Towards Understanding the Runtime Performance of Rust

Yuchen Zhang[†] *Yunhang Zhang*[‡] *Georgios Portokalidis*[†] *Jun Xu*[‡]
[†]*Stevens Institute of Technology* [‡]*The University of Utah*

ABSTRACT

Rust is a young systems programming language, but it has gained tremendous popularity thanks to its assurance of memory safety. However, the performance of Rust has been less systematically understood, although many people are claiming that Rust is comparable to C/C++ regarding efficiency.

In this paper, we aim to understand the performance of Rust, using C as the baseline. First, we collect a set of micro benchmarks where each program is implemented with both Rust and C. To ensure fairness, we manually validate that the Rust version and the C version implement the identical functionality using the same algorithm. Our measurement based on the micro benchmarks shows that Rust is in general slower than C, but the extent of the slowdown varies across different programs. On average, Rust brings a 1.77x "performance overhead" compared to C. Second, we dissect the root causes of the overhead and unveil that it is primarily incurred by run-time checks inserted by the compiler and restrictions enforced by the language design. With the run-time checks disabled and the restrictions loosened, Rust presents a performance indistinguishable from C.

ACM Reference Format:
Yuchen Zhang[†] *Yunhang Zhang*[‡] *Georgios Portokalidis*[†] *Jun Xu*[‡], [†]*Stevens Institute of Technology* [‡]*The University of Utah* . 2022. Towards Understanding the Runtime Performance of Rust. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559494

1 INTRODUCTION

Rust is a rising programming language designed to build system software [4, 10, 20]. On the one hand, Rust offers access to and control of the low-level system resources. On the other hand, unlike conventional systems programming languages, Rust ensures memory and concurrency safety. Thanks to these properties, Rust has gained tremendous popularity in recent years and has been adopted to develop infrastructural system software such as operating systems [3, 14, 18, 19], web browsers [16], and embedded systems [21].

One thing that has been broadly discussed but less systematically understood is the performance of Rust. Many people have argued that Rust is as fast as or even faster than C [2, 8, 12, 15, 17]. This is understandable since Rust introduces many language features and strict compiler checks to minimize run-time operations. For instance, Rust enforces explicit lifetimes of data objects so that all use-after-free issues can be identified at compilation time, avoiding the need for run-time checks. However, despite all those efforts, Rust can still introduce or force the use of extra operations, compared to C. For instance, §2.1 will point out that Rust often inserts bound checks at the execution time to rule out out-of-bound accesses. In this regard, Rust should be slower than C. But, *is this true*, and if so, *by how much and why*? Those questions are essential for understanding the performance of Rust but remain unanswered.

In this paper, we aim to shed light on the above questions by comparing the performance of Rust *v.s.* C. The key challenge is to build two sets of benchmark programs where the only difference between the two sets is the programming language. Previous observations of Rust performance are not based on such benchmarks. For instance, the Rust version of Coreutils [2] has been found faster than the C version in some programs. However, the two versions are likely using different algorithms and designs in certain places. Thus, the performance gain can be attributed to the implementation instead of the programming language.

To address the above challenge, we opt to build a set of micro-benchmark programs. All the programs are collected from The Algorithms [22], a large open-source algorithm library. Each program comes with both the Rust version and the C version, and we manually verified that the two versions implement the same algorithm, follow the same code structure, and use similar data structures. Further, the programs do not involve external libraries or system calls, allowing a more controlled comparison. In addition to these **Algorithm Benchmarks**, we further collect another set of programs from The Computer 22.05 Language Benchmarks Game [6], a platform asking for the fastest implementations of various computation tasks in different programming languages. Each program in this set also includes a Rust version and a C version. Both versions use the same algorithms, but they have different code structures, data structures, and library functions (which the developers believe to be the best option in the corresponding programming language). We call the second set of programs **Game Benchmarks**.

Measuring the performance on the two benchmarks, we observe that the Rust version is in general slower than the C version, but the extent of slowdown varies across different programs. Considering C as the baseline, the Rust version incurs an average "overhead" of **1.96x** and **1.35x** on the Algorithm Benchmarks and the Game Benchmarks, respectively. Throughout our manual analysis, we find the major source of performance overhead is the run-time checks inserted by the compiler to ensure memory safety. For instance, out-of-bounds checks account for 52.7% of all the overhead. Going beyond run-time checks, restrictions enforced by the language features can also lead to extra overhead. For example, Rust disallows

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3559494

indexed access to strings. Thus, any access to individual elements in a string often needs to convert the string to a vector first, inevitably introducing extra cost. More details can be found in §4.

This paper makes the following contributions.

- **New Benchmarks.** We build two set of micro-benchmark programs to compare the performance of Rust and C. All the benchmark programs are open-sourced and available at the repository: https://github.com/yzhang71/Rust_C_Benchmarks.
- **Performance Measurement.** We measure the performance of Rust and C on our micro-benchmark programs. It brings quantitative evidence to unveil the performance of Rust in various computation tasks.
- **Overhead Understanding.** We present a breakdown analysis on the performance overhead of Rust. The analysis gives an understanding of the major sources that introduce performance overhead to Rust.

2 BACKGROUND AND MOTIVATION

2.1 Safety Assurance of Rust

Rust introduces many mechanisms to ensure memory safety and concurrency safety. Those mechanisms are the major sources of extra run-time operations and the corresponding performance slowdown. In general, the mechanisms can be classified into several categories. The first category is language designs, which are validated and enforced at compile time:

- **Lifetime:** Every Rust object has a lifetime that is explicitly regulated. At compile time, Rust relies on lifetime analysis to ensure no reference can happen to an object that has not been created or has expired. This avoids use-after-free.
- **Ownership:** Each value in Rust is owned by a single variable that decides its lifetime. The ownership will be relinquished if the value is moved to another variable or function. This property can help avoid issues like double free: when the variable representing a buffer is freed, the variable loses the ownership of the buffer and, thus, it cannot be used with free again.
- **Borrowing:** Apart from moving or transferring the ownership, Rust also allows borrowing a value. Borrowing in Rust authorizes programmers to have multiple references to the same value throughout the lifetime of the owner variable without violating the "single owner" concept.
- **Exclusive Mutability:** With the borrowing mechanism, objects can be passed by reference. There are two types of references in Rust: 1) immutable reference to support one or more read accesses on a borrowed object, and 2) mutable reference that only allows one write access to a borrowed object. The compiler ensures an object cannot be both immutably and mutably borrowed at the same time, which helps avoid data races.

The second category of mechanisms is static compiler checks. For instance, the Rust compiler prohibits dereference of raw pointers unless it occurs in unsafe code. Static compiler checks have no direct impact on performance. Hence, we omit more details. In many cases, static checks are infeasible. Thus, Rust further adopts the mechanism of run-time checks, which we summarize below.

- **Out-of-Bounds:** Given an access to a stack-based array where the index cannot be determined statically, the Rust compiler will insert run-time checks to detect out-of-bound accesses. Given a dynamically sized data object (*e.g.,* a vector), Rust presents it as a "`fat-pointer`", which consist of a pointer to the data, the data size, and the capacity. At an access to the data object, the Rust compiler will insert a bound check, leveraging the length field in the fat pointer. Code below shows how this is done:

```
%_5 = icmp ult i64 %idx, %length ; compare idx with len
%1 = call i1 @llvm.expect.i1(i1 %_5, i1 true)
br i1 %1, label %bb1, label %panic ; abort if idx => len
```

- **Integer-Overflow:** Given an arithmetic operation on signed/unsigned integers where the value of any operand cannot be determined, the Rust compiler will insert a check for integer overflow detection. The code below shows how this is done using an LLVM intrinsic function. In practice, the integer-overflow checks are disabled by default when a program is built with the "`-release`" option.

```
%11 = call @llvm.uadd.with.overflow(i8 -1, i8 %u8_1)
%_23.0 = extractvalue { i8, i1 } %11, 0
%_23.1 = extractvalue { i8, i1 } %11, 1
%12 = call i1 @llvm.expect.i1(i1 %_23.1, i1 false)
br i1 %12, label %panic, label %bb9 ; abort if overflow
```

- **Division-by-Zero:** Given a division or modulo operation where the denominator cannot be determined to be non-zero, the Rust compiler will add a check for division-by-zero. The code below shows an example check.

```
%_34 = icmp eq i8 %denominator, 0 ; compare divisor with 0
%11 = call i1 @llvm.expect.i1(i1 %_34, i1 false)
br i1 %11, label %panic, label %bb9 ; abort if divisor = 0
```

2.2 Motivation

Rust remains young but is gaining market share in the industry. Many influential software vendors have been considering Rust for developing their products [7], including Meta [11], Google [18], and Amazon [13]. For these industry users to confidently and wisely adopt Rust, a key factor to consider is its run-time performance, compared to conventional system programming languages, in particular C.

In fact, the controversy about "is Rust as fast as C" has lasted since Rust was born. Some unofficial discussions [8, 15] indicate the answer is "Yes" and Rust can be even faster than C in certain cases. Their main reason is that Rust offloads many run-time checks to the compiler and offers better optimization opportunities (e.g., it can inline more functions). However, no data exists to back up their claims.

Another line of work, which can reflect on the above question, is the migration of existing C applications to Rust. Many of the cases show that the Rust replacement can run faster than the original C implementation. For instance, the "hck" [17] tool, a Rust version of the "cut" command, is demonstrated to run faster than the original C version. A similar observation is obtained on some programs from the Rust version [2] of GNU Coreutils. To the contrary, the Rust versions of low-level networking applications developed by Michal [12] only achieve 85% of the performance of their C counterparts. While these cases come with data, they still cannot fairly compare the performance between Rust and C. The Rust version may use new algorithms and different code structures. Thus, the performance difference can be a result of the disparity in implementations instead of the languages.

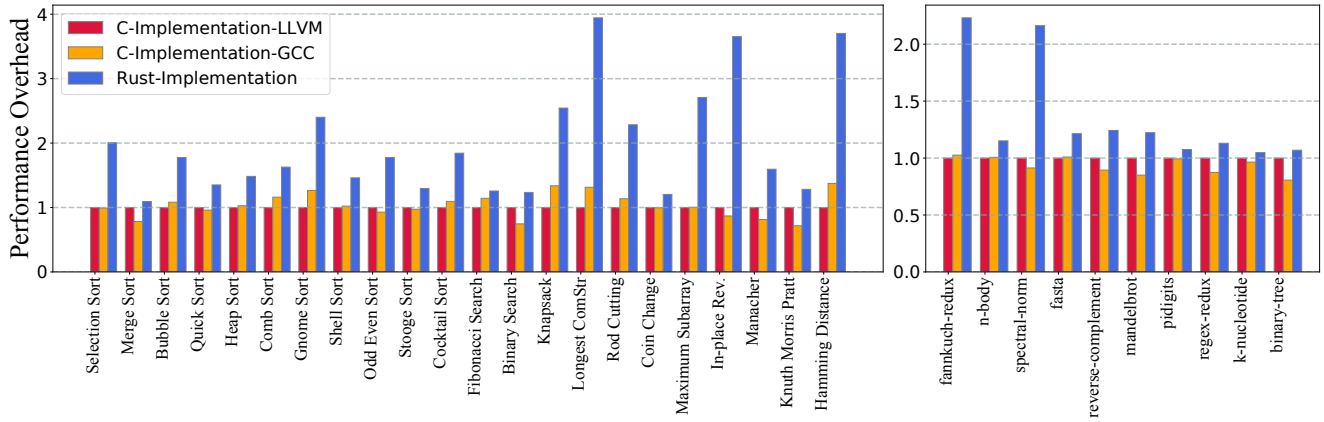

Figure 1: Performance comparison between `C-LLVM`, `C-GCC` and `Rust` on Algorithm Benchmarks (the *left* part) and Game Benchmarks (the *right* part). The `C-LLVM` version is considered as the baseline.

This study is motivated by the above practice. It aims to ***initiate*** *a quantitative, fair performance comparison between Rust and C*, thus providing insights for future applications.

3 PERFORMANCE COMPARISON: RUST V.S. C

3.1 Dataset Collection

To enable a fair comparison of Rust and C, it is essential to build two sets of programs that have minimal implementation differences. Towards this goal, we searched for public platforms gathering implementations of the same computation task in different programming languages. Eventually, we picked two platforms, The Algorithms [22] and The Computer 22.05 Language Benchmarks Game [6]. Both platforms provide programs developed in Rust and C for the same algorithm or "game". Importantly, their programs are small-sized[1], which can be manually validated to rule out implementation variations. We detail the collected programs below.

Algorithm Benchmarks: From The Algorithms library, we selected 22 programs that implement classic algorithms in both Rust [1] and C [9]. Names of the algorithms can be found in Figure 1 and all the programs are contributed by irrelevant third parties. We manually inspected the code of each program and ensured that the two versions (i) implement the same algorithm, (ii) follow the same code structure (e.g., both use for loops), (iii) use similar data structures when possible, and (iv) involve no library functions and system calls. We envision that, this way, the implementation differences are reasonably minimized. To further demonstrate a more fine-grained analysis, we also separate the benchmarks into different categories: "Compute-Intensive (Sorting Algorithm)", "Memory-Intensive (Searching Algorithm)" and "Memory-Heavy (Dynamic Programming)". Due to the fact that the programs do not come with test cases, we created a million test cases with random values and random lengths for each program. We also ensured that both versions receive the same set of inputs. All the programs, both the Rust and the C version, are compiled with optimization level O3, to ensure optimal performance is presented.

[1]On average, programs from the two platforms have about 50 lines of code and 125 lines of code, respectively.

Game Benchmarks: From the Computer 22.05 Language Benchmarks, we picked 10 code snippets (see Figure 1) and their fastest implementations in Rust and C. As required by the platform, both versions use the same algorithm, which we manually confirmed. However, the two versions may not use similar code and data structures. Also, they may use different library functions and system calls. While this benchmark involves an implementation discrepancy, we include it to complement the Algorithm Benchmarks. Essentially, it will demonstrate that, when using the same algorithm for the same task, how much performance difference can arise between the best Rust implementation and the best C implementation. All the programs are shipped with test cases, which we reuse in our evaluation. Similarly, we compile the programs with O3.

3.2 Experimental Setup

We compiled the Rust programs with Rust version `1.61.0`, which is supported by `LLVM-14.0.0`. When compiling the Rust programs, we also enabled the run-time checks described in §2.1, as those are essential for the safety guarantee of Rust. For consistency, we compiled all the C programs using `LLVM-14.0.0` (C-LLVM). For illustration purposes, we also compiled the C programs using `GCC-7.5.0` (C-GCC). To minimize the effect of randomness, we sequentially repeat each test case of each program five times and report the average results. All our experiments are conducted on a machine running Genuine Intel Processor (i7-8700, 3.20 GHz, 6 cores, 64GB RAM) and Ubuntu 18.04 LTS.

3.3 Results

Following the setups above, we separately measured the run-time performance of the Rust and the C implementations. Figure 1 shows the comparison results. On both benchmarks, the Rust version is slower than both of the C versions for every program. The extent of slowdown varies across different programs. For instance, on the "Longest ComStr" algorithm, the Rust version is 3.9 times slower than the `C-LLVM` version, while on the "Merge Sort" algorithm, the Rust version is only slightly slower than the `C-LLVM`

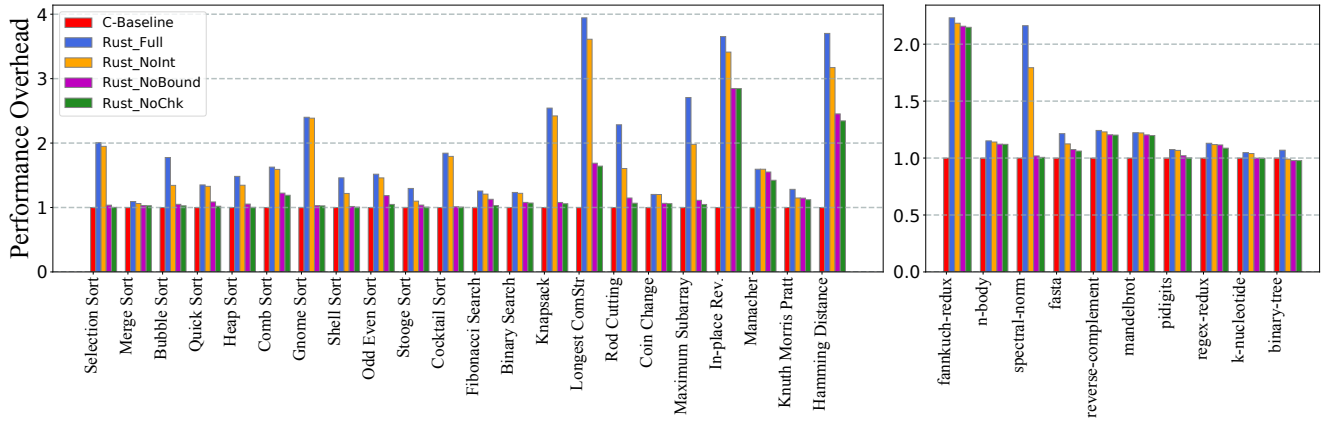

Figure 2: Performance comparison between Rust and C on Algorithm Benchmarks (the *left* part) and Game Benchmarks (the *right* part), after disabling Rust run-time checks. The red bar, representing the C version, is considered as the baseline. From left to right, the other bars show the overhead after we disable the corresponding checks in the Rust compiler, one after another.

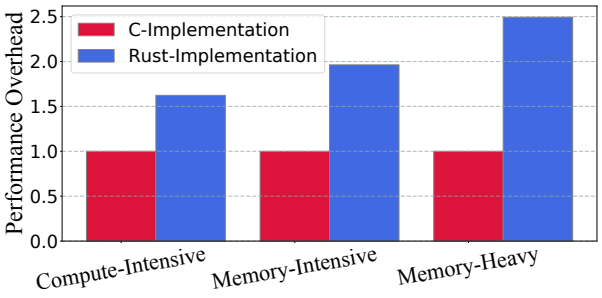

Figure 3: Performance comparison between Rust and C on different program categories. The C version is the baseline.

version. Considering `C-LLVM` as the baseline[2], the average performance overhead brought by Rust programs is `1.96x` on Algorithm Benchmarks and `1.35x` on Game Benchmarks. The lower performance gap on the Game Benchmarks seems to indicate that implementations can offset the performance disadvantage of Rust. However, the Rust programs in the Game Benchmarks prevalently include "unsafe" code blocks, where extra operations and run-time checks are omitted. We further break down the benchmarks into "Compute-Intensive" ones, "Memory-Intensive" ones (issuing intensive memory accesses), and "Memory-Heavy" ones (using a large amount of memory). As shown in Figure 3, the three groups incur a performance overhead of `1.62x`, `1.96x`, and `2.49x` respectively.

4 RUST OVERHEAD DECOMPOSITION

4.1 Experimental Setup

To understand the performance overhead of Rust, we conducted a follow-up experiment where we disable the run-time checks in turn and measure the performance after each. The specific settings are as follows:

- **Rust_Full:** all run-time checks covered in §2.1 are enabled.
- **Rust_NoInt:** disable checks on integer overflows.
- **Rust_NoBound:** further disable out-of-bound checks.
- **Rust_NoChk:** all run-time checks are disabled.

[2]C-GCC presents a similar performance to C-LLVM on average. Using C-LLVM as the baseline, the performance overhead of C-GCC is 1.01x.

4.2 Results

Algorithm Benchmark: The evaluation results for this benchmark are shown in the left part of Figure 2. The C version is considered as the baseline. Evidently, the run-time checks are a major source of the performance overhead. With all run-time checks enabled, Rust incurs an `1.96x` overhead, which drops to `1.77x`, `1.27x`, and `1.23x` after we in turn disable integer-overflow checks, out-of-bound checks, and division-by-zero checks. Particularly, the out-of-bound checks contribute to a majority part of the overhead (about 52.7% of the total).

Game Benchmark: The evaluation of this benchmark leads to similar conclusions. As shown in the right part of Figure 2, enabling all run-time checks in Rust results in a `1.35x` overhead, using C as the baseline. By disabling all the run-time checks, Rust's performance is improved and only incurs a `1.18x` overhead on average. Similarly, the out-of-bound checks incur most of the overhead.

4.3 Discussion

In the above experiment with the Algorithm Benchmarks, the Rust implementations still incur a `1.23x` overhead even after disabling all the run-time checks. Further analysis found that the overhead is mainly caused by extra operations enforced by the safety requirements of Rust.

Case 1: *Rust uses saturating floating-point-to-integer conversions for type casting, which is conservative but incurs extra run-time operations.*

Rust ISSUE#10184 [5] reports that floating-point-to-integer casting can cause undefined behavior. By default, LLVM performs floating-point-to-integer casting with the "`fptoui`" instruction, which can lead the resulting integer to wrap around if the source is larger than the maximal value of the destination type. To prevent such issues, Rust programs are compiled to use an overloaded intrinsic function "`llvm.fptoui.sat.*`" (termed *saturating casting*) on any floating-point-to-integer conversion. However, the saturating casting introduces extra operations to ensure safety, as shown below:

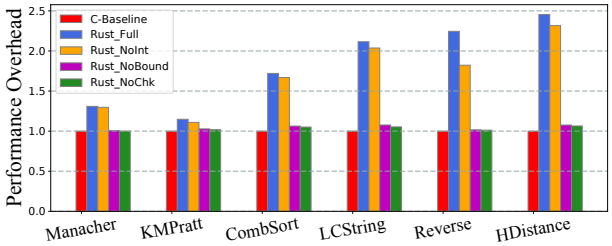

Figure 4: Performance comparison between Rust and C on certain programs from the Algorithm Benchmarks. The Rust implementations are fixed by us as described in the paper.

```
%A = fptoui float -1 to i8  ;-->  255
%B = fptoui float 256.0 to i8 ;-->  0
************************************************************
%X = call i8 @llvm.fptoui.sat.i8.f32(float -1)  ;-->   0
%Y = call i8 @llvm.fptoui.sat.i8.f32(float 256.0) ;--> 255
```

Specifically, when the source operand is smaller than zero, zero is returned. Otherwise, if the source operand is larger than the maximum value of the result type, the largest allowed value will be returned. Evidently, extra checks will be performed, compared to the native conversion.

In the Algorithm Benchmarks, "Comb Sort" involves saturating casting, which presumably leads to its `1.20x` performance overhead after all run-time checks are disabled. To confirm this hypothesis, we fixed the Rust code of "Comb Sort" to only involve integers and thus, remove the floating-point-to-integer casting. As shown in Figure 4, after the code is fixed, the Rust implementation presents performance comparable to the C implementation.

Case 2: *The Unicode encoding design guarantees the safety of strings, but it introduces extra overheads for modifying strings.*

By the programming language design, Rust prohibits access to strings through indexing. The main reason is that Rust uses UTF-8 encoding, where each ASCII character in strings is encoded as one byte, but other characters may require multiple bytes to store. Thus, directly indexing into strings' bytes does not necessarily correspond to a valid Unicode scalar value. For safety reasons, indexed access to strings is disabled to avoid undefined behaviors. However, such Unicode-safe string design has a trade-off. The extra conversion operation from "String" to "Vector" is often required before any modifications to strings in Rust. The code below showcases an example.

```
fn main() {
   let orig_string : String = "Hello, World!".to_string();
   let mut my_vec: Vec<_> = orig_string.chars().collect();
   ...
} // "my_vec" can be accessed or modified through indexing
```

The above is the main reason why "Longest ComStr", "In-place Rev", "Manacher", and "Hamming Distance" still incur an overhead after all run-time checks are disabled. To verify this part, we refactor the code to directly use "Vector" as input argument and redo the evaluation. As shown in Figure 4, without the extra conversion, the Rust implementation presents performance close to the C version.

5 CONCLUSION

This paper presents a study on the performance of Rust, considering C as the baseline. To support the study, we collect a set of micro benchmarks where each program is implemented with both Rust and C and the two versions only differ in the programming language. Our measurement based on the micro benchmarks unveils that Rust is in general slower than C, but the extent of the slowdown varies across different programs. On average, Rust brings a 1.77x "performance overhead" compared to C. We also unveil that the performance overhead of Rust is primarily incurred by run-time checks inserted by the compiler and restrictions enforced by the language design. With the run-time checks disabled and the restrictions loosened, Rust presents a performance indistinguishable from C. We envision that our study will shed light on a better understanding of Rust performance.

ACKNOWLEDGMENTS

We thank our the anonymous reviewers for their feedback. This research was supported by National Science Foundation (Grant#: CNS-2213727). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] Siriak Andrii and TheAlgorithms. 2019. All algorithms implemented in Rust. https://github.com/TheAlgorithms/Rust.
[2] Isaiah Ayooluwa, Ledru Sylvestre, and Ivy Roy. 2020. Rewriting the GNU Coreutils in Rust. https://github.com/uutils/coreutils.
[3] Kevin Boos. 2020. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on OSDI*.
[4] Yong Wen Chua. 2017. Appreciating Rust's Memory Safety Guarantees. *Government Digital Services, Singapore. July* 14 (2017).
[5] Daniel Micay. 2013. floating point to integer casts can cause undefined behaviour. https://github.com/rust-lang/rust/issues/10184.
[6] Isaac Gouy. 2022. The Computer Language 22.05 Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html.
[7] Sylvain Kerkour. 2021. 42 Companies using Rust in production. https://kerkour.com/rust-in-production-2021.
[8] Kornel. 2013. Speed of Rust vs C. https://kornel.ski/rust-c-speed..
[9] Vedala Krishna and TheAlgorithms. 2018. All algorithms implemented in C. https://github.com/TheAlgorithms/C.
[10] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.
[11] Meta. 2021. the Rust Foundation. https://developers.facebook.com/blog/post/2021/04/29/facebook-joins-rust-foundation/.
[12] Michal Niciejewski. 2021. Rust vs C: in low-level network programming. https://codilime.com/blog/rust-vs-c-safety-and-performance-in-low-level-network-programming/.
[13] Shane Miller and Carl Lerche. 2022. Sustainability with Rust. https://aws.amazon.com/blogs/opensource/sustainability-with-rust/.
[14] Vikram Narayanan. 2020. {RedLeaf}: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on OSDI*.
[15] Rhinotation. 2021. What makes Rust faster than C/C++? https://www.reddit.com/r/rust/.
[16] Servo. 2012. Servo: The Servo Browser Engine. https://servo.org/.
[17] Seth and Gert, Hulselmans. 2021. hck is a shortening of hack, a rougher form of cut. https://github.com/sstadick/hck.
[18] The Google Team. 2021. Android Gabeldorsche Bluetooth Stack. https://android.googlesource.com/.
[19] The Linux Team. 2021. Rust for Linux. https://github.com/Rust-for-Linux/linux.
[20] The Rust Team. 2015. Rust: A language empowering everyone to build reliable and efficient software. https://www.rust-lang.org/.
[21] The Tokio Contributors. 2016. Tokio: An applications with the Rust programming language. https://tokio.rs/.
[22] TheAlgorithms. 2022. Developers of The Algorithms. https://the-algorithms.com/.