STAMP-Rust: Language and Performance Comparison to C on Transactional Benchmarks*

Felix Suchert[0000−0001−7011−9945] and Jeronimo Castrillon[0000−0002−5007−445X]

TU Dresden, Germany
{felix.suchert,jeronimo.castrillon}@tu-dresden.de

Abstract. Software Transactional Memory has been used as a synchronization mechanism that is easier to use and compose than locking ones. The mechanisms continued relevance in research and application design motivates considerations regarding safer implementations than existing C libraries. In this paper, we study the impact of the Rust programming language on STM performance and code quality. To facilitate the comparison, we manually translated the STAMP benchmark suite to Rust and also generated a version using a state-of-the-art C-to-Rust transpiler. We find that, while idiomatic implementations using safe Rust are generally slower than both C and transpiled code, they guarantee memory safety and improve code quality.

Keywords: Software Transactional Memory · Memory Safety · Parallelism.

1 Introduction

Software Transactional Memory (STM) is a well-established method for synchronizing access to shared state in parallel programs. With the advent of emerging technologies such as Non-Volatile Memory (NVM) transactional operations on memory have found new importance [13,5,30]. However, the original *tl2* framework [14] and many subsequent frameworks have been written in C. The language itself has proven to be notoriously unsafe, requiring manual memory management and regularly exposing pointers to developers.

Rust [4] is a system programming language that has been designed with memory safety as one of its main goals. Its main selling point is the strong type system based on *Ownership types* [8,9]. This ensures that any well-typed program will not exhibit unsound behavior such as dangling pointers or data races through aliased references. Prior work has shown that such a strict type system can substantially simplify the specification and verification of system software [2]. By now, Rust has become a well-established language that is used in the development of systems applications like browser engines [1] and operating systems [24,25]. Rust's versatility also makes it an appealing language for

* This project is partially funded by the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST).

GPU programming [20], writing HPC applications [10] and as source language for accelerator programming [32]. In the context of transactional memory, the recent `rust-stm` library [6] offers STM functionalities. In contrast to the rich body of work on STM using C, there is, however, a lack of studies and benchmarks that help understand the impact of the Rust programming model on STM performance.

In this paper, we analyze the Stanford Transactional Applications for Multi-Processing (STAMP) suite [11], a benchmark collection specifically tailored towards Transactional Memory frameworks. Using accepted and recommended programming practices, we re-implemented the STAMP applications, which were originally developed in C/C++. We provide details on the challenges brought by the ownership and borrowing semantics of Rust to ensure a safe re-implementation of the benchmarks. The effort invested in creating safe implementations of the applications allows us to gage the impact of the programming model on the execution performance. To compare against plain unsafe implementations, we use the c2rust transpiler [12] to automatically generate Rust implementations directly from the original C applications. We discovered that Rust's strict borrowing semantics require the code to strictly adhere to using transactions for all variable accesses. While this generally improved the code's safety, it significantly decreased performance compared to C and unsafe Rust implementations.

This paper makes the following contributions:

1. Building atop the STAMP benchmark suite (Section 2), we manually implement a Rust version for the benchmarks, STAMP-Rust (Section 3).
2. We provide a qualitative comparison between the manually translated code and the code generated by `c2rust` (Section 4).
3. A performance evaluation of both Rust versions against the original C implementation using the `rust-stm` framework (Section 5).

2 Background and Related Work

Transactional Memory (TM) [19] is a synchronization mechanism for parallel programming. The key idea of this concept is to encapsulate sections of code that should run in parallel while modifying a shared data structure in *transactions*. During execution, each transaction keeps a log of all modifications and accesses to shared data which are played back once the transaction completes. From the outside, transaction blocks then seem to execute atomically, as all changes related to a transaction are either committed at once, or not at all. The latter case can occur when another transaction running in parallel is committed beforehand and has changed a shared variable that is also read or modified by the current transaction. Such write conflicts are resolved by reexecuting the transaction until it is successfully committed. This approach to synchronization is often referred to as *Optimistic Parallelism* [22]. Several potentially conflicting operations are scheduled in parallel under the assumption that conflicts are rare enough that occasional repeated computations of single transactions will not impact overall performance. These applications form their own sub-genre

of parallelization problems and usually involve large data structures based on pointers.

STM has been established more than 20 years ago. Since then, different approaches have been taken to test the performance of the mechanism. However, STAMP [11] has since prevailed and is still used today, more than 10 years after its inception [34,5,27]. It improves over existing approaches like RSTMv3 [31] and STMBench7 [17] by providing a wider variety of applications and better portability. Other works have resorted using microbenchmarks [13,15], which are not suitable as real-world examples. Additionally, the YCSB benchmark suite [30] has been used as benchmark to test key-value stores; however, a similar database-like application, i.e., vacation, is part of STAMP as well.

The rest of this section presents the STAMP benchmark suite and describes the `rust-stm` framework used to implement the benchmarks in Rust.

2.1 The STAMP Benchmark Suite

STAMP [11] is a benchmarking suite specifically tailored towards the needs of TM applications. It consists of 8 real-world applications. The suite tries to cover a wide spectrum of properties, such as varying transaction lengths, contention and time spent in transactions. Additionally, all applications are taken from different application domains, such as engineering, machine learning and scientific computation. For our comparison, we discuss the following benchmarks:

Labyrinth implements a path-finding algorithm in a three-dimensional maze, a variation of Lee's algorithm [23]. A set of paths is to be mapped in that data structure based on a set of points provided as inputs. Paths between those points are found using a breadth-first search. With STM, this is implemented by guarding manipulations on the shared grid structure using a transaction. Hence, when a conflicting path mapping occurs, the faster of both transactions may commit while the second has to attempt to find another path.

Genome implements a whole-genome shotgun sequencing algorithm [29]. The goal is the sequencing of a complete genome from a set of nucleotide sequences provided as input. The algorithm first deduplicates the set of DNA segments provided as input and then uses overlap matching with a decreasing overlap size to stitch the genome sequence back together. On an abstract level, the core workload of this benchmark is the construction of an acyclic graph from a set of nodes by finding neighboring elements. Transactions are utilized here to guard the forward and backward links of individual nodes (i.e., nucleotide sequences) in that graph.

K-Means is a popular algorithm for cluster analysis in data mining and for data classification. It partitions a set of n observations into k clusters [26]. A list of observations and the desired number of clusters to sort the data into

are provided as inputs. The algorithm then iteratively assigns a cluster to each observation and recomputes the cluster center from all assigned points. This is repeated until a convergence threshold is passed. Transactions are used here to guard access to the individual centroids, which are accessed as part of the processing of each observation.

Intruder implements a signature-based network intrusion detection system. It detects malicious activities and policy violations by inspecting live network traffic. The application implemented as part of STAMP is based on design proposal number five of Haagdorens et al. [18]. Due to the architecture of today's networks, namely the maximum size of network packets, individual network flows sometimes are split into multiple packets which may be transmitted and received in any order. Hence, all incoming network packets are captured and reassembled in parallel using a shared hashmap guarded by a transaction. The reassembled flows are then processed by the signature detection pass, in which a simple pattern matching is performed on the input.

Ssca2 implements kernel 1 from the Scalable Synthetic Compact Applications 2 [3]. It constructs a directed weighted multi-graph in parallel using adjacency and auxiliary arrays. Nodes are added in parallel to the graph, whereby the adjacency arrays are guarded by transactions to ensure safe parallel accesses.

Yada is *Y*et *a*nother *D*elaunay *a*pplication and implements a Delaunay mesh refinement. The algorithm modifies a mesh of triangles such that all interior angles of the triangles are larger than a certain threshold. If a triangle violates this criterion, it is merged with surrounding triangles and split into a set of new triangles. These operations are performed in parallel and the replacement of the formed cavity with new triangles is guarded by transactions.

2.2 Software Transactional Memory in Rust

The Rust implementation of STM used for our comparison is Rust-STM [6]. It abstracts over the transactional synchronization aspects by providing a dedicated type for transactional variables, `TVar`. The type encapsulates the variable to be

```
let val = TVar::new(42);
atomically(|trans| {
    let mut x = val.read(trans)?;
    x /= 2;
    val.write(trans, x)?;
    Ok(())
})
```

Listing 1: Working with Transactions in Rust-STM.

protected and provides an interface to modify it during a transaction. Transactions itself are implemented as functions that accept as argument a closure[1] that forms the transaction. Listing 1 shows this function on line 2. Within that closure, protected variabled may be accessed using a special transaction context variable. All these access functions return a type that indicates whether the operation is found to be in collision with another already-committed transaction. The `?` operator will enforce a retry on the transaction upon failure.

3 STAMP on Safe Rust

To facilitate a comparison between STM applications in Rust and C, we manually implemented the whole STAMP benchmark suite[2] in Rust and have published it under the name STAMP-Rust[3]. During translation, we followed the recommended coding practices put forth by Blandy et al. [7]. This section discusses how using Rust as implementation language impacts program performance and safety.

3.1 Type-Level Safety

The C-based `tl2` library provides opt-in transaction semantics that can easily be violated. Users are cautioned to not access shared data structures outside of transactions as it can easily lead to data races. In Rust-STM, however, this danger is alleviated by the type system. Sharing data between threads in Rust is guarded by its *trait system*, which behaves similarly to interfaces in other languages. A particular type `T` may only be safely shared between threads when it implements the `Sync` trait. This property holds if and only if a read-only reference `&T` of that type can be sent between threads safely. In other words, there must not be any possibility for undefined behavior (which includes data races) to occur if a reference to some data is shared among threads. Therefore, data shared among threads may not be mutated as no mutable references can be derived safely from an immutable one. A known workaround is to define a type that implements so-called *interior mutability*. These types can safely mutate their interior data even through a shared reference. A number of types in Rust's Standard Library implement this behavior and have been proven to be safe [21].

Rust-STM encapsulates transaction variables in a dedicated structure constructed from such types with interior mutability. Since the wrapped data is not exposed, accessing it is only possible through methods implemented on the container type. However, these methods require to be executed as part of a transaction. It is, therefore, impossible to circumvent Rust's safeguards regarding data sharing or to violate transaction semantics.

[1] Closures in Rust are comparable to Lambda functions in other languages. They can have arguments and capture variables from the outside context. The implications of the latter are not relevant for this paper.

[2] The code base for the original STAMP applications can be found on https://github.com/robert-schmidtke/stm.

[3] https://github.com/tud-ccc/stamp-rust

3.2 Composable Transactions

The strong typing of transaction variables further leads to better composability of transactions. In the `tl2` implementation, it is not transparently visible whether a function needs to be executed in a transaction context or spawns one itself. This can lead to problems when accidentally calling functions that expect to be run in a transaction context or calling a function that creates a transaction from an already-running transaction

Rust-STM addresses this problem in part by requiring a `&mut Transaction` type for all its non-atomic transaction variable modifications. Therefore, functions expecting to be run from a transaction context must accept such a type as function argument, clearly indicating the required context. Nesting transaction blocks, however, cannot be detected by the type system and, hence, will only result in a runtime error.

3.3 The Overhead of Safety

Although Rust's Ownership type system enforces transactional safety throughout the program, it can also lead to computational overhead compared to `tl2`. Every time a transaction variable is read, the reader receives a full copy of the underlying data structure. This is necessary as the variable must retain ownership of the data in case another transaction commits a change in the meantime. Depending on the size of the data structure, this copying gives transactions a substantial memory footprint besides internal data structures like logs. To bypass this copying, transactions can also receive an immutable pointer to the current value of the transactions internal data[4]. However, this is only feasible when the data will not be modified by a transaction.

The performance implication of this type safety becomes apparent in the *Labyrinth* application. Here, when mapping a path through the maze, many fields of the maze are read to determine the shortest path. A naïve implementation would read all fields of the maze on the go. But this inevitably leads to duplicated reads and a generally higher probability for the transaction to fail. Its read set gets blown up by the many read accesses to fields that are not even part of the final mapped path in the end. This is circumvented in both the C and the Rust implementation by creating a local copy of the maze before attempting a mapping. In Rust, we opted to use `TVar`s `read_atomic` function to create the copy, as shown in Listing 2. This incurs a high overhead, since each individual cell in the maze is copied individually, but the `TVar` type does not offer more efficient methods to access its contents.

In C, however, there are no safeguards regarding accesses to transaction variables as there is no strict notion of such a type. Instead, the labyrinth implementation of the original STAMP suite resorts to copying the data structure

[4] This internally uses an atomically reference-counted pointer. When a new value is written by another transaction, the `TVar`s internal pointer is replaced, not changing the contents of the shared pointer.

```rust
type StmGrid = Vec<Vec<Vec<TVar<Field>>>>;
type Grid = Vec<Vec<Vec<Field>>>;

fn create_working_copy(grid: &StmGrid) -> Grid {
  grid.iter()
    .map(|y_grid| {
      y_grid
        .iter()
        .map(|z_grid| z_grid.iter().map(|pt| pt.read_atomic()).collect())
        .collect()
    })
    .collect()
}
```

Listing 2: Creating a local copy in Rust incurs a high overhead due to the cumbersome data accesses.

as a whole using a single invocation of `memcpy`. This potentially brings increased performance but violates the STM concurrency model. It has been shown that such behavior increases the potential for deadlocks and memory races and is often done to circumvent limitations of the concurrency model used [33].

In the *ssca2* benchmark, we run into a similar problem: The C version utilizes thread barriers to synchronize individual threads and switch between data-parallel and transaction contexts. During transaction contexts, the adjacency and auxiliary arrays are accessed and updated as part of a transaction. Outside of that context, both arrays are frequently read by all threads to continue their computations without any memory overhead. In Rust, we can only implement a similar behavior by joining running threads periodically to update data structures before spawning new threads. This, of course, incurs additional overhead but does not violate the transactional model.

3.4 The Complexity of Using Associative Arrays

Using more complex data structures from external libraries as part of a transaction may quickly lead to a performance bottleneck in Rust due to data copying. As a rule of thumb, transactional variables should always encapsulate as few data as necessary to keep the memory footprint low. However, even using associative arrays from the standard library, such as Hashsets, then poses a challenge, as they offer no access to their intrinsics. As a result, these data types cannot be accessed unsafely due to Rust's type system, but also not efficiently out of the box.

The C implementation circumvented that problem. Since Cs standard library does not include such data types anyway, the authors of STAMP opted to write their own transaction-aware associative arrays. A similar solution could be implemented for Rust in the future, based on a suggested efficient algorithm by Paznikov et al. [28].

4 Analysis of Automatically Generated Benchmarks

Since the advent of Rust and similar memory-safe languages, the question has been raised whether or not its promises of safety could be leveraged automatically for larger code bases written in C. As a result, transpilers have been implemented that can translate C to Rust code. As part of our analysis of the STAMP benchmark suite, we used `c2rust` [12] to automatically generate Rust code for our selected benchmarks. In this section, we discuss the quality of the generated code and why automatic transpiling can as of now not serve as a replacement for manually translated Rust code.

The `c2rust` transpiler is built atop the `clang` compiler frontend and is designed to process individual files adhering to the C99 standard. `clang` emits the Abstract Syntax Tree (AST) of the input file, which is then transpiled and emitted in the form of Rust code.

```
1  pub unsafe extern "C" fn router_solve(mut argPtr: *mut libc::c_void) {
2      let mut routerArgPtr: *mut router_solve_arg_t =
3          argPtr as *mut router_solve_arg_t;
4      let mut routerPtr: *mut router_t = (*routerArgPtr).routerPtr;
5      let mut mazePtr: *mut maze_t = (*routerArgPtr).mazePtr;
6      let mut myPathVectorPtr: *mut vector_t = Pvector_alloc(
7          1 as libc::c_int as libc::c_long,
8      );
9      // ...
10 }
```

Listing 3: Beginning of the path-finding function from the *Labyrinth* application, generated by an automatic transpiler.

Unfortunately, this literal translation of programs results in code that still is more similar to C semantics than idiomatic Rust code. Listing 3 shows the code for the entry point of the path-finding function in the *Labyrinth* application. It has been declared as unsafe, as it internally mainly relies on the use of what Rust calls *"raw pointers"*, pointers not guarded by the languages safety guarantees. The existence of such pointers itself does not violate these guarantees; however, dereferencing them does, which happens in lines 4 and 5. Also, the generated code contains frequent uses of type casting, which is also unsafe. Previous work has found that the inability to generate safe code is one of the key drawbacks of these automatic approaches [16]. Additionally, automatically generated Rust code makes no use of the more sophisticated features of the Rust language such as `struct` member functions. This would require a deep understanding of the code structure and meaning on part of the transpiler that is hard to achieve.

Since the source code for this transpilation used to be C code, which uses manual memory management via `malloc` and `free`, this concept also surfaces in Rust. This is especially problematic since most generated code operates outside

of Rusts safety boundaries. As a result, it cannot be ruled out that double-frees and other undefined behavior occur in the Rust code if the C sources already contained such bugs.

As the transpilation happens file by file, all generated Rust files expose their data types and functions using the `extern "C"` calling convention. This is also shown in Listing 3 in line 1. This not only creates significant bloat in the code, but it also means that all generated Rust functions communicate with one another through C standard calling conventions.

As `tl2` is used as an external library in the STAMP suite, our transpiled STM code still relies on this library. We are thereby be forced to adhere to the C framework's general architecture or would need to restructure the code base significantly to use Rust-STM.

All in all, the code generated by automatic tooling is in this case inferior to a manual sound translation. The generated code needs extensive refactoring to remove all occurrences of unsafe code. We, therefore, deemed a manual rewrite as preferrable in our work as it allowed us to construct the applications from the bottom up in an idiomatic way. The generated code is still useful to gage the cost of switching to the Rust programming language.

5 Evaluation

To evaluate the performance of the different implementations of the STAMP suite, we execute all benchmarks with varying configurations. We then compare the resulting runtimes and speedups and classify both Rust approaches in terms of their code quality.

5.1 Methodology

To run the benchmarks, we use input data sets originally put forward by Minh et al. in their original work [11]. The paper describes three different input sizes for each benchmark: 'small', 'medium' and 'large'. Since the creation of the benchmark suite, numerous advancements in hardware have significantly increased processor speeds. Hence, for most applications, the 'small' and 'medium' sized inputs are ill-suited for a comparison. Most of these inputs are so small that the applications terminate after significantly less than 100 milliseconds. In such a small range, the measuring noise introduced by the operating system dominates the results. We thus use the 'large' or '++' data sets from the benchmark suite for our measurements of k-means, labyrinth, ssca2 and genome. For intruder and yada, we use the medium-sized input data set, as execution times of the STM implementations were extremely long for the large input set. The k-means application additionally provides a low-contention and high-contention input set differing in the number of clusters to be computed.

Additionally, we made some changes to the k-means code. Originally, the benchmark terminated either upon convergence or after 500 iterations. However,

due to variations in the floating point accuracy of the C and Rust implementations, both versions converge only after a wildly varying number of iterations. For the large input data set, convergence was always reached at the latest between 150 and 200 iterations. For a meaningful comparison between the Rust and C implementations, we hardcode the termination after 200 iterations.

As we pointed out in Section 4, automatically translated benchmarks also use the `tl2` library through a C interface. As a consequence, optimized non-debug builds fail execution due to memory faults, which are probably caused by API instabilities. For that reason, we conduct the rust-tl2 measurements using debug builds only. Resulting speedups are still valid, as the baseline is also measured from a debug build. For the runtime comparison, however, we exclude the rust-tl2 measurement results to not distort the plot. Instead, we only show the sequential execution time of the rust-tl2 applications to compare general language overheads in runtimes.

We run all measurements on a workstation with an Intel Core i9-10900K CPU, 32 GiB DDR4-2933 RAM and Ubuntu 22.04 LTS installed. All measurements are repeated 30 times to minimize the effect of random jitter caused by system processes.

To measure the speedup of STM applications, we run all measurements for 1, 2, 4, 8 and 16 threads. This limitation stems from the original C-based STAMP implementation requiring the thread count to be a power of 2.

5.2 Performance Comparison

Figure 1 shows the mean speedups achieved by all three implementations compared to their respective sequential baselines. Additionally, Figure 2 shows the mean execution times of all configurations. We observe that the manually implemented Rust version generally performs worse than the C implementation when transactions are used more frequently.

For Labyrinth, the manually implemented Rust version only manages to achieve half the speedup of the C implementation. This is mainly caused by the high overheads induced by repeatedly cloning the grid data structure, as outlined in Section 3.3. Almost 50 % of the total time spent inside transactions is used for creating local copies of the maze. The C version circumvents that by unsafely copying the memory of the grid to a new location, undetected by any transaction. Rust-tl2 and c-tl2 are on par in terms of speedup. However, Figure 2 reveals that the Rust version executes almost 50 % faster than the C implementation. This hints at Rust in this case being generally more efficient, which can also be seen in the other benchmarks.

In Genome, we observe a similar pattern of Rust-STM underperforming in comparison to the C version. While the speedup increase is generally there, it is offset by a factor two from the other implementations. This can directly be attributed to Genome's internal use of complex associative arrays, namely HashMaps and HashSets. As pointed out in Section 3.4, the C version (and therefore the transpiled Rust version, too) implements its own transaction-ready hash-based data structures. The Rust-STM library does not come with such

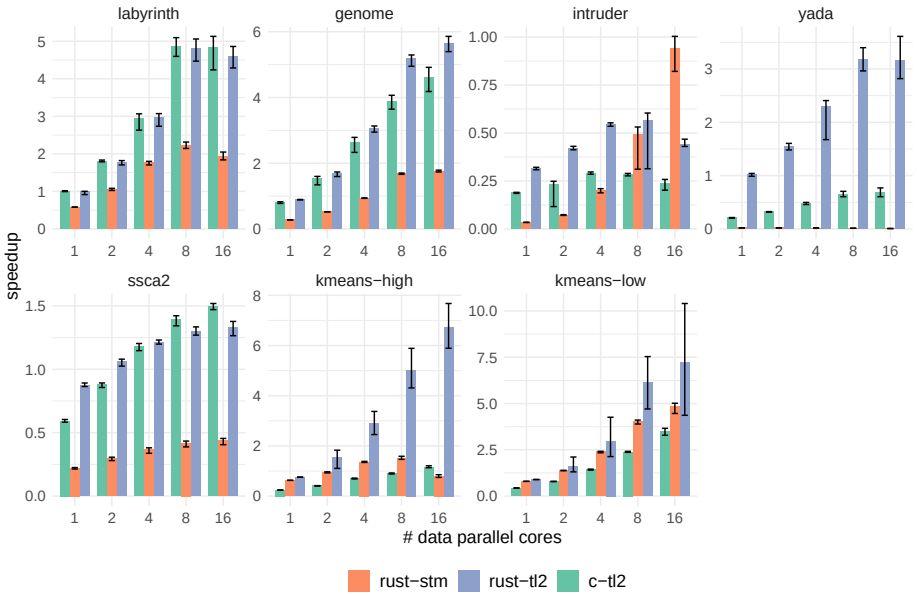

Fig. 1. Speedups of different benchmark implementations over their respective sequential implementations for a varying number of threads.

data structures and therefore has to resort to constructing a HashSet alternative using Standard Library methods. We implemented a transaction-aware HashSet and HashMap that internally uses a fixed number of buckets, each containing a HashSet or HashMap protected by a transaction variable. The performance then decreases because this is significantly less efficient than constructing such a type from scratch. Future work should re-evaluate these benchmarks with a data structure leveraging more efficient implementation approaches [28].

The Intruder benchmark reveals the same performance issue in Rust-STM. However, the Rust-STM implementation manages to overtake all other implementations when using more threads. This indicates that the transaction overhead for HashSets can indeed be offset in some cases by the use of more threads, although speedups still do not exceed 1.0 for this benchmark.

Yada's Rust-STM implementation indeed performs significantly worse than the two competing versions. Here, HashMaps and HashSets are used very prominently to store the mesh's triangles and depict neighborhood relations between different elements. Thus, every modification of the graph requires the copying and writing back of one or multiple buckets of our self-implemented Hash data structure. While this is slightly more efficient than having a single HashMap that is modified every transaction, it still incurs a huge overhead compared to specialized data structures.

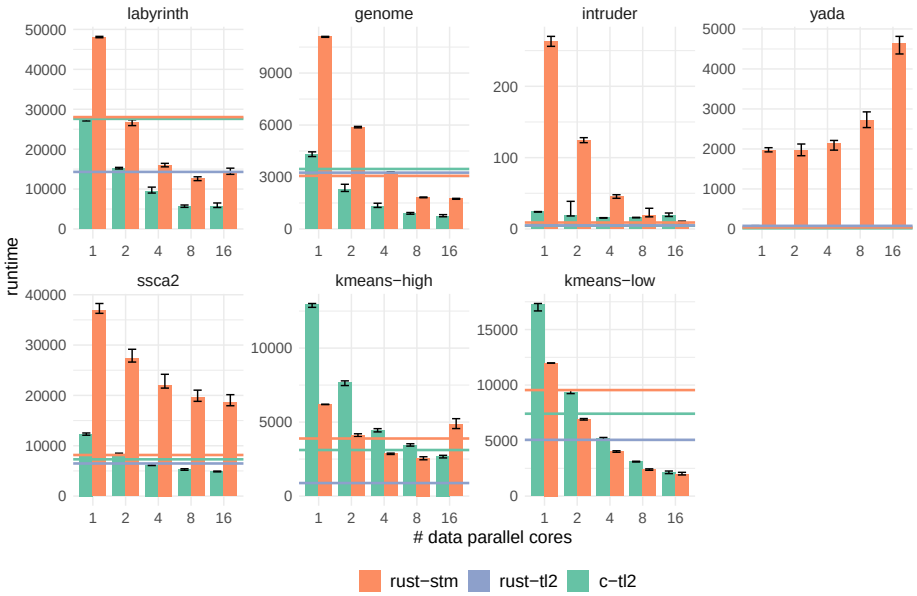

Fig. 2. Runtimes of the different benchmark implementations and their respective sequential implementations. The horizontal lines indicate the sequential execution time for comparison. Runtime data for parallel rust-tl2 executions has been omitted as no data could be obtained for non-debug builds.

In ssca2, the performance difference is rooted in more fundamental differences between C an Rust, however. As outlined in Section 3.3, this benchmark has various synchronization points at which execution switches between transactional and data-parallel computing. Furthermore, data structures are frequently written to in parallel without synchronization, heavily imparting any safety guarantees. In Rust, neither of both is safely doable as the type system strictly prohibits both unguarded accesses to transactional variables and shared mutability. Consequently, the Rust-STM implementation has to terminate threaded execution to synchronize after data-parallel sections. This added computational overhead is clearly visible both in the speedup and even more clearly in the running time of the benchmark. Rust-tl2 and c-tl2 however, are both on par in terms of speedup.

For k-means, both Rust versions outperform the C implementation. While the Rust-STM version executes generally slower than the C version, the Rust-tl2 implementation terminates significantly faster. Therefore, the speedup can be attributed to the Rust runtime's more efficient handling of contention and more radical vectorization of numerical computations.

5.3 Qualitative Analysis

Despite the performance problems Rust-STM shows in some of the benchmarks, it improves significantly on the safety of the applications. On the other hand, the C implementation, which in and of itself already lacks any safety guarantees, has chosen to trade further safety aspects by violating the STM concurrency model. A small percentage of benchmark runs for the C-version STAMP applications were aborted due to faulty memory management. Rusts cleaner approach to memory management rules out such behavior.

While the automatically-transpiled code generally performed better than the native C implementation, it combines several negative aspects in terms of code quality. The generated code itself is non-idiomatic as discussed in Section 4, while it still contains all the possibly undefined behavior of unsafe Rust code. Hence, understanding the source code is challenging, which makes it even harder to spot potential bugs. The manual implementation, on the other hand, leverages Rust's type system fully and provides better readability and maintainability: Transactions are clearly encapsulated into `atomically` blocks; and functions requiring to be run inside a transaction context are marked as such by their signature.

6 Conclusion

Motivated by STMs continued presence in both research and application development, we analyze how the performance of the mechanism is impacted by using the type-safe Rust language for implementation. We implement the STAMP benchmark suite in Rust (STAMP-Rust) and find that the existing C implementation regularly performs unsafe memory operations and violates the STM concurrency model. Hence, Rust-STM implementations of STAMP benchmarks are up to 50 % slower than their C implementations in transaction-intensive benchmarks. On the other hand, automatically generated, unsafe Rust code regularly outperforms the C equivalents, hinting at Rust being generally more efficient. We think that our safe re-implementations of the prominent STAMP suite along with the presentation of our design rationale can serve as baseline for further research on STM applications using the Rust programming model. We leave it as future work to explore more efficient implementations leveraging fine-granular transactions on complex data structures, such as genome and intruder.

Acknowledgements

The authors would like to thank Sebastian Ertel for his valuable input.

References

1. Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J., Sapin, S.: Engineering the servo web browser engine using Rust. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp.

81–89. ACM, Austin Texas (May 2016). https://doi.org/10.1145/2889160.2889229, https://dl.acm.org/doi/10.1145/2889160.2889229
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (Oct 2019). https://doi.org/10.1145/3360573, https://dl.acm.org/doi/10.1145/3360573
3. Bader, D.A., Madduri, K.: Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) High Performance Computing – HiPC 2005, vol. 3769, pp. 465–476. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11602569_48, http://link.springer.com/10.1007/11602569_48, series Title: Lecture Notes in Computer Science
4. Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamarić, Z., Ryzhyk, L.: System Programming in Rust: Beyond Safety. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems. pp. 156–161. ACM, Whistler BC Canada (May 2017). https://doi.org/10.1145/3102980.3103006, https://dl.acm.org/doi/10.1145/3102980.3103006
5. Beadle, H.A., Cai, W., Wen, H., Scott, M.L.: Nonblocking Persistent Software Transactional Memory. In: 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC). pp. 283–293. IEEE, Pune, India (Dec 2020). https://doi.org/10.1109/HiPC50609.2020.00042, https://ieeexplore.ieee.org/document/9406709/
6. Bergmann, G.: Software Transactional Memory (Aug 2022), https://github.com/Marthog/rust-stm, original-date: 2015-09-15T14:45:14Z
7. Blandy, J., Orendorff, J.: Programming Rust: fast, safe systems development. O'Reilly Media, Sebastopol, California, first edition edn. (2017), oCLC: on1019128949
8. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '02. p. 211. ACM Press, Seattle, Washington, USA (2002). https://doi.org/10.1145/582419.582440, http://portal.acm.org/citation.cfm?doid=582419.582440
9. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time Java. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation - PLDI '03. p. 324. ACM Press, San Diego, California, USA (2003). https://doi.org/10.1145/781131.781168, http://portal.acm.org/citation.cfm?doid=781131.781168
10. Bychkov, A., Nikolskiy, V.: Rust Language for Supercomputing Applications. In: Voevodin, V., Sobolev, S. (eds.) Supercomputing, vol. 1510, pp. 391–403. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-92864-3_30, https://link.springer.com/10.1007/978-3-030-92864-3_30, series Title: Communications in Computer and Information Science
11. Chi Cao Minh, JaeWoong Chung, Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: 2008 IEEE International Symposium on Workload Characterization. pp. 35–46. IEEE, Seattle, WA, USA

(Oct 2008). https://doi.org/10.1109/IISWC.2008.4636089, http://ieeexplore.ieee.org/document/4636089/
12. Contributors, C.: C2Rust (Aug 2022), https://github.com/immunant/c2rust, original-date: 2018-04-20T00:05:50Z
13. Correia, A., Felber, P., Ramalhete, P.: Romulus: Efficient Algorithms for Persistent Transactional Memory. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures. pp. 271–282. ACM, Vienna Austria (Jul 2018). https://doi.org/10.1145/3210377.3210392, https://dl.acm.org/doi/10.1145/3210377.3210392
14. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Dolev, S. (eds.) Distributed Computing, vol. 4167, pp. 194–208. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11864219_14, http://link.springer.com/10.1007/11864219_14, series Title: Lecture Notes in Computer Science
15. Dragojević, A., Harris, T.: STM in the small: trading generality for performance in software transactional memory. In: Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12. p. 1. ACM Press, Bern, Switzerland (2012). https://doi.org/10.1145/2168836.2168838, http://dl.acm.org/citation.cfm?doid=2168836.2168838
16. Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.: Translating C to safer Rust. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–29 (Oct 2021). https://doi.org/10.1145/3485498, https://dl.acm.org/doi/10.1145/3485498
17. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: A Benchmark for Software Transactional Memory (2006), http://infoscience.epfl.ch/record/89706
18. Haagdorens, B., Vermeiren, T., Goossens, M.: Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Lim, C.H., Yung, M. (eds.) Information Security Applications, vol. 3325, pp. 188–203. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31815-6_16, http://link.springer.com/10.1007/978-3-540-31815-6_16, series Title: Lecture Notes in Computer Science
19. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th annual international symposium on Computer architecture - ISCA '93. pp. 289–300. ACM Press, San Diego, California, United States (1993). https://doi.org/10.1145/165123.165164, http://portal.acm.org/citation.cfm?doid=165123.165164
20. Holk, E., Pathirage, M., Chauhan, A., Lumsdaine, A., Matsakis, N.D.: GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language. In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. pp. 315–324. IEEE, Cambridge, MA, USA (May 2013). https://doi.org/10.1109/IPDPSW.2013.173, http://ieeexplore.ieee.org/document/6650903/
21. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–34 (Jan 2018). https://doi.org/10.1145/3158154, https://dl.acm.org/doi/10.1145/3158154

22. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07. p. 211. ACM Press, San Diego, California, USA (2007). https://doi.org/10.1145/1250734.1250759, http://portal.acm.org/citation.cfm?doid=1250734.1250759
23. Lee, C.Y.: An Algorithm for Path Connections and Its Applications. IEEE Transactions on Electronic Computers **EC-10**(3), 346–365 (Sep 1961). https://doi.org/10.1109/TEC.1961.5219222, http://ieeexplore.ieee.org/document/5219222/
24. Levy, A., Andersen, M.P., Campbell, B., Culler, D., Dutta, P., Ghena, B., Levis, P., Pannuto, P.: Ownership is theft: experiences building an embedded OS in rust. In: Proceedings of the 8th Workshop on Programming Languages and Operating Systems. pp. 21–26. ACM, Monterey California (Oct 2015). https://doi.org/10.1145/2818302.2818306, https://dl.acm.org/doi/10.1145/2818302.2818306
25. Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., Levis, P.: The Case for Writing a Kernel in Rust. In: Proceedings of the 8th Asia-Pacific Workshop on Systems. pp. 1–7. ACM, Mumbai India (Sep 2017). https://doi.org/10.1145/3124680.3124717, https://dl.acm.org/doi/10.1145/3124680.3124717
26. Macqueen, J.: Some methods for classification and analysis of multivariate observations. In: In 5-th Berkeley Symposium on Mathematical Statistics and Probability. pp. 281–297 (1967)
27. Pasqualin, D.P., Diener, M., Du Bois, A.R., Pilla, M.L.: Online Sharing-Aware Thread Mapping in Software Transactional Memory. In: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 35–42. IEEE, Porto, Portugal (Sep 2020). https://doi.org/10.1109/SBAC-PAD49847.2020.00016, https://ieeexplore.ieee.org/document/9235046/
28. Paznikov, A., Smirnov, V., Omelnichenko, A.: Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory. In: 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon). pp. 1–5. IEEE, Vladivostok, Russia (Oct 2019). https://doi.org/10.1109/FarEastCon.2019.8934131, https://ieeexplore.ieee.org/document/8934131/
29. Pop, M., Salzberg, S., Shumway, M.: Genome sequence assembly: algorithms and issues. Computer **35**(7), 47–54 (Jul 2002). https://doi.org/10.1109/MC.2002.1016901, http://ieeexplore.ieee.org/document/1016901/
30. Ramalhete, P., Correia, A., Felber, P.: Efficient algorithms for persistent transactional memory. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–15. ACM, Virtual Event Republic of Korea (Feb 2021). https://doi.org/10.1145/3437801.3441586, https://dl.acm.org/doi/10.1145/3437801.3441586
31. Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Transactions and privatization in Delaunay triangulation. In: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing - PODC '07. p. 336. ACM Press, Portland, Oregon, USA (2007). https://doi.org/10.1145/1281100.1281160, http://dl.acm.org/citation.cfm?doid=1281100.1281160

32. Takano, K., Oda, T., Kohata, M.: Design of a DSL for Converting Rust Programming Language into RTL. In: Barolli, L., Okada, Y., Amato, F. (eds.) Advances in Internet, Data and Web Technologies, vol. 47, pp. 342–350. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-39746-3_36, http://link.springer.com/10.1007/978-3-030-39746-3_36, series Title: Lecture Notes on Data Engineering and Communications Technologies
33. Tasharofi, S., Dinges, P., Johnson, R.E.: Why Do Scala Developers Mix the Actor Model with other Concurrency Models? In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Castagna, G. (eds.) ECOOP 2013 – Object-Oriented Programming, vol. 7920, pp. 302–326. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_13, http://link.springer.com/10.1007/978-3-642-39038-8_13, series Title: Lecture Notes in Computer Science
34. Xu, Y., Izraelevitz, J., Swanson, S.: Clobber-NVM: log less, re-execute more. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 346–359. ACM, Virtual USA (Apr 2021). https://doi.org/10.1145/3445814.3446730, https://dl.acm.org/doi/10.1145/3445814.3446730