Precision Tuning the Rust Memory-Safe Programming Language

Gabriele Magnani
DEIB – Politecnico di Milano, Italy

Lev Denisov
DEIB – Politecnico di Milano, Italy

Daniele Cattaneo
DEIB – Politecnico di Milano, Italy

Giovanni Agosta
DEIB – Politecnico di Milano, Italy

Stefano Cherubin
NTNU – Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Precision tuning is an increasingly common approach for exploiting the tradeoff between energy efficiency or speedup, and accuracy. Its effectiveness is particularly strong whenever the maximum performance must be extracted from a computing system, such as embedded platforms. In these contexts, current engineering practice sees a dominance of memory-unsafe programming languages such as C and C++. However, the unsafe nature of these languages has come under great scrutiny as it leads to significant software vulnerabilities. Hence, safer programming languages which prevent memory-related bugs by design have been proposed as a replacement. Amongst these safer programming languages, one of the most popular has been Rust. In this work we adapt a state-of-the-art precision tuning tool, TAFFO, to operate on Rust code. By porting the PolyBench/C benchmark suite to Rust, we show that the effectiveness of the precision tuning is not affected by the use of a safer programming language, and moreover the safety properties of the language can be successfully preserved. Specifically, using TAFFO and Rust we achieved up to a 15× speedup over the base Rust code, thanks to the use of precision tuning.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software safety; Software and its engineering → Software performance; Computer systems organization → Embedded software

Keywords and phrases Approximate Computing, Memory Safety, Precision Tuning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.4

Funding The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, `http://www.apropos.eu/`). Also the Italian MISE project F/340043/01-03/X59 "Piattaforma ed ecosistema cooperativo, C- ITS ETSI standard per la mobilità digitale integrata" within the framework FCS - Accordi Innovazione Settore Automotive (DM 31/12/2021 e DD 10/10/2022).

1 Introduction

Proper memory management is crucial to ensure program stability and prevent software vulnerabilities which can be exploited by cyber-criminals to cause unauthorised actions for their own benefit. In fact, memory-related weaknesses remain a major concern in software development, as highlighted in the *Top 25 Most Dangerous Software Weaknesses* report by Mitre. Out-of-bounds writes held the top spot for three consecutive years, while several other

15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024).
Editors: João Bispo, Sotirios Xydis, Serena Curzel, and Luís Miguel Sousa; Article No. 4; pp. 4:1–4:12
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

memory-related issues, such as *use-after-free* and *NULL* pointer dereference, appear on the list every year [6]. This is also stated by the United States National Security Agency (NSA), which in a recent report (at the time we are writing) advocates the use of memory-safe programming languages [10]. However, the implementation of preemptive measures to ensure memory safety in programming languages can have an impact on performance. Specifically, the use of garbage collection and array bounds checking, which are common in memory-safe languages, can result in significant performance overhead. As a result, while memory-safe languages have been available for a long time, the embedded world is still dominated by the use of C and C++, which give no guarantee on how the programmer handles the memory. Some memory-safety features have been added to these languages over the years, specifically in C++ [7], however they are opt-in and not universally adopted. The reason is that the ability of C and C++ to operate at lower (unsafe) abstraction levels is *desired* by programmers in order to avoid the inefficiencies generated by memory-safety features. Indeed, some of the first memory-safe languages such as Java and C# completely disallow the use of unsafe concepts such as pointers. As a result, more recent language designs provide *safe* and *unsafe* subsets that also allows their use in low-level programming tasks.

One language of this sort that has gained considerable traction in the industry [8] is Rust[1]. Its goal is to allow the adoption of these languages in security-sensitive applications such as operating system development and embedded systems. According to the 2023 Annual Stack Overflow Survey [11], Rust is positively ranked among people who want to learn a new programming language – 6th in the *desired* programming language category – but what makes it really attractive is the experience of using it. Rust placed first in the *admired* category with a score of 84% – twice as much higher than Java and C, and more than 20% better than C#. The *admired* category represents the proportion of users who have used the same technology in the past year and want to continue using it. As such, it is reasonable to expect that the popularity of this language will continue to grow in the coming years.

In general, software development for embedded systems often requires the use of specific optimization techniques to cope with reduced computational capabilities and memory sizes. Among these techniques is Precision Tuning [3], which trades off result accuracy for time and energy efficiency by reducing the bit width of an operation, or by switching from floating point to fixed point arithmetic. Precision Tuning is part of the larger family of Approximate Computing (AxC) [12]. Precision Tuning impacts the data width, and therefore all memory accesses. As such, it can help to offset some of the penalties introduced by the memory management integrated within modern programming environments like Rust. However, at the time of writing and to the best of the authors' knowledge, no Precision Tuning tool supports Rust. Among the most notable recent efforts in the field of precision tuning, we notice that TAFFO [1] is an automated Precision Tuning set of plugins that leverages the LLVM compiler framework [9]. As such, it provides a useful baseline that could be integrated with other compiler components based on LLVM, which is currently the industry standard for Rust compiler development. Indeed, *rustc* – the reference Rust compiler implementation – is based on this framework.

In this work, we aim at filling the gap with automated Precision Tuning support in memory-safe programming languages. To this end, we use Rust as a test-bed for a proof-of-concept implementation that employs TAFFO as the engine for performing the analyses and transformations required for this kind of approximate computing task. More specifically, we integrate the TAFFO framework with *rustc* by providing the appropriate components that

[1] `https://www.rust-lang.org`

allow Rust code to have the programmer-inserted metadata and annotations required by TAFFO to work. As a side-effect, this demonstrates the language-independence of TAFFO, as well as its ability to optimise the code without negatively affecting memory safety.

We evaluate the effectiveness of TAFFO applied to Rust benchmarks, when compared to its effectiveness on C versions. The results show that TAFFO provides similar performance improvements in both C and Rust, when the *rustc* compiler is able to optimize away the out-of-bound access guards. When this is not possible, the benefits of TAFFO increase.

1.1 Key contributions

The main contribution of this work is the introduction of a precision tuning framework in the context of a memory-safe programming language. This task is non-trivial due to Rust's inability to classify as memory-safe some of the features required to perform precision tuning of selected variables. We therefore show that those features, when used to convey the information to TAFFO, do not lead to actual unsafe actions, and that the TAFFO precision tuning process – after minor adjustments – does not harm the memory safety guarantees enforced by Rust.

As additional contribution, we provide a new Rust port of the PolyBench suite, that is closer in design to the original PolyBench/C, to allow a better comparison between the *rustc* and Clang compilers when employing a particular optimisation, in this case precision tuning.

1.2 Structure of the paper

The rest of this paper is organised as follows. In Section 2, we cover the background on TAFFO and Rust. In Section 3, we describe the proposed approach for precision tuning a Rust program with TAFFO. In Section 4 we discuss the effectiveness of the integration, first by describing the experimental setup, and then by providing an assessment of the integrated system. Finally, in Section 5, we draw some conclusions and outline future directions.

2 Background

TAFFO and Rust are two different but complementary technologies. While Rust is a modern programming language, TAFFO is a set of compiler passes for precision tuning. In this section we discuss the peculiarities of these two tools in order to provide a background for the later discussion of their integration.

2.1 Memory-Safety Properties of Rust

To achieve its memory-safety properties, the Rust language enforces strict rules to prevent common programming errors such as *NULL* pointer dereferences and buffer overflows. In particular, the *rustc* compiler employs a dedicated component, the *borrow checker*, to enforce these rules. Among others, the borrow checker ensures that variables are initialised before they are used. It ensures that values, whether they are held in variables or temporaries, are not moved twice, or while borrowed. Moreover, it ensures no variable is accessed while mutably borrowed (except through the reference), and no variable is mutated while immutably borrowed. In general, Rust's ownership system [5, 13] is based on the idea that if a certain object (of type `T`) is owned by multiple aliases (`&T`), then none of them can be used to modify it.

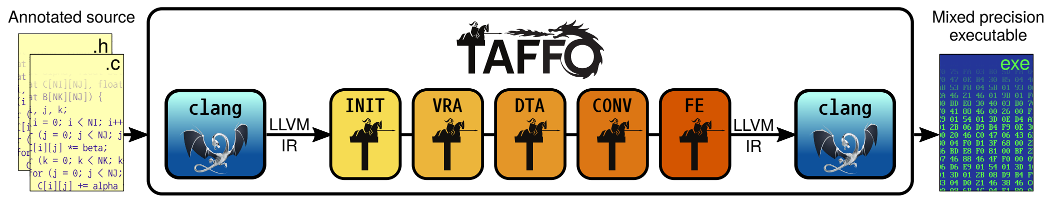

Figure 1 Architecture of TAFFO.

The borrow checker can be a severe limitation when implementing data structures and synchronisation mechanisms, which require the ability to mutate an aliased state. To overcome these limitation, Rust provides the `unsafe` keyword which allows developers to switch to unsafe Rust. Unsafe Rust is a strict superset of Rust that enables developers to perform actions that are normally prohibited, among which the invocation of `unsafe` function will be relevant to this work. It is important to understand that the `unsafe` keyword does not disable any of the other Rust safety checks, and it does not interact with the borrow checker in any way.

The combination of strict memory safety guarantees and flexibility provided by the `unsafe` code regions makes Rust a desirable target for our efforts. However, it is critical to investigate the best approach to integrate the precision tuning process within the Rust environment. In particular, it is in our interest to reuse the memory checks provided by *rustc* and do not invalidate them. The *rustc* compiler internally employs three Intermediate Representations (IRs), namely *High-Level Intermediate Representation* (HIR), *Mid-level Intermediate Representation* (MIR), and *LLVM-IR*. The borrow checker works at the MIR level. As a consequence, analyses and transformations performed at the LLVM-IR level have no visibility on the constraints imposed by it, and may in principle thwart the safety guarantees provided by the language if they do not strictly adhere to the semantics imposed by the others intermediate presentation.

2.2 TAFFO: The Compiler-based Precision Tuner

TAFFO [1] is a precision tuning that operates at the level of the intermediate representation provided by LLVM (LLVM-IR). The archiecture of TAFFO consists of five LLVM analysis and transformation passes which cooperate in building the final precision-tuned program. These passes are *Initializer*, *Value Range Analysis* (VRA), *Data Type Allocation* (DTA), *Conversion*, and *Feedback Estimator* (FE), as depicted in Figure 1. All of these passes are independent from each other, as they share data entirely through the *metadata* facilities provided in LLVM-IR.

A programmer who wishes to use this framework should first add specific annotations to the source code in order to tell TAFFO which variables must be involved in the tuning process. This task is achieved through Clang *annotations* that can be added to any variable declaration – including function arguments – as shown in the following example.

```
float x __attribute__((annotate("target('init') scalar(range(-16384, 16384) final)")));
```

Through annotations the user of TAFFO may also specify estimates for the range of values that a variable can have at runtime. This hint is required for variables whose initial definition does not depend on any other annotated variable, i.e. for *input variables*. As an example, in the annotation shown above, the "`target`" declaration gives a name to the variable and informs TAFFO about where to start collecting value range information. The "`scalar`" declaration contains information about the values the variable will assume at runtime: in particular, "`range`" specifies their range, and "`final`" also adds the information that this range is valid for the entire execution of the program. After annotating the source code, the precision tuning operation is performed simply by invoking the "`taffo`" command line tool, which is a drop-in replacement for GCC or Clang. The output will be an executable tuned for the current build machine.

Internally, the passes of TAFFO operate by first reading the annotations inserted by the user (the *Initializer* pass). Then, the *VRA* pass calculates the numerical intervals for annotated variables and other variables that depend on them. At this point, the tuned data types are selected in the *DTA* pass. At the time of writing, three approaches are available within this pass. The first one is a simple greedy algorithm that always assigns the fixed-point data type with the highest valid point position to each variable. Alternatively, the user can manually specify a floating point format which will be used for all the annotated variables. Finally, the third and most sophisticated option allows for a fully automated tuning based on precision estimation through the construction and optimization of an integer programming model of the code [2]. At the end of the pipeline, the *Conversion* pass modifies the LLVM-IR using the data types picked by the previous passes. Optionally, the *FE* pass can provide the user with an estimation of the error in the tuned program [4].

3 A Precision Tuner for Rust

The key challenge when it comes to perform precision tuning on Rust is the positioning of the precision tuning action within the compilation flow of Rust code. As anticipated in Section 2.1, the *rustc* compiler implements its memory checks on the code when it reaches the MIR stage of the compilation pipeline. It follows that precision tuning – to avoid conflicting with the *rustc* checks – should be performed either entirely before this stage, or entirely after it. This situation can be compared to the dilemma of performing precision tuning on high-level description of the code, such as on the source code, or onto a lower-level representation of the code. As mentioned in a survey on this topic [3], fine-grained precision tuning requires a lower-level representation of the code, and therefore we opt for introducing the precision tuning process entirely after the *rustc* memory checks.

On the positive side, this challenge can easily be solved by employing precision tuning components that are already built for this level of code representation, such as TAFFO. However, there are also integration issue between TAFFO and the *rustc* compiler: one in particular is the mechanism underlying the creation of the programmer annotations, as required by the TAFFO Initializer pass. For C and C++, Clang annotations are employed for this purpose, as exemplified in Section 2.2. During the generation of the LLVM-IR code, Clang simply converts each annotation to a call to the `llvm.var.annotation` intrinsic exposed by LLVM. This intrinsic function takes four parameters, representing the annotated variable, the annotation text (a string), the source file name and line number of the annotation. Notice that these annotations are a built-in extension provided by Clang, and their existence is independent from TAFFO.

Ideally, the same approach could be replicated with *rustc*. Unfortunately, this is not possible since Rust does not have an equivalent annotation syntax, hence we must design a dedicated solution. To solve this problem, we propose a Rust-native variable annotation mechanism. This annotation mechanism leverages the metaprogramming features offered by Rust *procedural macros*. The peculiarity of this kind of macros is that they can manipulate the program's token stream. We introduce the `annotate!` procedural macro, which takes two parameters: the annotated code and the annotation string. This last string supports the same syntax as TAFFO annotations for C and C++. As an example, consider the following fragment of Rust code:

```
annotate!(let mut tmp = [0_f32 ; I * J],
        "target('init') scalar(range(-16384, 16384) final)");
```

In this snippet, the `tmp` variable is associated with a value range of $[-16384, 16384]$. The procedural macro `annotate!` translates the variable declaration into the following Rust code:

```
static mut range: &'static str = "target('init') scalar(range(-16384, 16384) final)";
static mut name: &'static str = "2mm.rs";
let mut tmp = [0_f32; I * J];
unsafe {
    var_annotation(
        &mut tmp as *mut _ as *mut i8,
        &mut range as *mut _ as *mut i8,
        &mut name as *mut _ as *mut i8,
        2,
    );
};
```

The original variable declaration is preserved, while two new variables – `range` and `name` – are declared to hold the annotation string and the name of the source file respectively. Finally, the expanded body of the macro calls the `var_annotation` function with the same four parameters as the `llvm.var.annotation` LLVM intrinsic in the original Clang TAFFO interface. `var_annotation` is indeed a call to the LLVM intrinsic, which is exposed through a feature of the Rust compiler that exposes all the LLVM intrinsics as a set of foreign functions, employing the foreign function interface for C calls (FFI).

It is worth noting that the FFI call to `var_annotation` is wrapped in an `unsafe` construct. Indeed, FFI calls in Rust are considered unsafe actions in general, and must always be wrapped in an `unsafe` block guard.

3.1 Soundness of `unsafe`

Since the `annotate!` procedural macro introduces an `unsafe` region in the code for each variable involved in the precision tuning, we need to assess the impact of these regions in terms of memory safety. As mentioned before, the `unsafe` region is added because of the presence of a compiler intrinsic – that is, a function defined and handled directly by the compiler. Intrinsics are typically provided by compiler implementations to allow the use of non-standard functionalities or extensions that heavily depend on specific machine instructions to be implemented in the most optimised way. The most common application of intrinsics is for exploiting vectorisation primitives when the language does not provide a machine-independent way to access them.

In terms of memory safety, it is useful to partition intrinsics between code-generating and non-code-generating ones. Code-generating intrinsics in LLVM include, for instance, `llvm.log10.*` and `llvm.sin.*`, which generate machine-optimised code. Non-code-generating intrinsics, instead, are used only for internal purposes by the compiler, and are ignored by the code generator. Code-generating intrinsics may be memory-unsafe, especially if the code they generate involves memory operations, whereas non-code-generating intrinsics are safe, unless the compiler uses the information they carry to otherwise affect which memory accesses are made.

In our specific case, `llvm.var.annotation` belongs to the *non-code-generating* class. It is only handled in the middle-end of LLVM by keeping the annotation content tied to its variable declaration, while it is ignored by the back-end code generation. These annotations do not prevent neither enable additional transformation steps in LLVM. Thus, we can conclude that our `unsafe` block is safe for the final binary since it does not generate any code, as long as TAFFO itself is safe – which we will informally prove in the following section. Furthermore, to prevent compiler crashes, TAFFO checks each argument to confirm that it is of the correct type before it is used. If an incorrect annotation is used, the compilation emits a warning, discarding the wrong annotation. Finally, TAFFO also deletes all annotation intrinsics it is able to read at the end of its transformation, adding another layer of safety.

3.2 Soundness of TAFFO

To ensure TAFFO's transformations do not affect Rust's runtime guarantees on memory accesses, we need to demonstrate that it does not interfere with the methods used by Rust to ensure these guarantees. Memory-safety depends on the fact that memory accesses occur at addresses that fall within the bounds of allocated memory. For scalar variables, this requirement only implies forcing all accesses to happen through references to the variable – i.e., avoiding explicit pointer arithmetics. However, for array accesses, the situation is more complex, since pointer arithmetics is induced by the use of indices. The *rustc* compiler first attempts to prove at compile time that all memory accesses occur within bounds. This attempt may fail, because the values of indices may not be entirely predictable at compile-time. In this case, *rustc* introduces a guarding conditional statement before the access operation, which checks whether the index is actually within the array bounds. To ensure the safety guarantees of Rust are always mantained, TAFFO's handling of memory allocations was modified in order to disable the following operations:

1. Increasing the size of the elements of an array.
2. Changing the size of a dynamic memory allocation.

With the introduction of these two additional constraints, we can ensure that TAFFO never modifies Rust's code in a way that violates memory access bound checks.

In our informal proof we are going to consider both the compile-time and the runtime scenarios. In the first scenario, *rustc* generates an LLVM-IR equivalent to the one generated by Clang for the equivalent C code. More in detail, the generated code consists of a sequence of two instructions: a `getelementptr` instruction which computes the correct address of an element given a base pointer and an index, and a `store` or `load` instruction that actually performs the access at that address. When TAFFO needs to generate a new `getelementptr` instruction to index a fixed-point array, it copies the original `getelementptr` used to index the corresponding float arrays, and modifies only the returned datatype. The returned datatype, however, must be consistent with the one employed at the time of buffer allocation. As a result TAFFO must also modify the buffer allocation code, whose location is unfortunately not always detectable nor modifiable at LLVM-IR level. In case the buffer allocation cannot be modified, TAFFO mantains safety by only allowing datatypes that are smaller than the original one, therefore employing only part of the allocated buffer and preventing any buffer overflow. In the second scenario, Rust introduces a guarding conditional by means of a branch in LLVM-IR. The branch employs a condition obtained by comparing integer data, which are not modified by TAFFO and therefore reflect the array element size employed in the Rust code before precision tuning. In order to maintain the validity of the guard condition at all times, TAFFO *must not* modify the size of the memory allocation being guarded. With this additional constraint alone, the Rust code will keep its memory-safety properties. However when TAFFO increases the element size the program will stop working, as array accesses to correct indices in the non-tuned program become locations outside of the buffer bounds. Therefore the first condition (not changing the size of the array elements) needs to be applied for runtime-checked arrays as well. Clearly, the goal of reducing the precision of a program is better achieved with *smaller* data types rather than with *larger* ones. As a result we expect a minimal impact to Precision Tuning from these restrictions.

4 Experimental Evaluation

A key feature of Rust is its high performance, which enables its use as a C or C++ replacement in embedded systems. The ability to combine the guarantees provided by this language with the even greater speedups obtainable through precision tuning makes the combination of TAFFO and Rust the ideal platform for development of high-performance applications. In this section, we perform an experimental assessment of the impact of TAFFO on the Rust compiled code, employing a new port of the PolyBench benchmark suite to Rust.

4.1 Experimental Setup

All the experiments were performed on an STM32F207ZG microcontroller chip that features an ARM® Cortex®-M3 32-bit RISC core operating at 120 MHz. This chip includes a region of Flash memory for code of 1 Mbyte, and 128 KBytes of working RAM. The ARM Cortex-M3 core does not have a hardware floating-point unit, hence all floating-point operations are executed in software emulation. We generated the board setup code using STMCubeMX, compiled it with Clang, and linked it with each benchmark as a separate compile unit for both C and Rust.

For C benchmarks, the Precision Tuning process was done through the unmodified "`taffo`" command line tool. Instead, for Rust benchmarks, we modified the "`taffo`" tool to accept Rust code. In order to introduce Rust within the TAFFO compilation pipeline, invocations to Clang were replaced with invocations to *rustc* used as a front-end to generate LLVM-IR code. The rest of the pipeline is identical to the one employed for C code. This method ensures that both C and Rust codes are processed in the same way, applying the same optimization levels in the middle- and back-end. Additionally, we take into account that *rustc* is able to perform some optimizations at the internal IR level (MIR, discussed previously in Section 2.1), by passing the option "`-Z mir-opt-level`" to the *rustc* frontend. This is not the case and is not required for Clang. The baseline non-tuned benchmarks were compiled using the *rustc* and Clang tools. The version of LLVM and Clang employed was 15.0.7, while the version of *rustc* was 1.64.0. The optimization level used in `opt` for Rust and C was `-O3`.

4.2 Polybench/Rust

For our experimental work we selected the PolyBench suite Version 4.2.1, as its intended purpose is indeed to evaluate novel compiler optimizations, and as such it has been widely adopted in the literature. It consists of several numerically-intensive kernels which natively rely on floating point arithmetics, making it a good target for Precision Tuning. Moreover, its C implementation is already supported by TAFFO [1], which we will employ as-is.

For our comparison, we must employ a port of PolyBench to Rust which is as similar as possible to the baseline C code, in order to reduce any confounding factor determined by implementation differences between the benchmarks. To this end we evaluated the existing rewrite of PolyBench in Rust, PolyBench-rs[2]. However, after careful consideration

[2] `https://github.com/JRF63/polybench-rs`

we deemed this port not suitable for a direct comparison with PolyBench/C, as it heavily modifies the original benchmarks by adopting a functional programming style. While the choice is legitimate per-se, analysis of the resulting LLVM-IR code evidenced several non-functional differences with PolyBench/C, which result in very different execution time and memory consumption patterns between the two. For this reason, we developed a new port of PolyBench/C, named PolyBench/Rust, which aims to be as close as possible to PolyBench/C while still avoiding any *unsafe* block. In all the following experimentations, we have used the SMALL_DATASET as the array size. A comparison between the baseline PolyBench/C and PolyBench/Rust is provided in Figure 2.

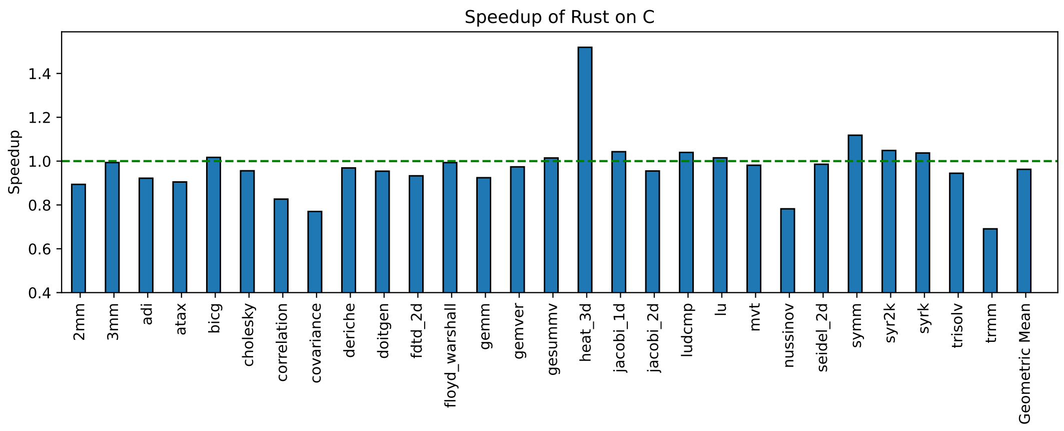

Figure 2 The ratio of the execution time between PolyBench/Rust and PolyBench/C as a speedup. The behavior of the two benchmarks is very similar, with exceptions for *heat_3d*, *correlation*, *covariance*, *nussinov* and *trmm*. The geometric means of the speedup of all benchmarks is represented in the last entry under "Geometric Mean".

Indeed, we get very similar execution times to PolyBench/C. In 15 benchmarks out of 28 the speedup is between 0.9 and 1, and in 8 benchmarks the speedup is between 1 and 1.1. One benchmark has a speedup greater than 1.1, *heat_3d*, while 4 benchmarks have a speedup lesser than 0.9 (*covariance*, *correlation*, *nussinov* and *trmm*). Overall, we notice PolyBench/Rust is slightly slower than PolyBench/C, due to the additional runtime checks introduced by the language. For the specific case of *heat_3d*, the *rustc* compiler applies – at the LLVM-IR level – an aggressive loop unrolling strategy enabled by better alias disambiguation offered by Rust. Speedups lower than 0.9 are instead caused by a combination of a greater amount of runtime checks and less efficient code generation from *rustc*. While for *trisolv* and *trmm* the main cause of the slowdown are the bound checks, for *correlation* and *covariance* the main cause is a greater number of memory accesses around floating point emulation calls that could be optimized away by Clang but not by *rustc*. We expect most of these differences to be smoothed over and disappear as *rustc* matures as a compiler.

4.3 Experimental Results

Figure 3 displays the speedups of each benchmark in PolyBench/C and PolyBench/Rust achieved through the Precision Tuning process of TAFFO (blue and orange bars). The baseline is always the same benchmark, for the same language (Rust or C), but compiled without TAFFO. After Precision Tuning, PolyBench/Rust is faster than PolyBench/C in 9

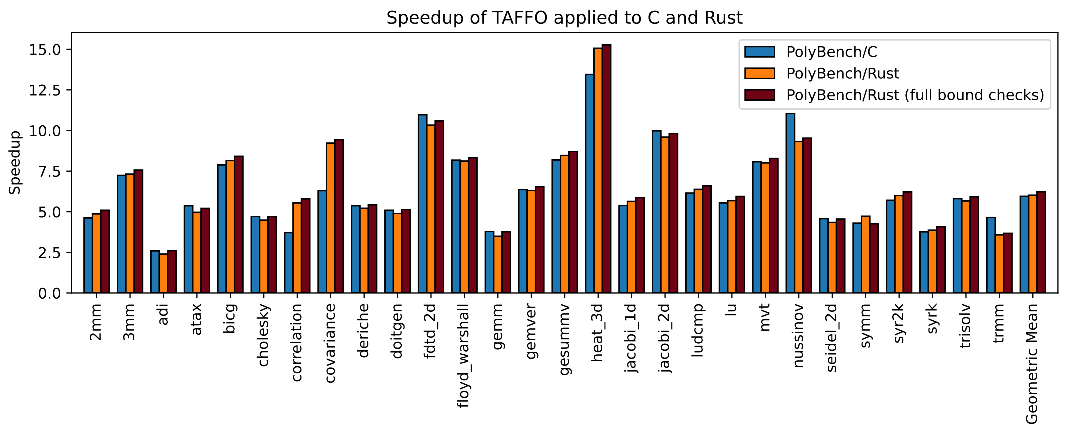

Figure 3 Comparison of the speedup achieved by TAFFO on PolyBench/C and PolyBench/Rust. Rust results are also shown with additional bound checking. The geometric means of the speedup of all benchmarks grouped by their configuration is represented in the last entry under "Geometric Mean".

out of 28 benchmarks, but aside from a few outliers, TAFFO performs in a similar way in both languages. The speedup for Rust ranges between 2.3 and 15.0, while the speedup for C ranges between 2.5 and 13.4.

Looking at individual benchmarks, *covariance* and *correlation* get a much higher speedup in Rust than in C. This is due to the aforementioned fact that *rustc* sometimes generates suboptimal code around calls to floating point emulation functions. After the Precision Tuning passes performed by TAFFO, the floating point emulation code gets replaced with integer fixed-point primitives, lessening the register allocation constraints for the backend, which then manages to eliminate redundant memory accesses. Benchmark *heat_3d* is another example where Rust code achieves a higher speedup with respect to the C one. In this case, the speedup is due to constant propagation optimisations enabled by the reordering of instructions performed by *rustc*, which further improves with the aggressive loop unrolling mentioned in Section 4.2 and is amplified by the TAFFO precision tuning transformations.

Overall, the difference in the speedup introduced by TAFFO on Rust and C is mainly due to the fact that *rustc* generates code that more closely represents the actual data flow within the program. While the PolyBench/C code relies on re-using existing arrays for storing intermediate computation results, Rust is able to remove these superfluous array accesses by using temporaries instead. Indeed, Rust guarantees that accessing memory through a mutable reference cannot be aliased, hence the compiler can prove that the re-use of an array in such a way does not have side-effects. This transformation makes the overall data flow more visible to the analyses and transformations of TAFFO (especially the Value Range Analysis), allowing for more performant code in the end.

Since most Polybench kernels are written to work with fixed-size arrays, only a few require extensive bound checks in the code for ensuring safety. Real-world applications, though, may exhibit more arrays with size dependent on input data, thus leading *rustc* to be less effective when optimizing bound checks than it would appear from benchmarking. In order to show how TAFFO is affected when the code contains more extensive bound checks, in Figure 3 we also show the speedups obtained when disabling bound check optimization (orange bars). The speedups are extremely similar to the ones obtained with the bound check optimization enabled, so we can state that TAFFO is not significantly affected by it.

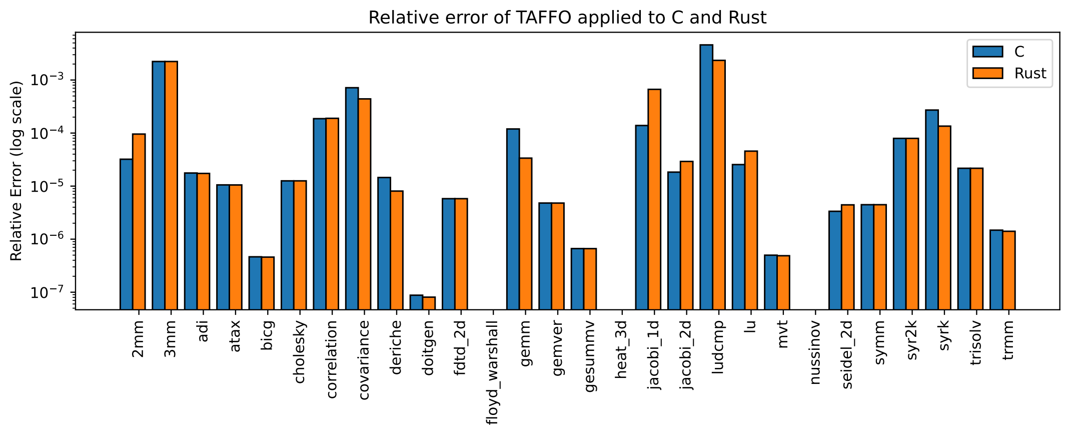

Figure 4 The graph displays the relative error of TAFFO C and TAFFO Rust as compared to the C version. The plain C version and the plain Rust version generate identical values.

Finally, we compute the relative errors resulting from the Precision Tuning approximation for both programming languages, which are shown in Figure 4. These figures do not depend on the amount of bound checks performed by the code. First of all, we observe that in most cases the relative error introduced is the same both for Rust benchmarks and C ones. The largest relative error was found to be approximately 0.45%, which occurred in the case of *ludcmp* of PolyBench/C. We can conclude that TAFFO introduces errors of similar magnitude regardless of the source language.

5 Conclusions

We provide the first automated Precision Tuning framework supporting the Rust programming language. We employ TAFFO to provide the analyses and transformation required for Precision Tuning, and we integrate it with the Rust compiler *rustc* by introducing the appropriate macros that allow for passing the programmer annotations required by TAFFO from Rust. As these macros use the unsafe features of Rust, we demonstrate that they do not actually affect the actual memory-safety of the code. Then, we evaluate our approach by developing and annotating a Rust version of the PolyBench suite for use with TAFFO. We see that Precision Tuning in Rust is just as effective as Precision Tuning in C, within a margin of error that can be fully attributed to differences in the compiler front-end, and further, Precision Tuning can help in offsetting the penalties imposed by the introduction of runtime checks. Rust is just one of the several memory-safe programming languages available for use in embedded systems, therefore a natural extension of this work includes support for languages such as Go and Swift. A further research direction is the design and implementation of Rust language extensions that allow the integration of precision tuning without the need for potentially-unsafe macros for generating compiler intrinsics.

References

1 Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. TAFFO: The compiler-based precision tuner. *SoftwareX*, 20:101238, 2022. doi:10.1016/j.softx.2022.101238.

2 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 673–678, 2021. doi:10.1109/DAC18074.2021.9586303.

3 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys*, 53(2), April 2020. `doi:10.1145/3381039`.

4 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Agosta Giovanni. Dynamic precision autotuning with TAFFO. *ACM Transaction on Architecture and Code Optimization*, 17(2), May 2020. `doi:10.1145/3388785`.

5 David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, October 1998. `doi:10.1145/286942.286947`.

6 MITRE Corporation. The common weakness enumeration (CWE) initiative, 2021. URL: `http://cwe.mitre.org/`.

7 Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. Ironclad C++: A library-augmented type-safe subset of C++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '13, pages 287–304, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2509136.2509550`.

8 Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 597–616. USENIX Association, August 2021. URL: `https://www.usenix.org/conference/soups2021/presentation/fulton`.

9 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75. IEEE Computer Society, 2004.

10 National Security Agency (NSA). The national security agency (nsa) cybersecurity information sheet, 2022. URL: `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`.

11 StackOverflow. 2023 developer survey. `https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages`. Accessed: 2023-12-01.

12 Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. Exploiting errors for efficiency. *ACM Computing Surveys*, 53:1–39, July 2020. `doi:10.1145/3394898`.

13 Jesse A. Tov and Riccardo Pucella. Practical affine types. *SIGPLAN Not.*, 46(1):447–458, January 2011. `doi:10.1145/1925844.1926436`.