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

Anonymous Submission #149

***Index Terms*—transactional memory, Rust, real-time systems**

The artifact (including the virtual machine) is available for download here: https://github.com/scaspin/RTSS21-Artifact

I. Introduction

This document provides an overview of how to replicate the results and figures from our paper *TORTIS: Retry-Free Software Transactional Memory for Real-Time Systems*. We provide information about:

- the results can be reproduced with our artifact (Sec. II),
- the platform on which we conducted experiments and the platform expectations for reproducability (Sec. III),
- how to set up the environment to run our evaluations (Sec. V),
- how to run the artifact (Sec. VI)
- the licenses for our code (Sec. VIII),
- the most relevant files for each evaluation (Sec. VII), where we outline code structure and also explain certain files or changes, particulalry for the Rust compiler code.

As described in the paper, our results come from three separate evaluations: a schedulability evaluation, a throughput evaluation using different data structures and access patterns, and a throughput evaluation of a demo scenario. We also provide a detailed explanation for compiler modifications made as part of TORTIS, highlighting appropriate files. Thus, for each of the above categories, we provide the details for each component.

II. Scope

All figures can be replicated with different our artifact package. The schedulability evaluation results in data summarized by three observations and highlighted in Figure 2. The throughput evaluation using different data structures and access patterns in Figures 5-9. The throughput evaluation of a demo scenario in Figure 10.

Details of how to reproduce the results from each of these evaluation are specified in the README files corresponding to the given evaluation, although some dependencies and install instructions are specified here for ease of use.

The general trends of the schedulability evaluation are repeatable. Because random task sets are generated, the exact schedulability ratios may vary slightly.

III. Platform

The figures in the paper were generated by running these experiments on a two-socket, 18-cores-per-socket x86 machine running the Linux 4.9.30 LITMUS^{RT}kernel [2] with two Intel Xeon E5-2699 v3 CPUs @ 2.30GHz with 128GB RAM and 32KB L1, 256KB L2, and 46080KB L3 caches. We performed the evaluations on up to 36 cores, with the first 18 cores on the same socket. The performance and figures vary wildly across platforms, and some of the observations highlighted in the paper may not be supported by running the experiments in a virtual machine.

We tested the VM with 4GB of RAM, 20GB of storage, which was needed to build the compiler. The VM submitted as part of the artifact evaluation process can be used with less memory as long as the compiler is not built again. The artifact evaluation is designed to run on a multiple-core machine. We recommend running on a research machine if you cannot allow the VM access to more than one core on your personal computer.

A. Schedulability Evaluation

The reproducibility of our results does not depend on the underlying platform, but the time which it takes to produce the output does. We originally conducted our schedulability evaluation on a platform with 36 cores, and running the full set of experiments took about 14 hours.

The potentially long running time can be mitigated in several ways. How to make these changes is described in `README.TXT`. First, the number of samples for each point can be reduced. This will result in curves that are less smooth, but will still illustrate the general trend. Second, specific scenarios can be run. For example, the experiment setup file can be modified to produce only the output of Figure 2. Third, the scenarios can be split up and run in batches. For example, starting by holding all values constant except the write probability can give an approximation of how long the entire set of experiments will take (and also, how many parameters to vary within a single run).

If you would like to run the schedulability study on a research machine (not within the VM), copy off all of the files. You can use make to compile the code within directory *lib/schedcat*. Running this evaluation requires the following:

- Python 2.7
- Python NumPy Library
- Python SciPy Library
- SWIG 3.0

- GNU C++ compiler (G++)
- GNU Multiple Precision Arithmetic Library (libgmp)
- LP Solver (we use GLPK)

B. Compiler

Dependencies are listed in the rust-stm README. If you are building on a linux machine they are as follows:

- g++ 5.1 or later or clang++ 3.5 or later
- python 2.7 (but not 3.x)
- GNU make 3.81 or later
- cmake 3.4.3 or later
- curl
- git
- ssl which comes in libssl-dev or openssl-devel
- pkg-config if you are compiling on Linux and targeting Linux

C. Performance Evaluation

The performance evaluation can only be run with our fork of the compiler.

IV. Downloading Artifact

The artifact can be found through the submission. The artifact can be used by accessing the virtual machine (`artifact/RTSS21.vdi`) using VirtualBox. The artifact also includes standalone directories of the code to be used for local set-up.

V. Local set-up

However, if you want to run this on a research machine then the directories under `artifact/local` will provide all necessary code. Installation instructions for running the artifact locally can be found in each directory's README with a general outline here.

A. Schedulability

All the dependecies listed above must be downloaded prior to building schedCAT. We used GNU Linear Programming Kit (GLPK) for the linear programming solver, although different LP solvers are supported. Schedcat can be built using *make* in subdirectory `lib/schedcat`. Then you can navigate back to the `schedulability` directory and execute the code from there.

B. Compiler

Our fork of the Rust compiler is located in the `rust-stm/` directory. The compiler is already built and ready to run on the provided virtual machine, but the following steps rebuild it from scratch.

To build the compiler for the first time, follow the instructions in `README.md` under ”Installing from source”. Build from the default branch `cnord/rwlock`. For every subsequent build, *check* then *build*. The Rust compiler takes a long time to build, so checking first will quickly let you know if you have some syntax error. The Guide to Rustc Development at elaborates on this.

The compiler can build in our VM, but takes a long time. The "System Requirements" section of the Guide to Rustc Development recommends the following:

- Recommended $\geq 30GB$ of free disk space; otherwise, you will have to keep clearing incremental caches. More space is better, the compiler is a bit of a hog; it's a problem we are aware of.
- Recommended $\geq 8GB$ RAM.
- Recommended ≥ 2 cores; having more cores really helps.
- You will need an internet connection to build; the bootstrapping process involves updating git submodules and downloading a beta compiler. It doesn't need to be super fast, but that can help.

Check with `./x.py check`. Build with `./x.py build -i --stage 1 src/libstd --keep-stage 1`.

C. Performance Evaluation

First, make sure directory `pflock` is checkout to branch *master* and directory `swym` is on branch *cnord/sort*.

The performance evaluation is located in the `transactional-memory/` directory in the `txcell` crate. `txcell` imports the `pflock` and `swym` crates from other directories as shown in Lst. 1.

```
[dependencies]
swym = { path = "../../swym" }
pflock = { path = "../../pflock" }
```

Listing 1. A code sample from the `txcell/Cargo.toml` that specifies dependency locations.

`swym` can be built using *cargo +nightly build* and `pflock` can be built using *cargo build*. Once the compiler and all the supporting are built, there is no other local setup needed to run the evaluation as described in Sec. VI-D.

VI. Using The Artifact

A. Using VM

The username is **user** and the password is **password**. Use the terminal to navigate to and run the files as specified below.

B. Schedulability Evaluation

To run the schedulability evaluation, first navigate to `Documents>schedulability`. The experiments can be started by running `./start_experiment`. This script starts the process of generating random task sets, testing schedulability, storing the output, and graphing the results. To first change the scope of the experiment, go to `exp/pedf_lp.py` and make the modifications described in `README.txt`. To run any piece of the process separately, refer to `README.txt` for the details. Currently the schedulability code in VM is configured to run 100 samples per scenario while the figures in the paper were generated with 1000 samples per scenario. This script may take a while to run depending on the platform and the number of samples. The final plots will be under `plots/$timestamp$/`.

C. Compiler

The compiler is already built, but must be installed. Navigate to `rust-stm` and run `sudo ./x.py install`. Afterwards to use it navigate to any Rust package (such as `transactional-memory/txcell/` and build it with `cargo +stage1 build` or run tests with `cargo +stage1 test`.

The `RUSTC_LOG` environment variable determines which modules emit logs at what levels (`warn`, `debug`, or `info`). For example, `RUSTC_LOG=rustc_mir::transform::transaction=debug cargo +stage1 test` compiles and runs the tests in the current crate. While compiling, it emits all logs (debug, info, and warn level) from the `rustc_mir::transform::transaction` package.

The `txcell` crate has the `transaction_level` flag set to 2 by default to use reader-writer locks. The flag value is set in `.cargo/config`.

```
[build]
rustflags = ["--emit", "mir", "-Z", "
   transaction_level=2"]
```

Listing 2. The `.cargo/config` file that sets the `transaction_level` flag for the compiler.

D. Performance Evaluation

Navigate to `Documents > performance > transactional-memory > txcell` and execute the script `run_experiment.sh` which will kick off experiments and generate plots. Notice the size of the tree is smaller for these experiments than in the paper, which can be changed by setting `NUM_NODES=` in `tests/rbtree.rs`. The number of cores is also smaller on the virtual machine and can be individually adjusted in the tests of in each file. This script may take a while to run depending on the platform and size of data structures. Plots can be found in `plots/img/` and match the paper figures as follows:

- `rbtree.rs` generates Figure 5: `plots/img/fig5*.png`
- `linear.rs` generates Figures 6-9: `plots/img/fig6789*.png`
- `mixed.rs` generates Figure 10: `plots/img/fig10.png`

Tests for figure 5: rbtree_mix.rs tests for figure 6,7,8,9: linear.rs build pflock: TXN=true cargo +stage1 build

VII. Content

Below we list the relevant files for each evaluation, including those referenced or responsible for any data or plots. Additional details can be found in the README corresponding to a given evaluation.

A. Schedulability Evaluation

This portion of the artifact is in the directory named `schedulability`:

- Guide: `README.TXT`
- Experiment setup: `exp/pedf_lp.py`
- Plots: `plots/`

This evaluation builds on two existing artifacts [5], [8]; it compares the schedulability of task sets under a retry-based, lock-free scheme [5] to those task sets instead under retry-free, lock-based scheme [8]. This comparison is set up in the file `exp/pedf_lp.py`, which uses the existing schedulability tests. Additional details on the development of these tests are available from the published artifacts, and we provide information on how to run our comparison in the remainder of this document.

B. Compiler

The compiler changes touch the Abstract Syntax Tree (AST), High-level Intermediate Representation (HIR), High-level Abstract Intermediate Representation (HAIR), Mid-level Intermediate Representation (MIR), configuration, and lang items.

Abstract Syntax Tree (AST). AST code is contained in the `libsyntax` and `libsyntax_pos` packages. `src/libsyntax/ast.rs` adds `TransactionBlock`, `Lock`, and `Unlock` types to the `ExprKind` enum.

`src/libsyntax_pos/symbol.rs` adds `kw::Transaction` to the list of symbols so that the string ”transaction” prefixes a transaction code block. `src/libsyntax/parse/parser/expr.rs` parses the block if it sees `kw::Transaction`.

The remaining files changed in the `libsyntax` package pass the lock, unlock, and transaction `ExprKind`s through to the HIR.

High-level Intermediate Representation (HIR). HIR code is mostly located in the `rustc::hir::lowering` package.

`src/librustc/hir/lowering.rs` contains the key code that ”lowers” an `ast::ExprKind::TransactionBlock` to `hir::ExprKind::Lock` and `hir::ExprKind::Unlock` calls surrounding a standard code block. `src/librustc/hir/lowering/expr.rs` also contains some code that handles transaction blocks the same as regular code blocks.

Other files changed in the `rustc::hir`, `rustc::middle`, `librustc_passes`, and `librustc_typeck` packages pass these new `hir::ExprKind::Lock` and `hir::ExprKind::Unlock` types around.

High-level Abstract Intermediate Representation (HAIR). HAIR is a small middle step between HIR and MIR. It's located in the `rustc_mir::hair` package. We look up the lang items VII-B to create lock and unlock calls that link to our actual lock implementations in `src/librustc_mir/hair/cx/expr.rs` .

Mid-level Intermediate Representation (MIR). TORTIS's static analysis takes places in the MIR, mostly in the `rustc_mir::transform::transaction` package. `src/librustc/mir/mod.rs` contains new struct definitions so they can be imported across the compiler. Note that both `rustc/mir/` and `rustc_mir/` directories exist in this version of the compiler because the compiler team is in the process of making it more modular.

`src/librustc/query/mod.rs` contains a list of query macros. The Rustc Dev Guide [1] explains queries in Chapter 20, Overview of the Compiler. TORTIS adds two new queries, `get_shared_objects` and `conflict_analysis`. `get_shared_objects` gets the shared objects for a given `DefId` (crate index and def index). `conflict_analysis` performs conflict analysis on an entire crate using the shared objects and returns the conflict sets.

We had to add `#[derive(Eq, PartialEq, Hash)]` to some structs in `src/librustc/mir/interpret/error.rs` so that they can be used in the query system. The query system needs these properties for memoization. See Rust by Example chapter 16 on Traits for an explanation of `derive` [3].

The following files in `src/librustc_mir/transform/` include key static analysis:

- `mod.rs` implements the query functions and applies the patches that insert lock calls. It uses the `rustc_mir::transform::transaction` modules.
- `transaction/mod.rs` contains some other helper functions for patching the lock calls.
- `transaction/transaction_map.rs` maps function calls to the transactions in which they are contained.
- `transaction/use_def_analysis.rs` is the main def-use analysis. It imports `TransactionMap`.
- `transaction/conflict_analysis.rs` performs conflict analysis.

Lang items. Lang items are functions from external libraries that the compiler can reference and use. We implement our lock using new lang items in `src/librustc/middle/lang_items.rs` called `transaction_read_lock`, `transaction_read_unlock`, `transaction_write_lock`, and `transaction_write_unlock`. The Rust Unstable book has more details on lang items in chapter 2.98 [4].

Compiler config. `src/librustc/session/config.rs` contains settings for compiler flags. We create the `transaction_level` compiler flag to determine the level of transaction optimization. Level 0 does not create transactions, level 1 uses a ticket lock, and level 2 uses our reader-writer lock. The compiler flag is checked at the MIR stage.

C. *Performance Evaluation*

This portion of the artifact is in multiple directories. The following files are in the directory named `transactional-memory/`:

- Guide: `txcell/README.md`
- Experiments: `txcell/tests/`
  - Figure 5: `rbtree.rs`
  - Figure 6-9: `linear.rs`
  - Figure 10: `mixed.rs`
- Plotting: `txcell/plots/`
  - Figure 5: `plot-rbtree.rs`
  - Figure 6-9: `plot-linear.rs`
  - Figure 10: `plot-queue.rs`

Our compiler implementation uses a read-write lock from a different *crate* (project directory in Rust) called `PFLock` with the following notable files:

- Brandenburg's implementation of a phase-fair reader/writer lock: `pflock_c/pft.h` [6]
- Modified implementation of a phase-fair reader/writer lock with cache-line aligned lock attributed: `pflock_c/pftc.h`

The `txcell` crate imports `pflock` as a local crate in its `Cargo.toml` file. The locks are initialized and used in lock and unlock calls in `txcell/src/lib.rs`.

We modified an open source library *swym* [7]. We modified the following files by adding a sort function after calls to function `epoch_locks`:

- `src/internal/commit.rs`
- `src/internal/read_log.rs`
- `src/internal/write_log.rs`

VIII. Licenses

The Rust compiler is dually licensed under both the Apache 2.0 license, and the MIT License. All TORTIS-specific code is solely distributed under the terms of the MIT License. All other source code is distributed under the MIT License.

References

[1] "Guide to rustc development," https://rustc-dev-guide.rust-lang.org/.
[2] "LITMUS^{RT} home page," http://www.litmus-rt.org/.
[3] "Rust by example," https://doc.rust-lang.org/rust-by-example/index.html.
[4] "The rust unstable book," https://doc.rust-lang.org/unstable-book/the-unstable-book.html.
[5] A. Biondi and B. Brandenburg, "Artifact for Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors," http://people.mpi-sws.org/~bbb/papers/ae/ecrts16/pedf-synch.html, 2016.
[6] B. B. Brandenburg and J. Anderson, "Reader-writer synchronization for shared-memory multiprocessor real-time systems," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.
[7] T. Kopf, "swym," https://github.com/mtak-/swym, 2019, commit f7b635d.
[8] C. Nemitz, S. Caspin, J. Anderson, and B. Ward, "Light reading: Optimizing reader/writer locking for read-dominant real-time workloads (artifact)." Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.