $B^{(PDC)}$, we determine the additional processor demand that can be caused by synchronization interference. After describing how we account for both local and remote interference, we then discuss how the calculation of $B^{(AC)}$ differs.

A.1.1 TL2 Details

The TL2 [20] algorithm has six basic steps: (1) sample the global version clock, (2) run a speculative execution to determine the read-set and write-set, (3) lock the write-set, (4) increment the global version clock, (5) validate the read-set, and (6) commit and release the locks. A transaction can be forced to retry in Steps 2 and 4.[1] Additionally, a transaction can block while acquiring all of the locks for the write-set in Step 3. We state the assumptions we make for the analysis of TL2 below.

- We reason on the maximum transaction length. This is the duration of a transaction from start to finish (executing each of the steps described above) without any retries; we account for retries as taking this entire duration.
- The duration of each transaction is included in a task's WCET.
- Commit loops are preemptable.
- We do not consider a specific locking protocol in our analysis.

The first assumption accounts for time spent on a single transaction execution that results in a commit. What remains is for us to account for any additional transaction execution caused by retries. The second assumption reflects that non-preemptivity is not specified for TL2 execution; one can certainly imagine a system in which TL2 is modified to execute portions of transactions non-preemptively, and thus the analysis accounts for both cases. Similarly, we do not constrain our analysis to a specific locking protocol. Some protocol choices would allow additional constraints on interfering transaction to be imposed, but some would cause significant blocking (in turn impacting L_{max}). Instead, our analysis looks at which transactions could interfere on the basis of their required resources, not a specific locking mechanism.

A.1.2 Definitions

The following discussion is based on looking at a single processor of interest, P^*. We show how to account for the synchronization interference over a busy-period of length

[1] Depending on implementation decisions, a transaction may also retry in Step 3.

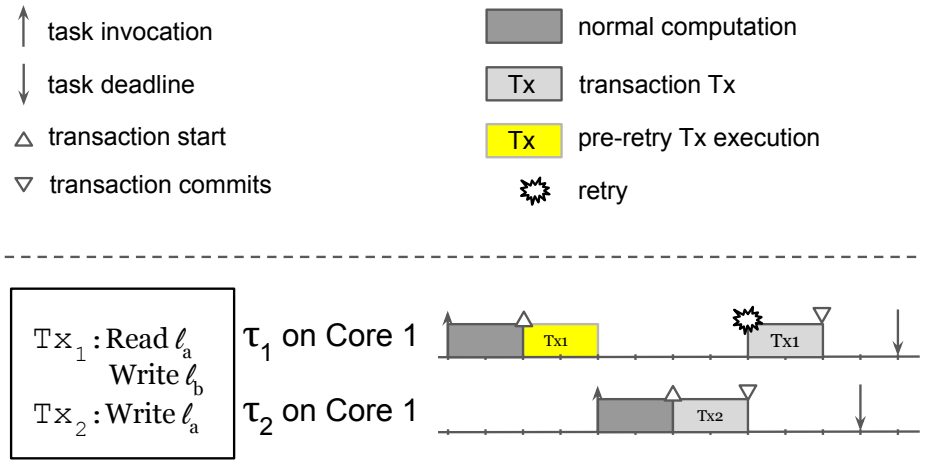


Figure A-1: An execution scenario in which a local transaction can cause a retry.

t to compute $B^{(PDC)}(P^*, t)$. The entire task system (across all partitions) is denoted Γ. The set of tasks on P^* is denoted $\Gamma(P^*)$. We denote an arbitrary task as τ_i. The period of τ_i is denoted p_i, and its deadline is denoted d_i. In the following examples, we reason about an arbitrary job of a given task. The execution of a job can contain one or more transactions. We denote an arbitrary transaction $\mathtt{Tx}_y$. Each transaction has a worst-case duration of L_{max}.

Based on the possible sequences of execution for tasks' transactions coordinated by TL2, we analyze the maximum duration of synchronization delays. We determine for each processor how much additional work (retries or blocking) may occur based on transaction interference.

A.1.3 Local Transaction Interference

A transaction is local when it shares the same partition as the arbitrary task τ_i .The following example shows how a local transaction can cause a retry.

Example 1 *As illustrated in Fig. A-1, a job of τ_1 is released at $t = 0$. At $t = 2$, it begins executing $\mathtt{Tx}_1$, which requires reading ℓ_a and writing to ℓ_b. $\mathtt{Tx}_1$ begins executing the steps of TL2 summarized above, starting with reading the global clock and doing the speculative execution. At $t = 4$, τ_2 releases a job with a higher priority (due to*

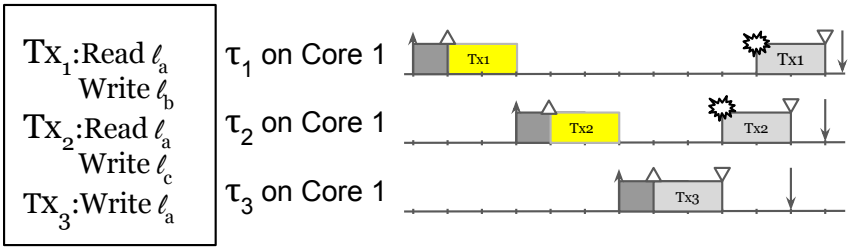


Figure A-2: An execution scenario in which a single local transaction can cause multiple (local) transactions to retry.

its earlier deadline). $\mathtt{Tx}_2$ *executes and commits, writing to* ℓ_a*. After* $\mathtt{Tx}_1$ *resumes, it detects (in Step 5) that* ℓ_a *has been updated. This forces* $\mathtt{Tx}_1$ *to retry, rendering its previous execution useless; it must start over from Step 1.* ◊

The transaction time is accounted for in a task's execution time, but our analysis of synchronization interference must also account for the wasted time spent executing the failed transaction prior to the retry.

Ex. 1 illustrates that a local transaction writing to a resource can force a transaction reading that resource to retry if its task preempts the reading transaction. The following example illustrates that a preempting task can cause multiple transactions to retry.

Example 2 *As illustrated in Fig. A-2,* τ_1 *is preempted while executing* $\mathtt{Tx}_1$ *by* τ_2*. Unlike in Ex. 1,* $\mathtt{Tx}_2$ *cannot cause* $\mathtt{Tx}_1$ *to retry. However, at* $t = 6$*,* τ_3 *begins executing, and at* $t = 7$ *issues* $\mathtt{Tx}_3$*.* $\mathtt{Tx}_3$ *writes to* ℓ_a*. Therefore, when both* τ_2 *and* τ_1 *resume, their respective transactions, which both read* ℓ_a*, must retry.* ◊

This example serves to illustrate that a single writing transaction preempting an executing transaction can cause multiple other transaction to retry.

For both of the above examples, we highlighted only a single job of each task. However, during the interval t, multiple jobs may execute of a given task may execute, each of which may contain a transaction. If we consider that τ_j contains a transaction that may preempt other tasks, we must consider $nljobs(\tau_j, t) = \lfloor \frac{t+p_j-d_j}{p_j} \rfloor$ jobs of this task during the busy-period of length t. This is the definition given with the original inflation-free analysis [8].

An additional consideration is that each task may contain multiple transactions. However, a given task can only be forced to retry once due to all other transactions of a given job. As illustrated in both of the examples above, a preemption is required to cause a retry, and each job can only preempt a given task once under P-EDF.

Based on the above observations, we define a binary variable to indicate possible interference due to a forced retry. For tasks τ_i and τ_j, we let $X_{i,j} = 1$ if τ_j's period is less than τ_i's period and any of τ_j's transactions writes to one or more resources that any transaction of τ_i reads. Otherwise, $X_{i,j} = 0$. Thus, if $X_{i,j} = 1$, there is an execution pattern that can result in a transaction of τ_j causing a transaction of τ_i to retry. Recall that, each access of an element of a larger data structure like a buffer could interfere with other access to that data structure. Therefore, for our schedulability analysis, if tasks τ_i and τ_j shared any data structure (through their respective transactions), then $X_{i,j} = 1$.

Over the busy-interval of length t, τ_i may incur retry interference of up to $\sum_{\tau_j \in \Gamma(P^*)} nljobs(\tau_j, t) \cdot X_{i,j} \cdot L_{max}$. As illustrated in Ex. 2, a preempting job of τ_j can cause multiple retries. Therefore, we account for the total local interference on P^* over an interval t as $B_L^{(PDC)}(P^*, t) = \sum_{\tau_i \in \Gamma(P^*)} \sum_{\tau_j \in \Gamma(P^*)} nljobs(\tau_j, t) \cdot X_{i,j} \cdot L_{max}$.

A.1.4 Remote Transaction Interference

For considering the interference caused by transactions remote to P^*, we begin by building intuition about the types of interference that can occur. Then we describe how we compute an upper-bound on interference.

As we saw with local transactions, a remote transaction writing to a resource can force a transaction on P^* to retry. In fact, when the read-set is validated in Step 5 of TL2, if any of the memory locations in the read-set are locked by a different thread, the transaction must retry.

Example 3 *Consider the execution scenario illustrated in Fig. A-3.* $\mathtt{Tx}_1$ *begins executing at* $t = 2$*. After executing Steps 1-4, it must check its read-set. However, at* $t = 4$*,* $\mathtt{Tx}_2$ *has locked* ℓ_a*. Thus, the read-set validation (Step 5) for* $\mathtt{Tx}_1$ *fails, forc-*

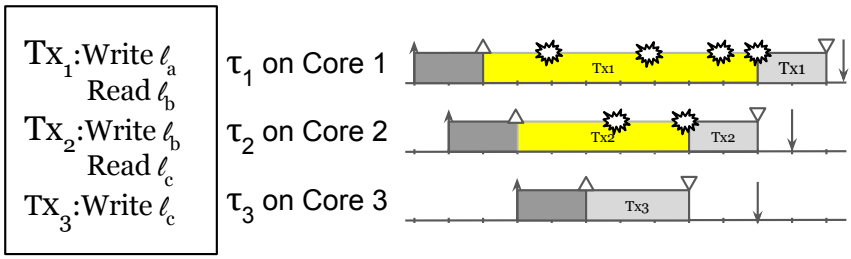


Figure A-3: A possible execution pattern that illustrates remote interference.

ing a retry. (This is the specified behavior for TL2 [20]; a transaction observing the write-lock held cannot progress, as the value read during the speculative execution is anticipated to change.) The same read-set invalidation occurs for $\mathtt{Tx}_1$ *at* $t = 8$*, and similarly* $\mathtt{Tx}_2$ *is forced to retry (caused by* $\mathtt{Tx}_3$ *holding the write-lock on* ℓ_c*) at* $t = 7$*.* ◊

Ex. 3 highlights that a transaction (here, $\mathtt{Tx}_1$) can repeatedly retry while a conflicting transaction (here, $\mathtt{Tx}_2$) holds a write-lock. During that time period, that conflicting transaction (here, $\mathtt{Tx}_2$) can also be forced to retry in the same manner. Thus, even without considering a specific lock implementation, it is clear that other tasks that write an object shared between the considered transactions can cause delay for up to a duration of L_{max}.

Example 4 (cont'd) *When looking at the duration of time for which* $\mathtt{Tx}_1$ *is executing only to be forced to retry, we see that this is longer than simply the duration of* $\mathtt{Tx}_2$*, even though* $\mathtt{Tx}_1$ *only overlaps resources with* $\mathtt{Tx}_2$*. Here,* $\mathtt{Tx}_3$ *is transitively interfering with* $\mathtt{Tx}_1$*.* ◊

To detect all possible transitive interference, we must consider all possible transactions that overlap in some way (even transitively) with a given transaction. Doing so can be accomplished by constructing a graph of transactions, with edges added between possible conflicts, and determining which transactions are reachable.

Though we have been looking at these individual resources, recall that any larger set that may overlap can correspond to a broader data structure, and that we constrain our evaluation to transactions operating on only one data structure. Addition-

ally, for real-time analysis, we must assume that all accesses to a given data object may interfere, as we cannot determine which elements will be used ahead of time. Thus, $\texttt{Tx}_1$, $\texttt{Tx}_2$, and $\texttt{Tx}_3$ must be considered as if they all access the same data object and can conflict as shown in Ex. 3. (Because they operate on separate elements at runtime as supported by TL2, they cannot be assumed to simply execute their transactions sequentially.)

Unlike with local interference, multiple transactions from a remote job can cause interference on P^*. Additionally, due to the possibility of preemptions, a single remote transaction may cause multiple retries on P^*.

When analyzing τ_i on P^*, we let $Y_{i,j}$ be the number of transactions of a single job of τ_j (on a partition remote to P^*) that conflict with any of τ_i's transactions. As in prior work [8], we define the number of remote jobs of τ_j as $nrjobs(\tau_j, t) = \lceil \frac{t+d_j}{p_j} \rceil$. Thus, across all tasks on P^*, we estimate the remote interference as $B_R^{(PDC)}(P^*, t) = \sum_{\tau_i \in \Gamma(P^*)} \sum_{\tau_j \notin \Gamma(P^*)} nrjobs(\tau_j, t) \cdot Y_{i,j} \cdot L_{max}$.

A.1.5 Overall Computation of Synchronization Delay

Based on the local and remote interference that can be incurred by P^*, we compute $B^{(PDC)}(P^*, t) = B_L^{(PDC)}(P^*, t) + B_R^{(PDC)}(P^*, t)$.

As in prior work [8], on the processor of interest, the term $B^{(AC)}(P^*, t)$ denotes the synchronization delay for a busy-period of length t, and the term $B^{(PDC)}(P^*, t)$ denotes the sum of arrival blocking and synchronization delay for a deadline busy-period of length t. Above, we described our approach for computing $B^{(PDC)}(P^*, t)$. Because we assume preemptive execution, there is no arrival blocking included in that term. To instead compute $B^{(AC)}(P^*, t)$, we simply change the definition of *nljobs* to $nljobs(\tau_i, t) = \lceil \frac{t}{p_i} \rceil$. This reflects that the bound $B^{(AC)}(P^*, t)$ must consider all jobs released in the window of length t, not only those that have a deadline before the end of the window [8].

Appendix B

Schedulability Metrics and Figures

Here we further describe metrics used to evaluate the schedulability results, as well as include all graphs (for each of the 72 scenarios) from our study.

B.1 Task Schedulable Area Ratio

As discussed in Sec. Sec. 5.2, we compute the *task schedulable area* (TSA) for each of the figures by calculating the area under the schedulability curve using a midpoint sum. We compared TORTIS and TL2 on the basis of two ratios:

$$\text{Ratio 1} = \frac{\text{TSA}(\textsf{TORTIS})}{\text{TSA}(\textsf{TL2})}$$

and

$$\text{Ratio 2} = \frac{\text{TSA}(\textsf{TORTIS}) - \text{TSA}(\textsf{TL2})}{\text{TSA}(\textit{no synchronization}) - \text{TSA}(\textsf{TL2})}.$$

These ratios capture the difference between TORTIS and TL2 and the ratio of improvement with regards to total possible improvement. After calculating these ratios for all 72 experiments, statistics presented in Table B.1 allowed us to select the representative cases and evaluate TORTIS's overall schedulability improvements.

	Ratio 1	Ratio 2
MAX	10.19	0.95
MIN	1.01	0.02
AVG	2.46	0.49
MEDIAN	1.77	0.51

Table B.1: Statistics of TSA ratio calculations.

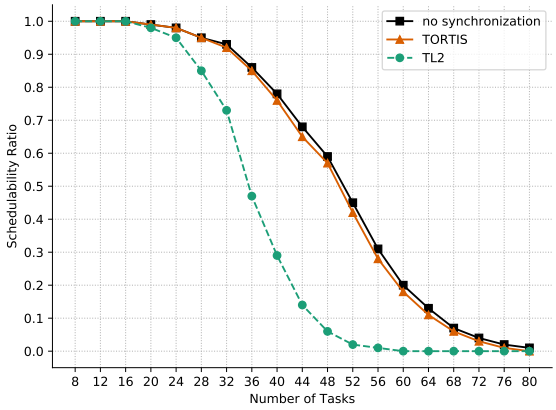


Figure B-1: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

B.2 Complete Figures

Below we include all of the graphs from our schedulability experiments in order of decreasing value of Ratio 2.

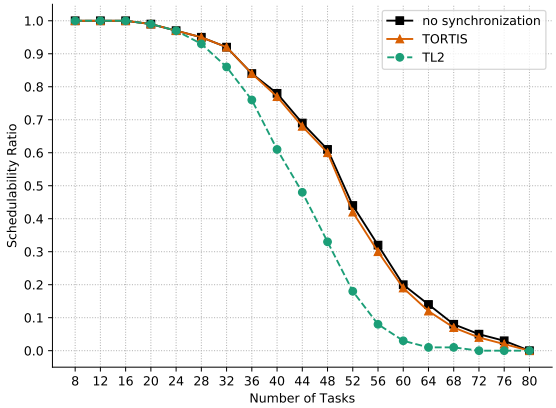


Figure B-2: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

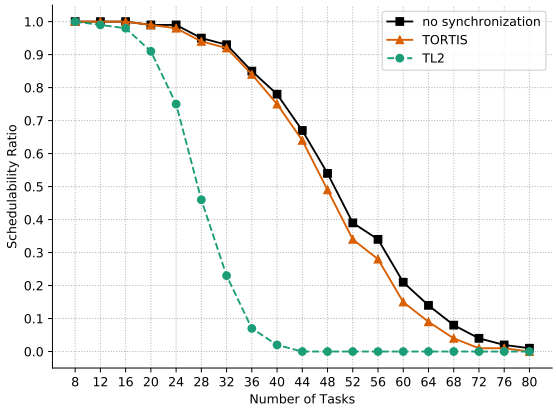


Figure B-3: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

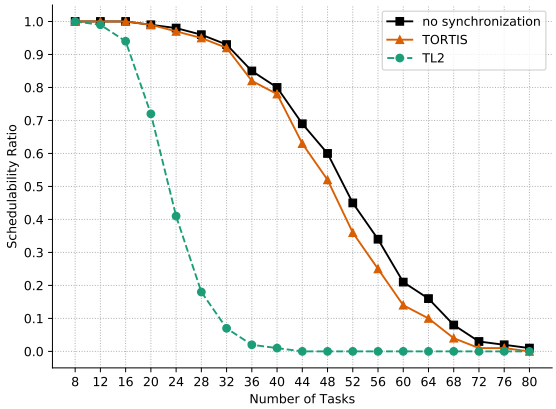


Figure B-4: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

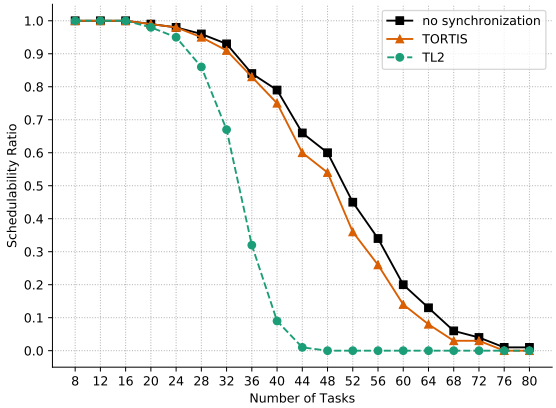


Figure B-5: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

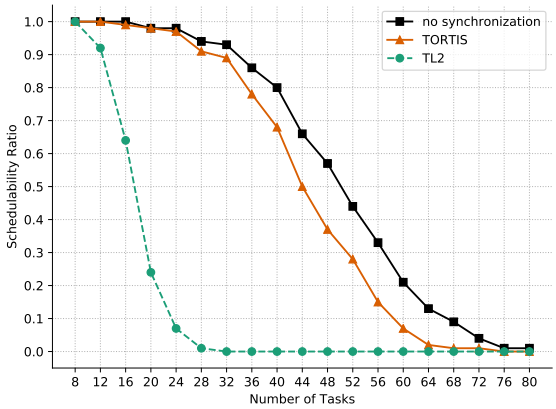


Figure B-6: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

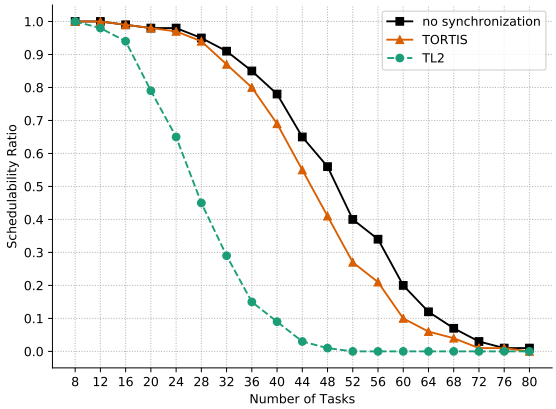


Figure B-7: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

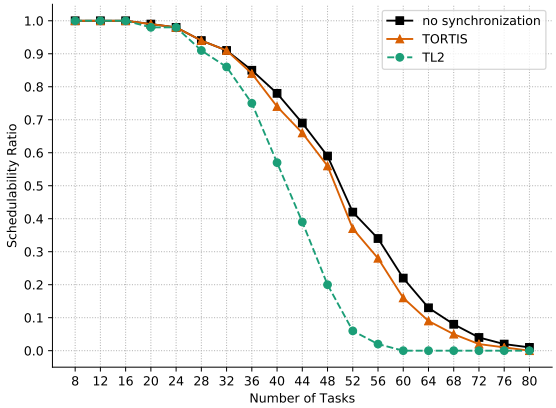


Figure B-8: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

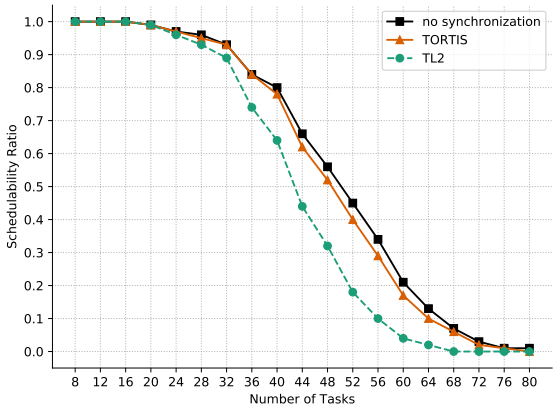


Figure B-9: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

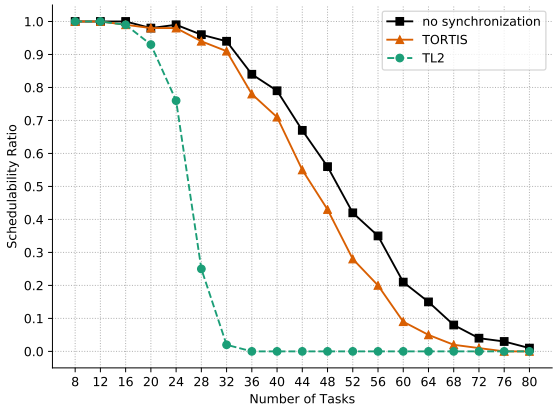


Figure B-10: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

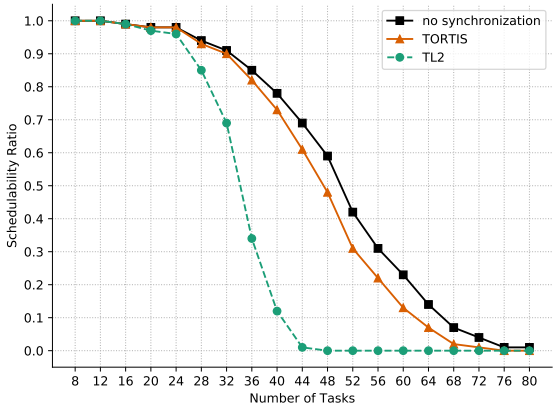


Figure B-11: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

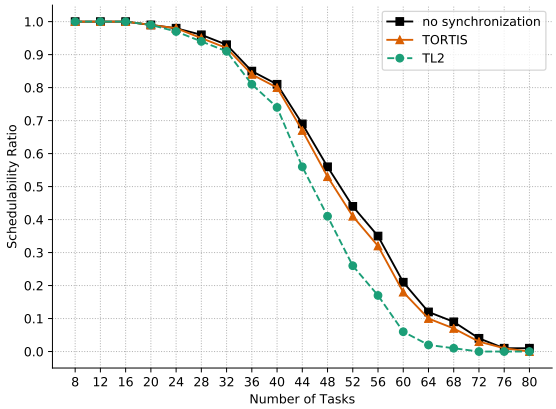


Figure B-12: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

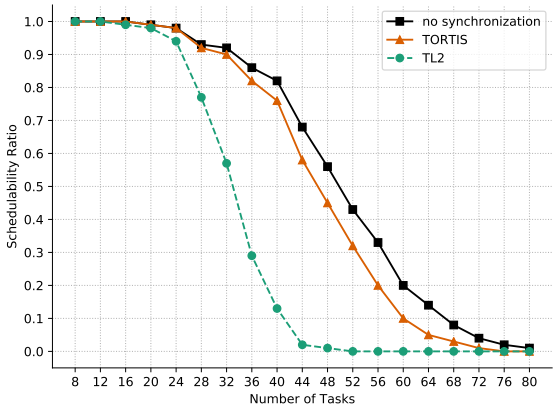


Figure B-13: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

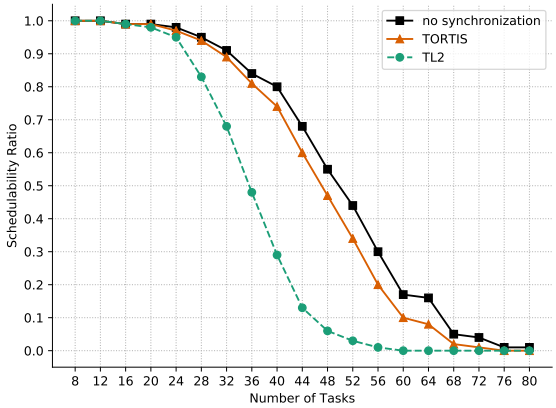


Figure B-14: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

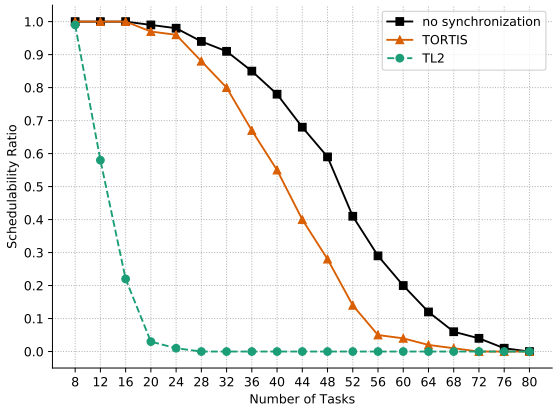


Figure B-15: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

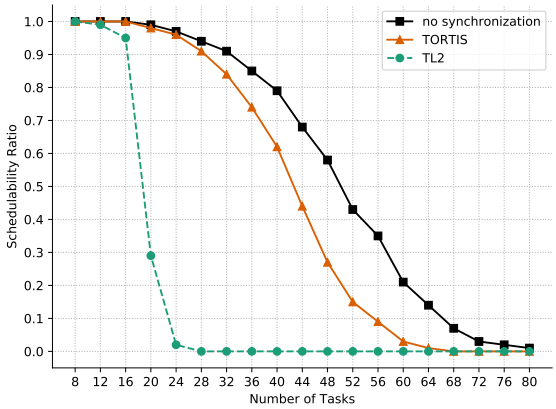


Figure B-16: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

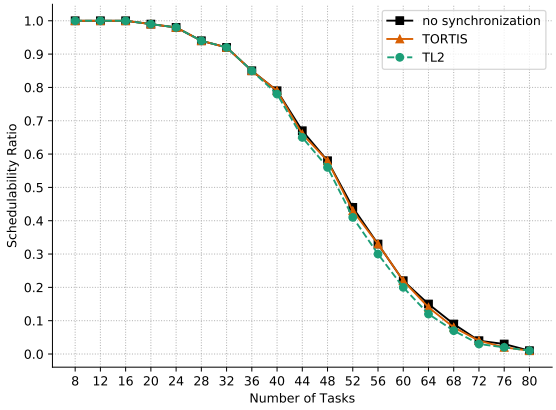


Figure B-17: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

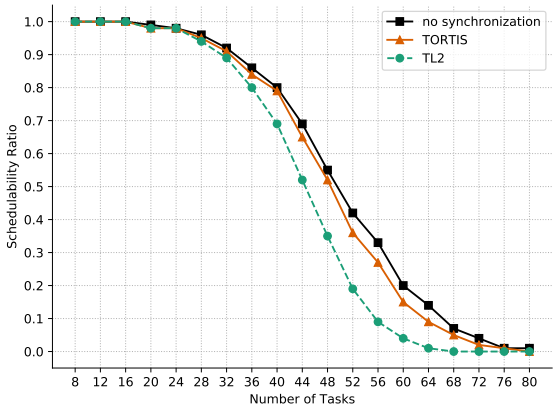


Figure B-18: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

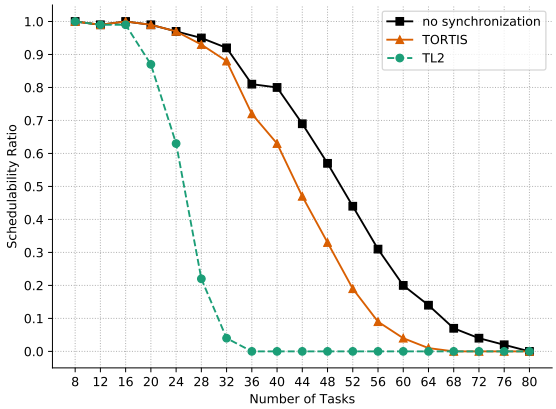


Figure B-19: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

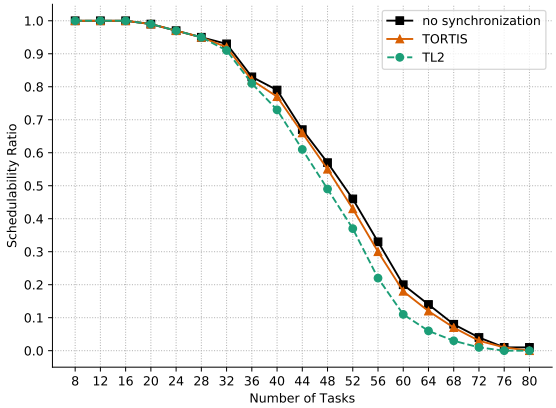


Figure B-20: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

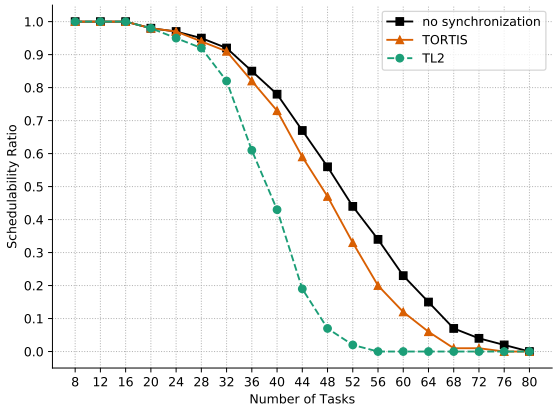


Figure B-21: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

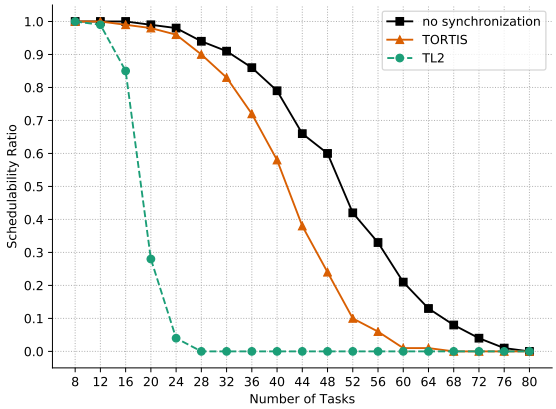


Figure B-22: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

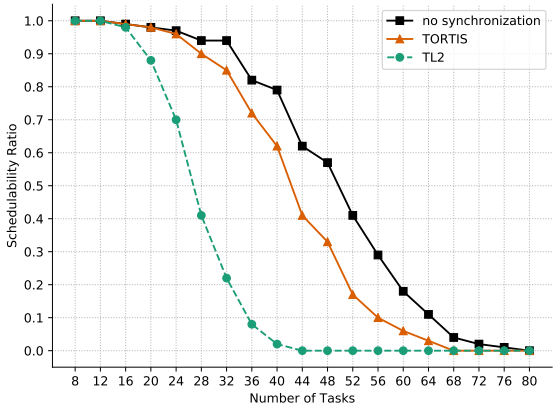


Figure B-23: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

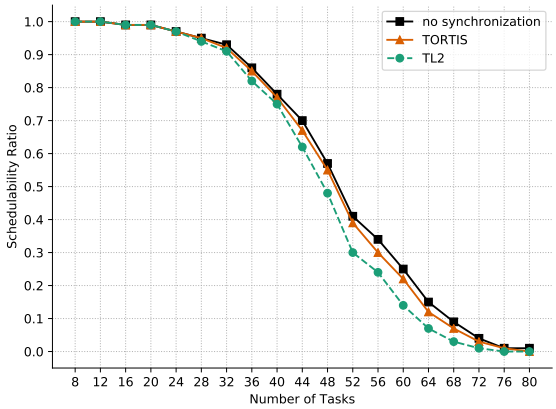


Figure B-24: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

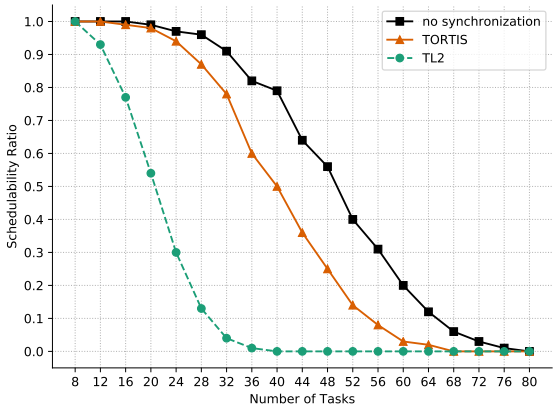


Figure B-25: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

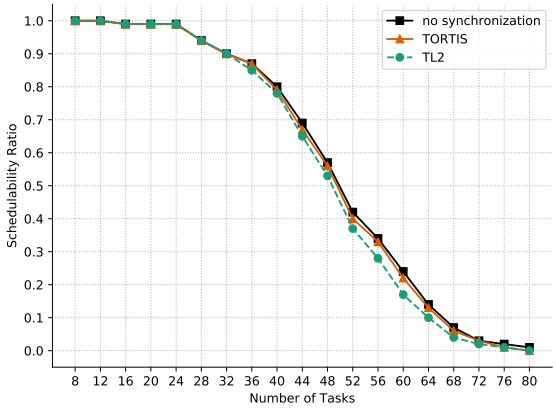


Figure B-26: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

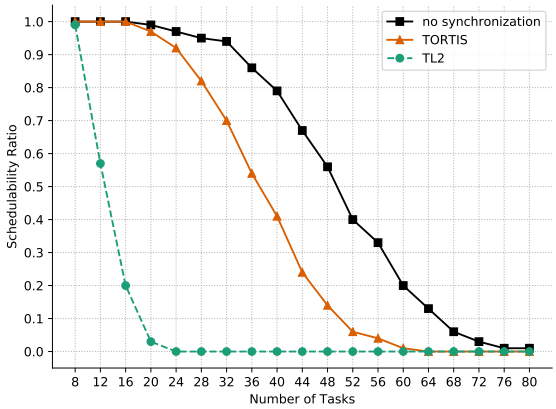


Figure B-27: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

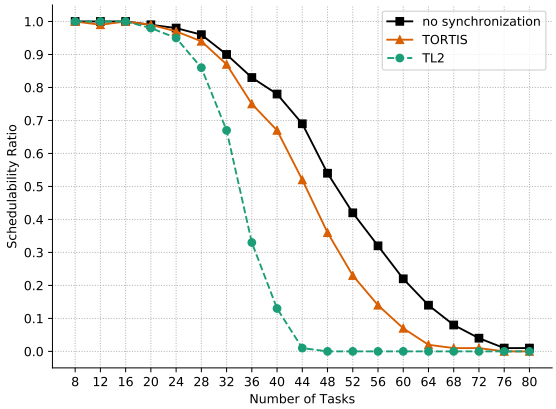


Figure B-28: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

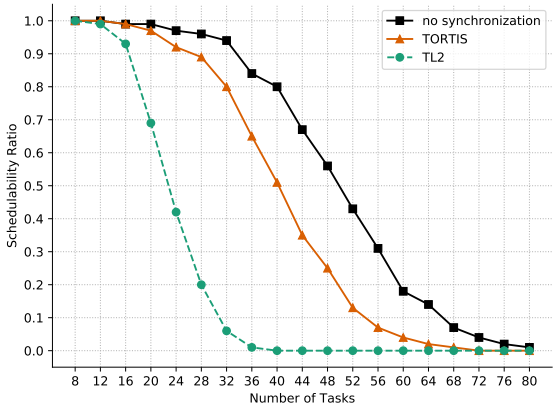


Figure B-29: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

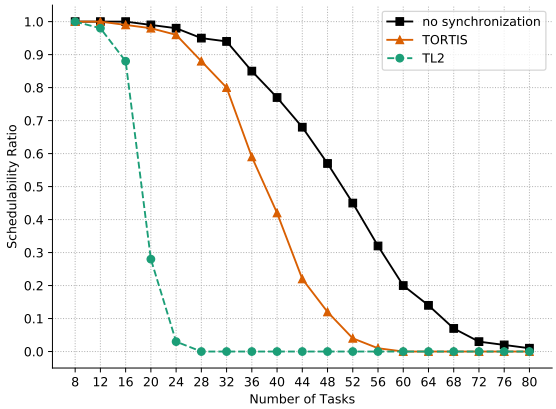


Figure B-30: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

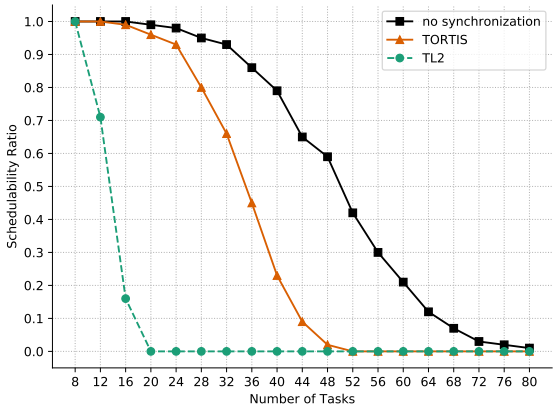


Figure B-31: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

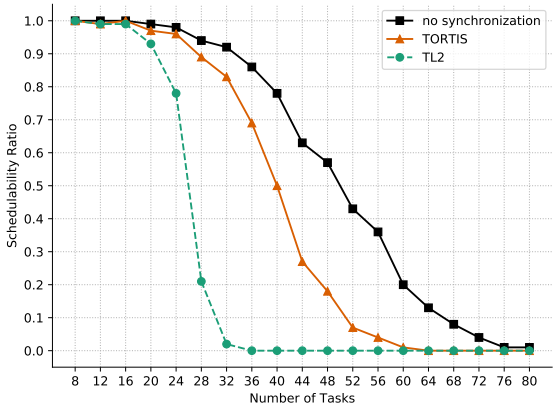


Figure B-32: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

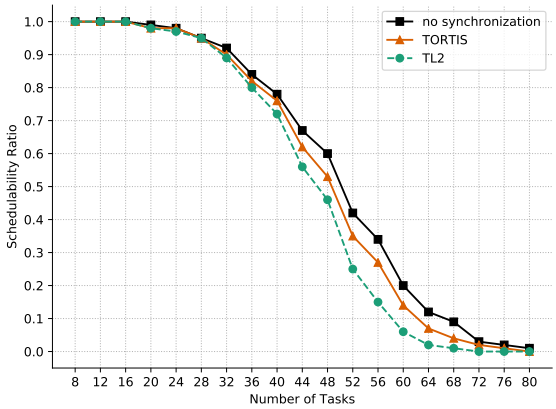


Figure B-33: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

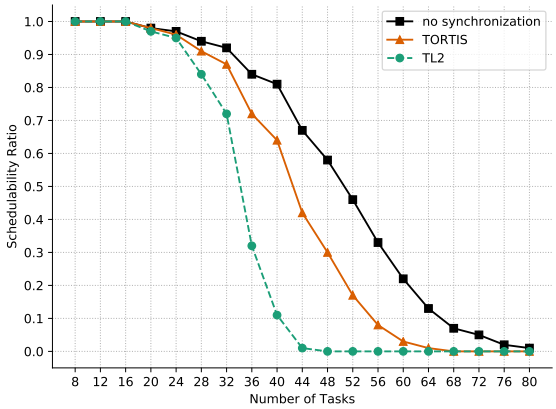


Figure B-34: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

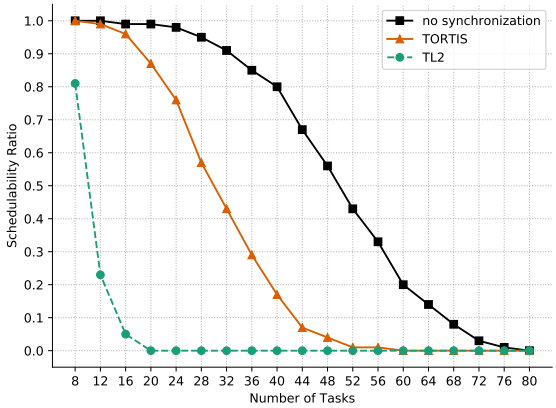


Figure B-35: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

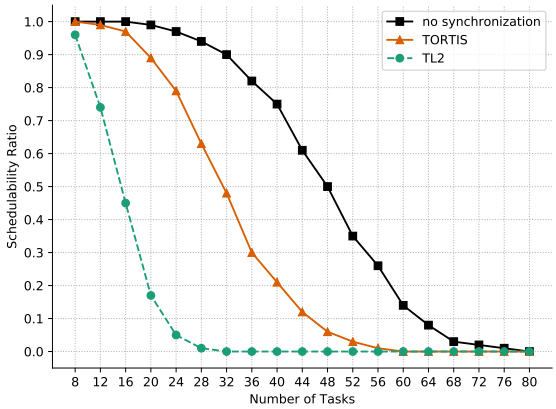


Figure B-36: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with with a single transaction.

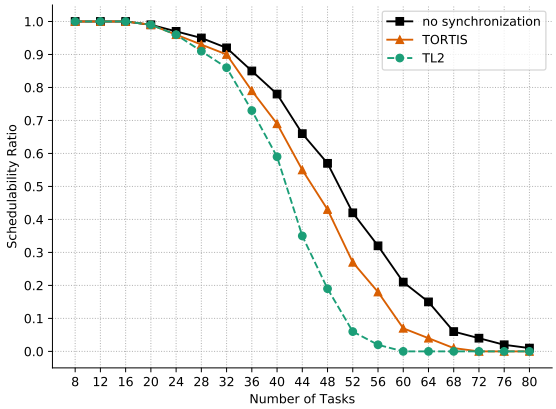


Figure B-37: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

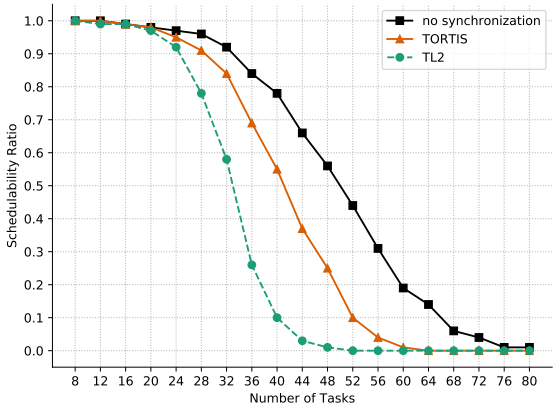


Figure B-38: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

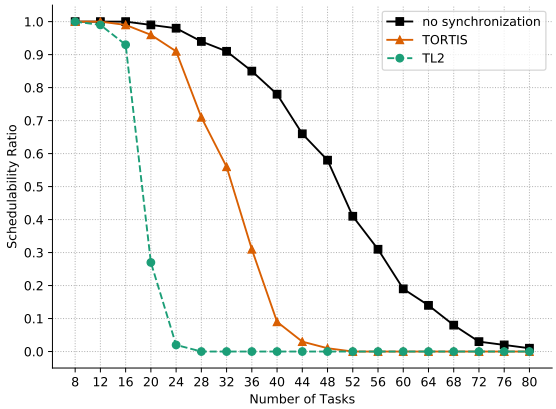


Figure B-39: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

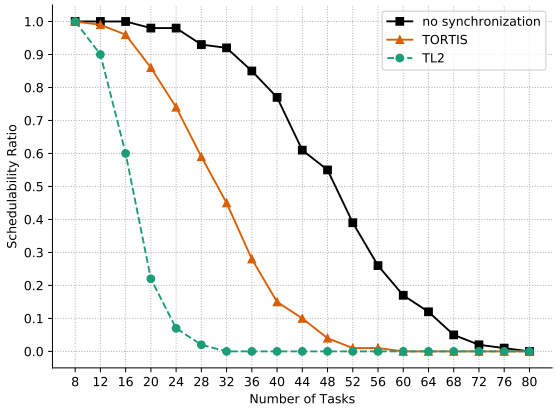


Figure B-40: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

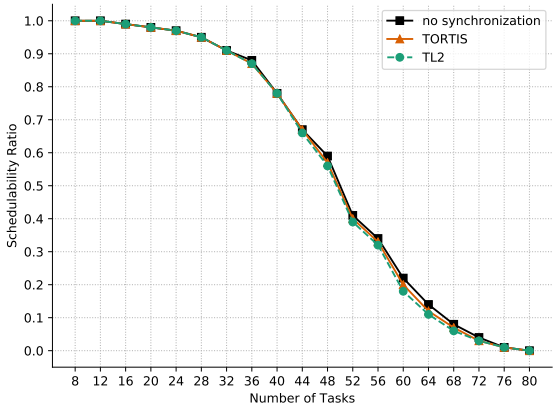


Figure B-41: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

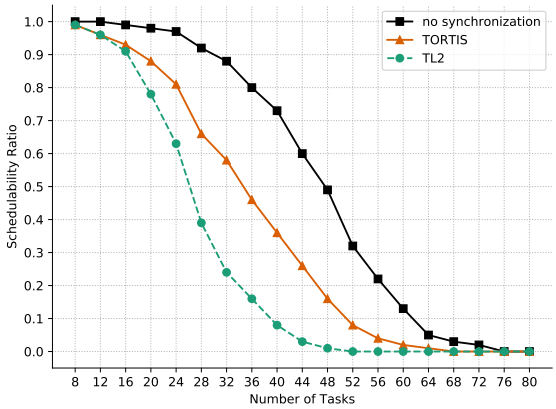


Figure B-42: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

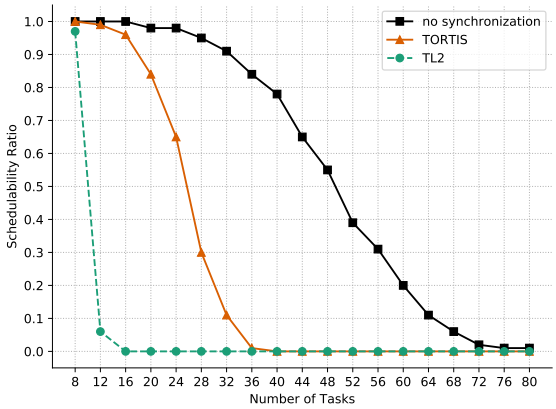


Figure B-43: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

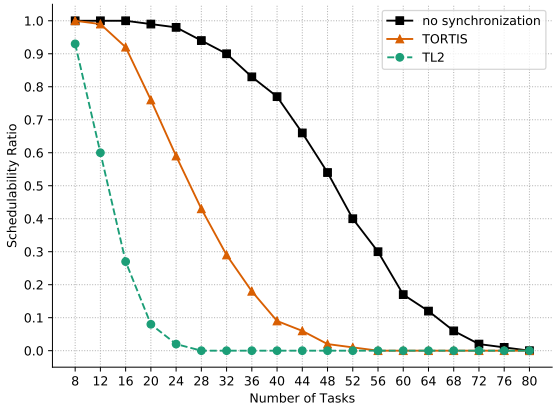


Figure B-44: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

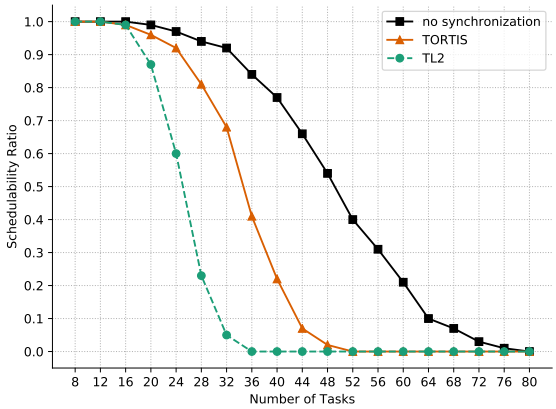


Figure B-45: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

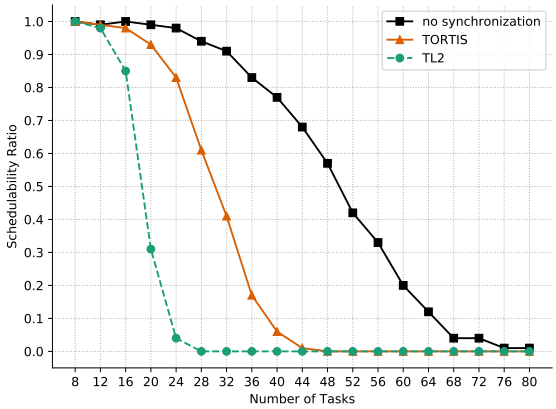


Figure B-46: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

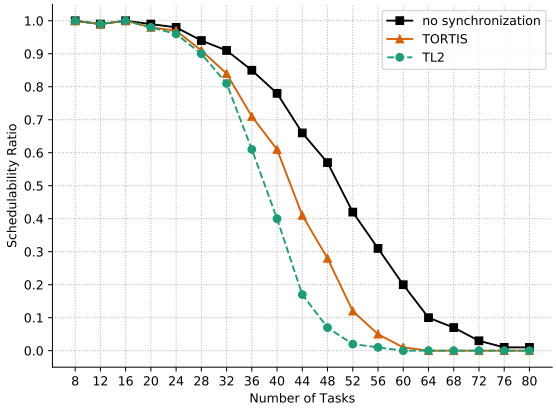


Figure B-47: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

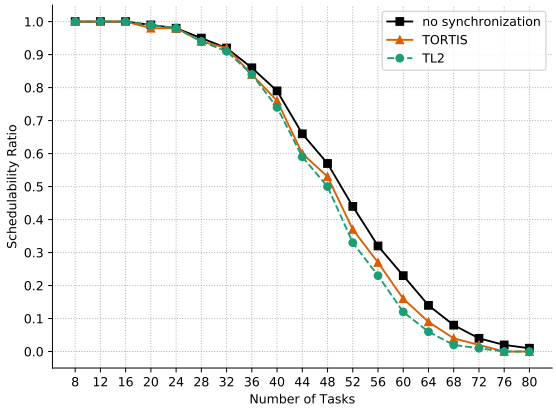


Figure B-48: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

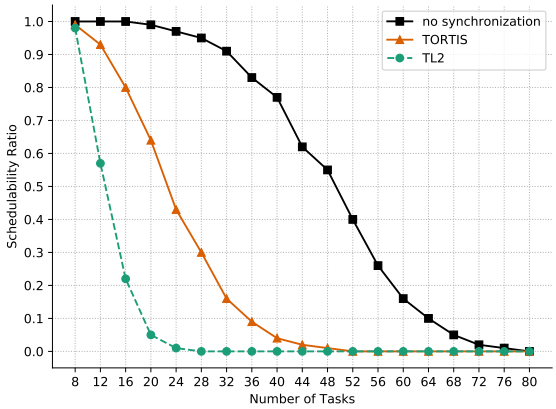


Figure B-49: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

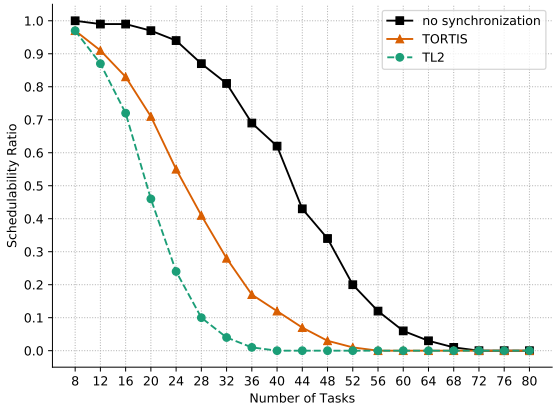


Figure B-50: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

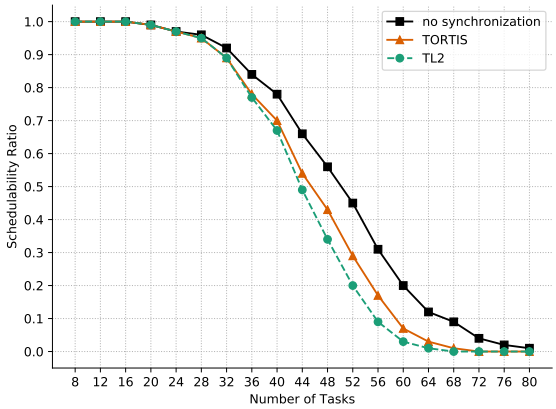


Figure B-51: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

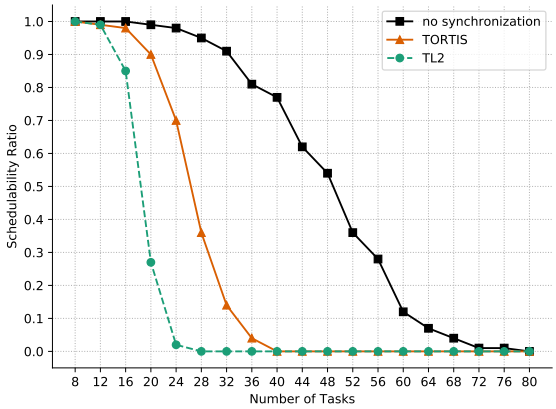


Figure B-52: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

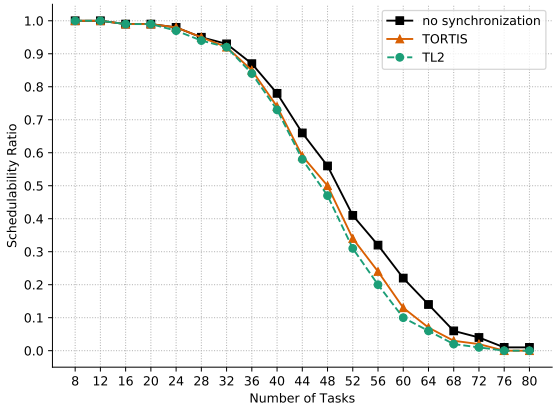


Figure B-53: Results from the scenario with periods selected from $[10ms, 100ms]$, 4 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

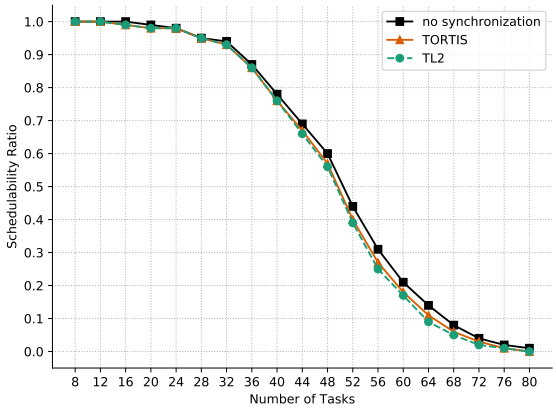


Figure B-54: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

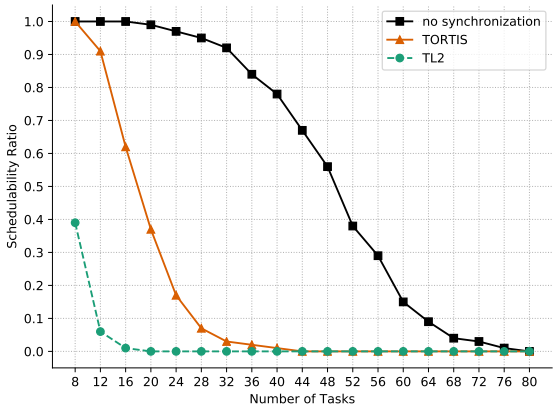


Figure B-55: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

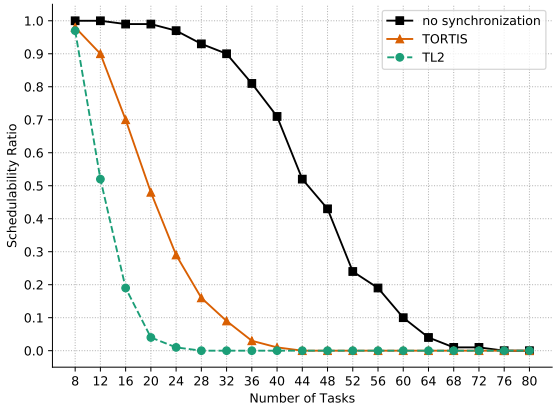


Figure B-56: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

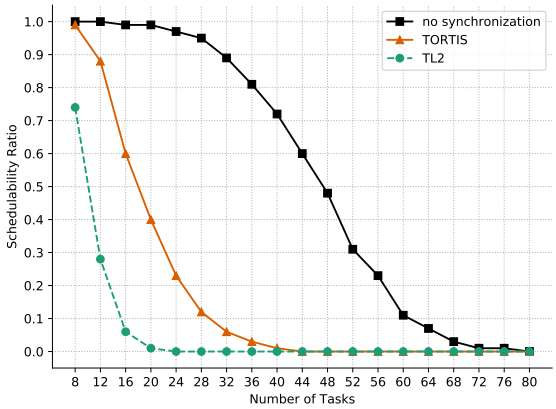


Figure B-57: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

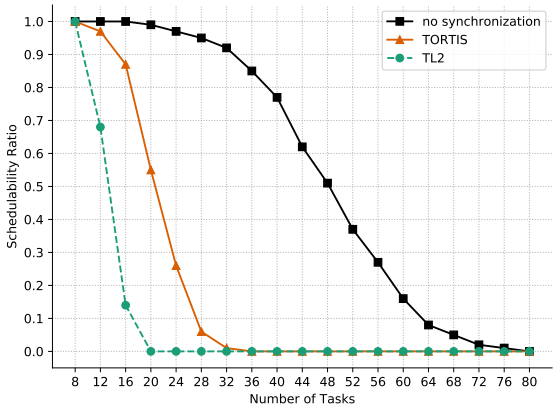


Figure B-58: Results from the scenario with periods selected from $[10ms, 100ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

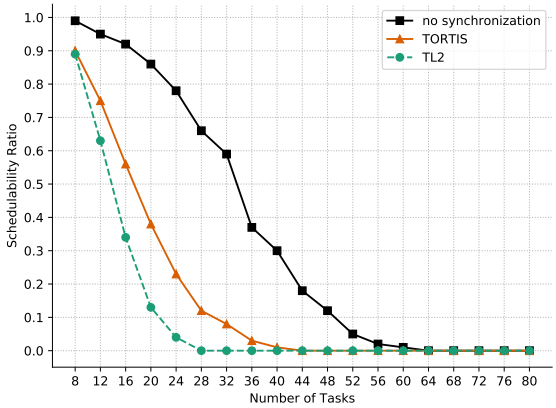


Figure B-59: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.1. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

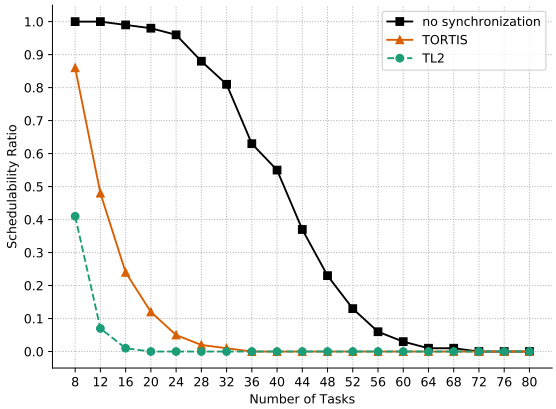


Figure B-60: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with with a single transaction.

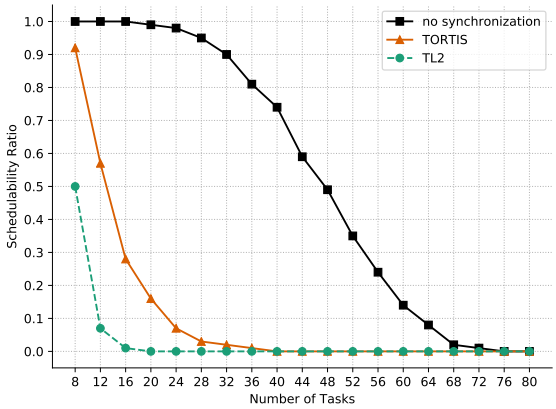


Figure B-61: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

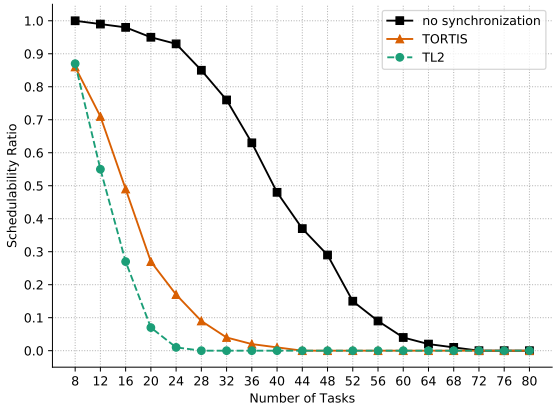


Figure B-62: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

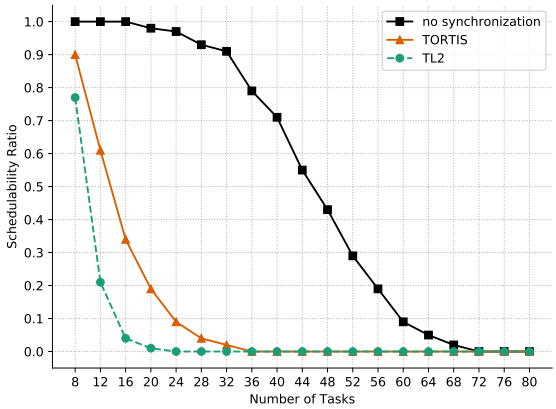


Figure B-63: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

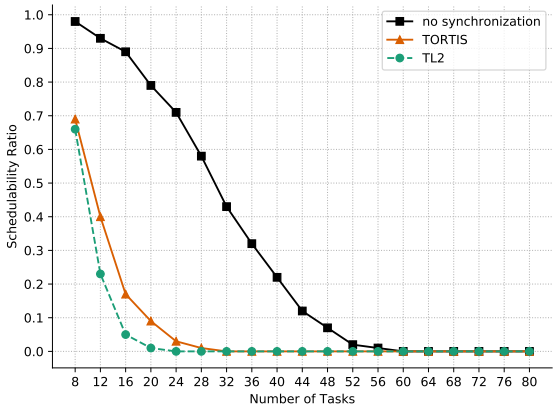


Figure B-64: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

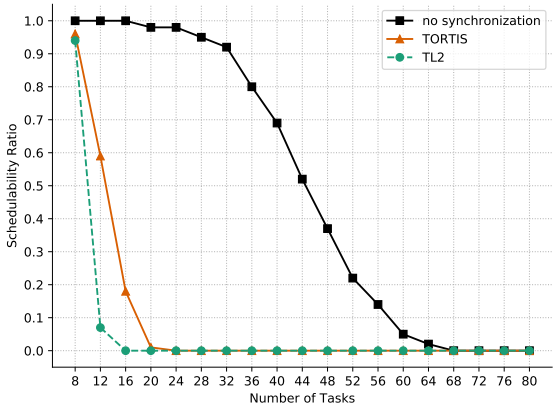


Figure B-65: Results from the scenario with periods selected from $[10ms, 100ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

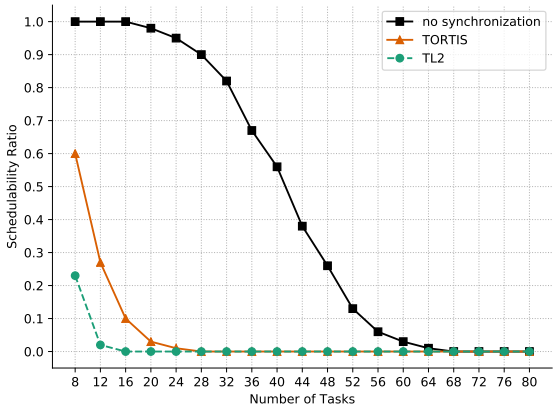


Figure B-66: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

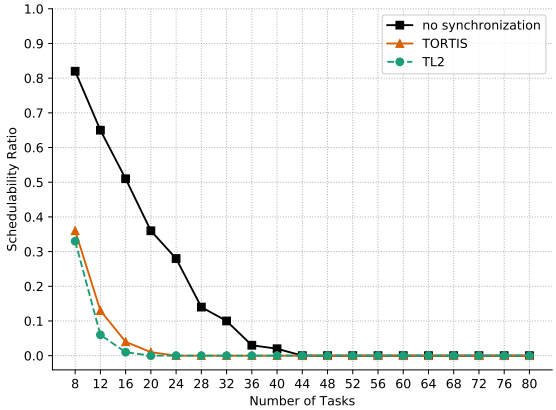


Figure B-67: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.25. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

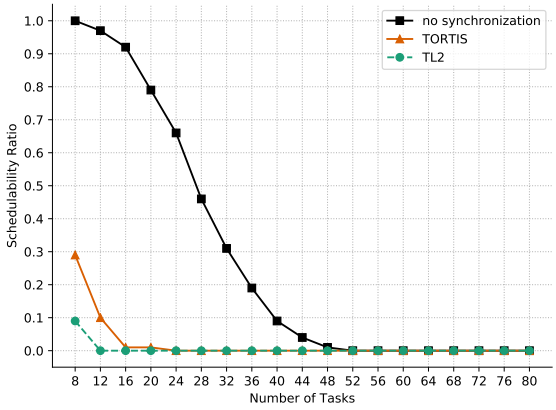


Figure B-68: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with with a single transaction.

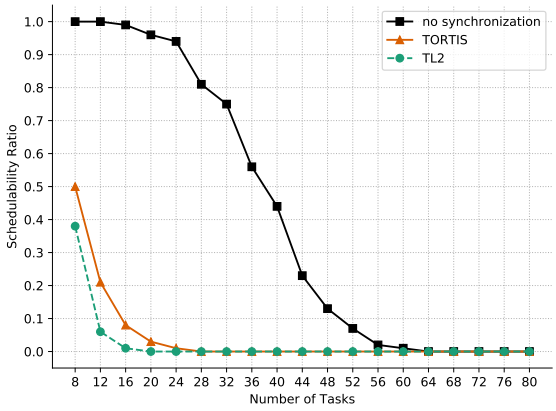


Figure B-69: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

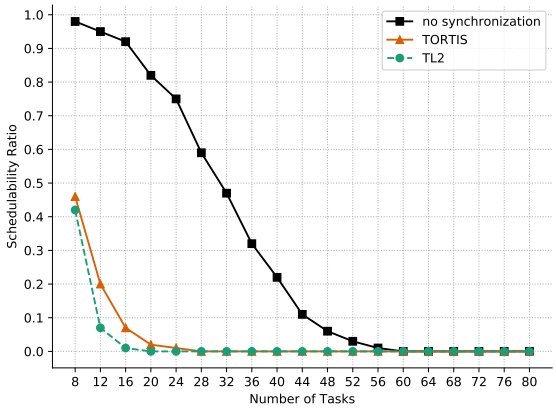


Figure B-70: Results from the scenario with periods selected from $[1ms, 1000ms]$, 4 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

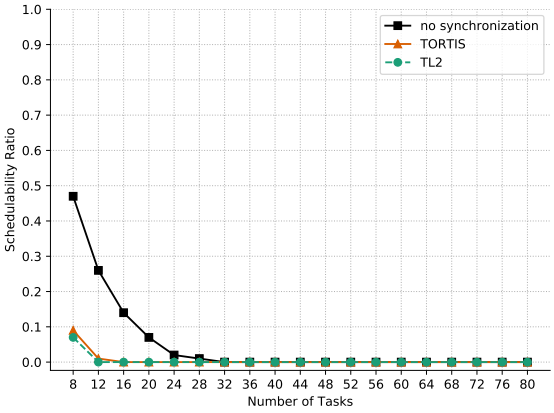


Figure B-71: Results from the scenario with periods selected from $[1ms, 1000ms]$, 16 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

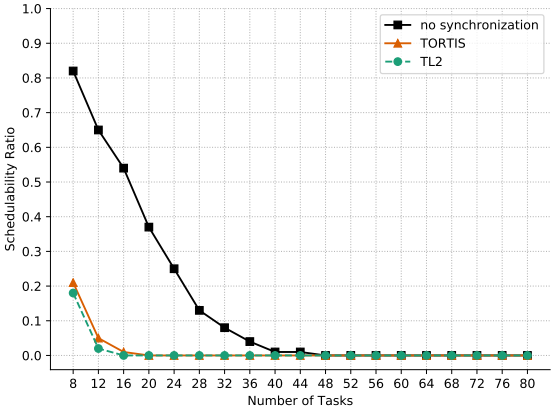


Figure B-72: Results from the scenario with periods selected from $[1ms, 1000ms]$, 8 data objects, and tasks accessing objects with probability 0.5. Tasks accessing a data object did so with a number of transaction selected from $\{1, \ldots, 5\}$.

Appendix C

Throughput Benchmark Results

Here we include additional graphs for throughput benchmarks with varying parameters.

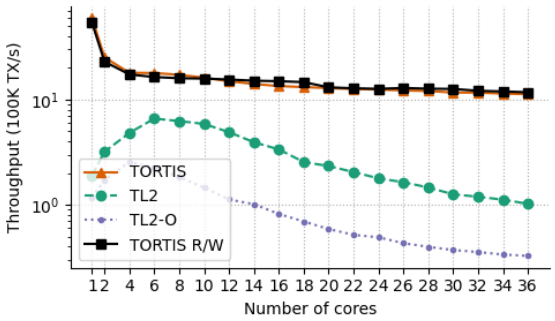


Figure C-1: Varying number of cores. 128-element buffer, 10% of elements accessed, 5% writes.

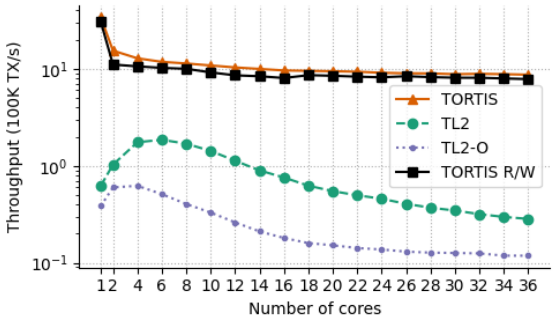


Figure C-2: Varying number of cores. 256-element buffer, 10% of elements accessed, 5% writes.

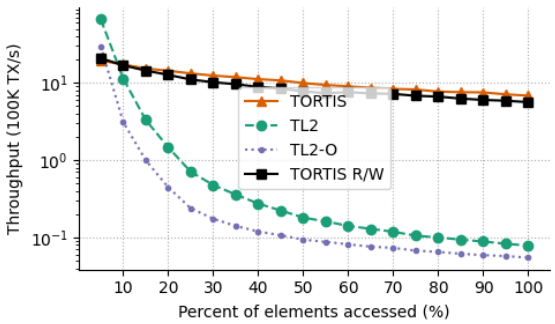


Figure C-3: Varying the number of elements accessed. 16 cores, 64-element buffer, 5% writes.

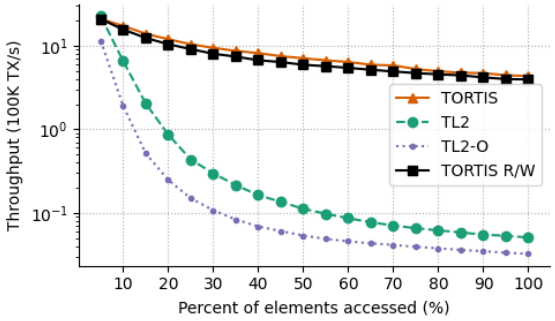


Figure C-4: Varying the number of elements accessed. 8 cores, 128-element buffer, 5% writes.

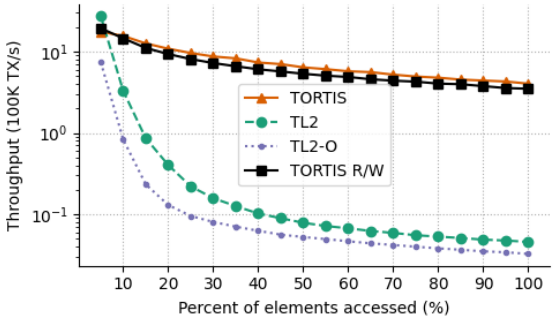


Figure C-5: Varying the number of elements accessed. 16 cores, 128-element buffer, 5% writes.

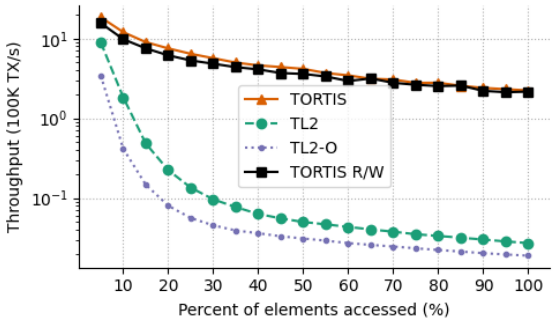


Figure C-6: Varying the number of elements accessed. 8 cores, 256-element buffer, 5% writes.

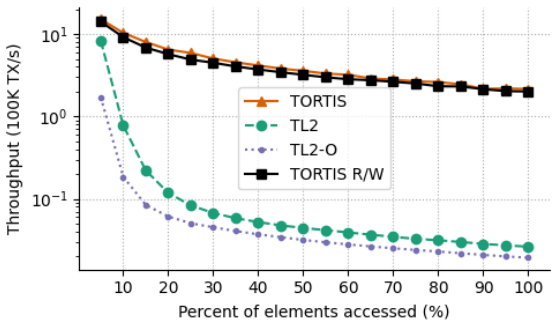


Figure C-7: Varying the number of elements accessed. 16 cores, 256-element buffer, 5% writes.

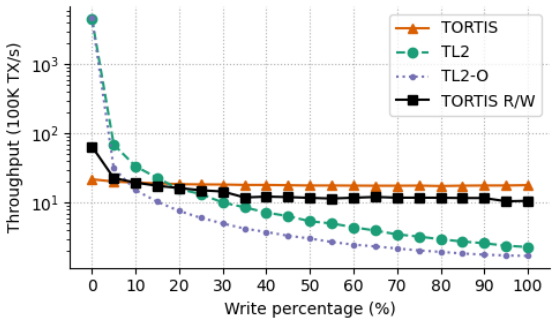


Figure C-8: Varying the write percentage. 16 cores, 64-element buffer, 10% of elements accessed.

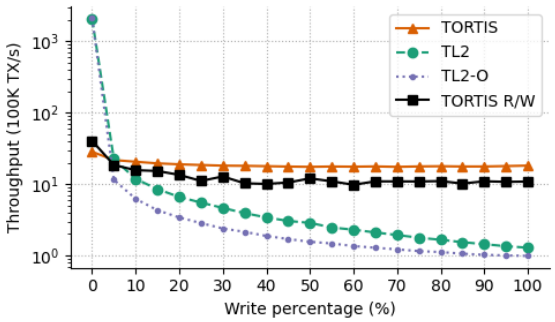


Figure C-9: Varying the write percentage. 8 cores, 128-element buffer, 10% of elements accessed.

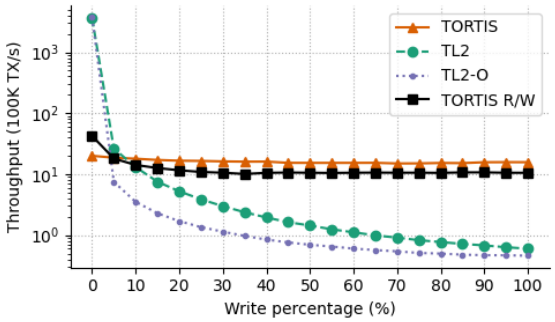


Figure C-10: Varying the write percentage. 16 cores, 128-element buffer, 10% of elements accessed.

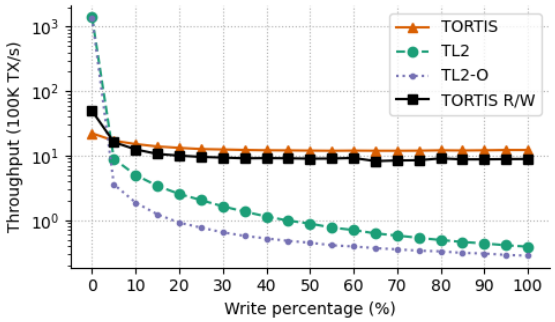


Figure C-11: Varying the write percentage. 8 cores, 256-element buffer, 10% of elements accessed.

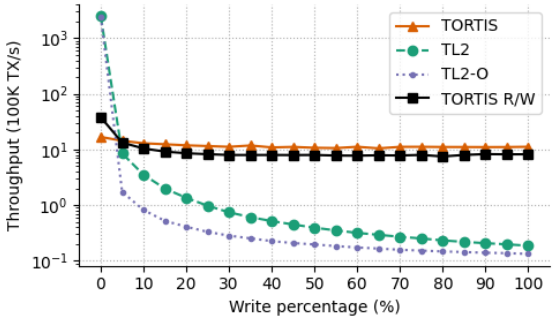


Figure C-12: Varying the write percentage. 16 cores, 256-element buffer, 10% of elements accessed.

Bibliography

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

[2] std::cell - rust. `https://doc.rust-lang.org/std/cell/`, 2020. commit a879f9c.

[3] James H Anderson and Mark Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999.

[4] 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*, 61(10):553–566, 2015.

[5] Alexis Beingessner and Steve Klabnik. The rustonomicon: The dark arts of unsafe rust. `https://doc.rust-lang.org/nomicon/`, 2016.

[6] C. Belwal and A. Cheng. Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective. *Embedded Systems Letters*, 3(1):37–41, March 2011.

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

[8] 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*, pages 39–49. IEEE, 2016.

[9] A. Biondi, B. Brandenburg, and A. Wieder. A blocking bound for nested FIFO spin locks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*, pages 291–302. IEEE, 2016.

[10] 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*, pages 71–80. IEEE, August 2007.

[11] Aaron Block, Hennadiy Leontyev, Bjorn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE international conference on embedded and real-time computing systems and applications (RTCSA 2007)*, pages 47–56. IEEE, 2007.

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

[13] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60. IEEE Press, December 2010.

[14] Björn Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, 2011.

[15] Björn B Brandenburg. The fmlp+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 61–71. IEEE, 2014.

[16] Björn B Brandenburg and James H Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 184–193. IEEE, 2009.

[17] Björn B Brandenburg, John M Calandrino, Aaron Block, Hennadiy Leontyev, and James H Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE, 2008.

[18] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 2017.

[19] Andreia Correia, Pedro Ramalhete, and Pascal Felber. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–116, 2020.

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

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

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

[23] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 48–60, 2005.

[24] 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*, pages 289–300. ACM, 1993.

[25] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[26] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[27] R. Hussain and S. Zeadally. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys and Tutorials*, 21(2):1275–1313, 2019.

[28] M. Jones. What really happened on Mars Rover Pathfinder. *The Risks Digest*, 19(49), 1997.

[29] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

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

[31] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, 1987.

[32] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[33] Nicholas D Matsakis. Introducing mir. `https://blog.rust-lang.org/2016/04/19/MIR.html`, 2016.

[34] C. Nemitz, T. Amert, and J. Anderson. Real-time multiprocessor locks with nesting: Optimizing the common case. *Real-Time Systems*, 55(2):296–348, 2019.

[35] Catherine E Nemitz. New approaches to contention-sensitive nested locking in real-time systems. *JRWRTC 2017*, page 13, 2017.

[36] Glenn E Reeves. What really happened on mars? 1998.

[37] 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*, pages 477–485, Aug 2009.

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

[39] 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*, pages 279–284, 2010.

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

[41] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2015.

[42] B. Ward. *Sharing Non-Processor Resources in Multiprocessor Real-Time Systems*. PhD thesis, 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*, pages 223–232, 2012.

[44] Bryan C Ward and James H Anderson. Supporting nested locking in multiprocessor real-time systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 223–232. IEEE, 2012.

[45] 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*, pages 169–178, 2008.