Comparing C++ and RUST Performance Differences

Kushagra Srivastava   
*Manning CICS*  
University of Massachusetts *Amherst, MA*  
ksrivastava [at] umass [dot] edu

Prof. Meng-Chieh Chiu  
*Manning CICS*  
University of Massachusetts *Amherst, MA*  
joechiu [at] umass [dot] edu

Prof. Timothy Richards  
*Manning CICS*  
University of Massachusetts *Amherst, MA*  
richards [at] umass [dot] edu

*Abstract*—The following research paper attempts to compare between two of the most popular low-level systems programming languages: RUST and C++. We aim to find performance differences between the two by taking the x86 CPU Platform as a baseline, compiling to the Assembly, using a proprietary Assembly Tracer and Analyzer Program (*tra86*) built to trace the execution of the executable, and analyzing the output trace files. The findings lean towards favoring RUST, which on top of the security, inheritance, and debugging benefits, also provides a huge gain in performance on the systems level. The tooling and software created as part of this thesis are also available Open Source, at [https://tra86.skushagra.com](https://tra86.skushagra.com/).

Keywords—compilers, programming languages, assembly, systems, performance, C++, RUST, Intel x86

# Introduction

Computers are an intricate piece of machinery, wherein decades of development and contributions have shaped the field to what it is today. What started as an electronic method of crunching numbers in its infancy has evolved to the primary method of data exchange and communications today. These developments have taken place at various stages of computation: from low-level systems that communicate directly with the hardware, to high-level code such as Artificial Intelligence or Internet Webpages. Decades of contributions to all these different levels of computation have rendered the systems that we use daily.

The building block of any computer program are the set of commands that a programmer passes to the computer, namely via a programming language. In a compiled-language, the user-written source code is translated to Assembly, Assembled into an Object File with external files linked, and then compiled into an executable file. Thus, in modern-day programming, the programming language at use is one of the most crucial element throughout the developer's journey, and even more so in low-level code where dealing with hardware directly requires one to write optimized, high-performance, code.

The following Honors Thesis attempts to compare between two low-level programming languages: RUST and C++. These are both languages that were made during different eras, yet are widely utilized in low-level systems programming. C++, which was an evolution of C, was popularized during the 80s and 90s, and RUST was released in 2013 and has gained traction since. As a result of the vast time difference, both programming languages have different approaches as to how to achieve successful low-level code compilation. RUST prioritizes Code Safety, Security, and Optimization. On the other hand, C++ gives the user complete freedom and control over their code, even if it comes at a risk of breaking the system.

The motivation to compare the two programming languages lies in my interest to understand if RUST, being a newer low-level programming language, is an overall better option to use compared to C++ when it comes to low-level systems programming. The decades of contributions that have gone into C++ since release has made it the primary systems programming language. Thus, if an undertaking should be made to shift most of the code from C++ to RUST, the results of this thesis would give developers an adequate, hardware-agnostic idea of the performance benefits that RUST provides for their specific use case.

## Scientific Problem

The following research thesis draws inspiration from the dissertation, "Run Time Program Phase Change Detection and Prediction", authored by Prof. Meng-Chieh Chiu[1]. The dissertation attempts to trace phase detection in programs written on Java, C, Python, among many others using Machine Learning techniques. In a similar fashion, the following research thesis also attempts to trace program execution for RUST and C++ in a real-time, hardware-agnostic manner. The trace outputs generated through the executable programs for each programming language can then be analyzed using a Python script written as part of the thesis as well.

During the compilation of a program in each programming language, the compiler for a given programming language goes through different phases, namely: initialization, parsing, semantic analysis, optimization, and code generation. At the end of this process, the compiler generates a file known as Assembly, which essentially contains the logic of the code in a way that a CPU can parse through and execute. Assembly is a low-level human-readable language that consists of a set of instructions specific to a CPU architecture, and most compilers output the logic of their code in Assembly for a given CPU architecture. After this step, the program is translated to non-human-readable machine code: namely an Object file with header files linked to the same, before it is packaged into an executable (the "program"). For the execution of this program, the CPU goes sequentially through the Assembly instructions, and uses a certain amount of resources to execute each instruction. These resources, more specifically CLock Cycles, can be used to determine the performance of a program at the lowest level possible in the hierarchy of computation.

Crucially, it is to be noted that different combinations of these instructions can be used to achieve the same output. Therefore, to compare performance metrics between two programming languages, we can notice how each of their compilers generate their respective Assembly files for a given CPU architecture, which can then be "traced" in real-time during execution. Tracing, for the purpose of this thesis, is the method used to record each Assembly instruction that the CPU executes during the live run-time of a program, which can then be analyzed to deduce the amount of Clock Cycles the CPU has executed in the execution of this given program.

By writing the same program on these two languages, if we were to extract their respective Assembly files and analyze them, we can gain insights into how each language approaches code execution based on their trace outputs. Since different combinations of Assembly instructions can have the same execution, this can give an insight into how each compiler creates their respective Assembly file. For a hypothetical example, if we write the same program on RUST and C++, and the execution of the RUST version of the code required the CPU to process and execute lesser Assembly instructions than for the comparitive C++ program, we can say that RUST performs better than C++ in this specific case. Thus, by collecting real-time Assembly Instruction Trace outputs for the same exact code written in the two programming languages, these trace outputs can be analyzed to provide a comprehensive view of how optimized and performant each programming language is on a low-level scale. Moreover, this approach circumvents potential issues such as hardware requirements, operating systems, and different configurations of the machine or the state that the program is run on. Since the Assembly file for a given CPU architecture is consistent, low-level, and is executed the same regardless, we can get comprehensive insights for the two programming languages on the entire architecture by focusing on the execution and logic of the same.

I plan to conduct this research on the x86 CPU Architecture, popularized by Intel and AMD in consumer machines. The x86 Architecture consists of various instructions as detailed in the Intel IA64 and IA32 Architecture Manuals[2], and Prof. Agner Fog's Instruction Tables provide comprehensive results on Clock Cycle estimates on various x86 processors per instruction[3]. This hardware has seen immense development in both C++ and RUST, and the hardware has matured to a stable point over the past coupl of decades, having released in the 1980s at first[4]. Unlike novel architectures such as ARM, I believe that the x86 architecture would minimize potential setbacks to conduct such an experiment, especially when dealing with C++ and RUST compilers in an era-specific context. For example, a potential setback could arise with Apple's ARM Architecture popularized in the M1 processors and above, which rely on a lot of x86 Architecture Virtualization through Apple's proprietary translation mechanism: Rosetta 2[5]. These processors run older x86 programs in a software-defined x86 environment, which may lead to errors gathering rea-time Assembly Trace Output data (since it relies on fetching data during live execution). However, I believe that the tooling created as part of this thesis can be extended to other architectures as well, such as RISC-V or ARM, given the programs run were compiled native to the platform.

## Significance of the Problem

Most of the low level systems code is written in C or C++, and are continued to be used still today. A quick look at the Linux kernel shows that about 98.3% of the codebase is based on C[6], and GNU Coreutils (the set of programs that run on the Command Line and help a user in navigation and operation) is 59% based on C[7]. Similarly, macOS being a UNIX variant, and the Windows NT Kernel are mainly C-based as well[8]. This is mainly because C/C++ hold legacy value, and have seen a lot of contributions since their initial release in 1983. They have a dedicated community, ranging from Open-Source Developers to companies that are invested in the architecture due to back-end and server infrastructures that rely on them.

However, this also means that there is a lot of legacy holdover from these languages that are still being used today. For example, C++ is prone to various errors if users are not mindful: memory leaks, segmentation faults, kernel errors, and so on. Error messages given out by the compiler can be often cryptic and hard to understand, and debugging code is often more complicated than other high-level languages such as Python.

On the other hand, we have RUST: a language whose development started in 2010 and is continuing today. RUST attempts to employ stricter typesetting, better error tracking, and more stringent memory usage to run more efficiently. To demonstrate the differences in how RUST and C++ handle error tracking through a live example, let's take an example of a deliberate Race Condition below.

// Race Condition Demo, written by Kush.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int shared\_variable = 0;

void \*increment(void \*arg) {

for (int i = 0; i < 1000000; i++) {

shared\_variable++;

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

if (pthread\_create(&thread1, NULL, increment, NULL) != 0) {

perror("pthread\_create");

return 1;

}

if (pthread\_create(&thread2, NULL, increment, NULL) != 0) {

perror("pthread\_create");

return 1;

}

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

printf("Shared variable: %d\n", shared\_variable);

return 0;

}

}

In the C code, two threads are incrementing the shared\_variable concurrently, leading to a race condition where the final value is unpredictable. However, C will let us run this with no issues or errors, as demonstrated below 10 times on my own system:

suobset@Kush-Surface:a.out

Shared variable: 1138441

suobset@Kush-Surface:a.out

Shared variable: 1339034

suobset@Kush-Surface:a.out

Shared variable: 1070599

Since we were trying to change the same variable on different threads, the value of the output is unpredictable. However, upon translating the exact same code into RUST (not shown here) and trying to run it, the ownership and borrowing system of RUST actually prevents the code from compiling, raising an error:

error[E0502]: cannot borrow `shared\_variable` as mutable because it is also borrowed as immutable

--> src/main.rs:9:13

|

7 | let thread1 = thread::spawn(|| {

| ---------------- immutable borrow occurs here

8 | for \_ in 0..1\_000\_000 {

9 | shared\_variable += 1;

| ^^^^^^^^^^^^^^ mutable borrow occurs here

...

15| thread1.join().unwrap();

| ------------------------ mutable borrow later used here

Thus, I feel that it is worth to look into the performance benefits of RUST, on top of the smoother debugging and security features that the language provides. Being a more recent Systems Programming Language, RUST has not yet seen widespread adoption as anticipated initially.

Moreover, I believe that there exists a gap in knowledge pertaining to the performance metrics between RUST and C++ as a result of the majority of systems being C++ based. In the Software Developer landscape, companies and developers currently gravitate towards C++ because the costs attached with refactoring such huge codebases to an entirely new language takes time and effort, and the return of such an investment is currently unknown. Thus, a deep analysis into the comparison of the performance of RUST and C++ may give a holistic overview of both languages in a way that may help developers make a choice between RUST and C++ for their projects, find if there’s a return on investment into either of the two languages, as well as provide two major benefits:

* Environmental Impacts by more efficient programs scaled to servers.
* Lowers the barrier to low-level systems programming, thus attracting newer talent.

### Hypothesis 1: Environmental Impacts

According to Energy Innovation [8], global data centers consumed about 205 terawatt-hours (Twh) of electric power, or about 1% of Global Consumer Electricity Consumption. Let us put this in perspective: given the world population, servers alone accounted for the electricity that would have been used by 70,000,000 people. This number is about twice the population of Canada, about 65% of Mexico's Population, and about 4 times the population of Australia.

Hypothetically, let us assume that we have moved all low-level systems to RUST, which means that all servers in the world run on RUST now. While this is a bit flawed in its nature, let us also assume that there is a direct co-relation between energy consumption and the effectiveness of a language. If RUST enables, through its various carefully-constructed safety paradigms, about 5% more efficiency in servers, this would result in savings of about 10.25 terawatt-hours of electricity. That number is greater than the electricity used in about 119 countries of the world [9], even while having taken only conservative metrics, since inter-connection of servers through Networking, and a higher efficiency of consumer electronics (or client devices) has not been taken into consideration.

### Hypothesis 2: Lowers the Barrier of Entry to Low-Level Systems Development

Low Level Systems: including, but not limited to, CPU/GPU Architectures, Compilers, Operating Systems, Networking Interfaces, Communication Protocols, and the like are a culmination of decades of work, most of which really gained traction in 1970s. As a result, there exists a high amount of intimidating legacy code: thus, making the development process inaccessible to many.

The way languages such as C/C++ behave can also differ between systems and architectures. While most of the functionality remains the same, low-level system calls or memory management may differ at times making the same C++ code act differently. As a result, C++ behaves differently on different kernels and hardware, even today. Moreover, due to the large amount of time put into C++ development, there exist a tremendous number of different C++ versions and compilers, all acting differently and built for their specific purposes. This introduces a lot of noise and confusion for novices in the field, which can increase the barrier of entry into the field.

On the contrary, RUST was developed from the first day as an Open Source project, placing proper standardization across different platforms and architectures. Essentially, RUST behaves the same on every platform, every piece of hardware. Paired with incredible documentation, and a unified compilation process on every system, RUST makes low-level development significantly easier for novices and professionals alike.

In knowing if RUST has comparable, or better, performance than C++ will give an even more holistic view to developers to make a decision regarding the programming language of choice: C++ with free control over everything, or RUST with safety mechanisms and standardization in-place.

# Literature review

The key documentation for this thesis includes documentation for the RUST and C++ Compilers, which gave tremendous insights into how each language approaches compiling code into Assembly. I also read Professor Meng-Chieh Chiu’s Paper on analyzing Phase Change Metrics for Java Programs, and certain papers that analyze RUST and C++ Performance using different approaches, including a white-paper published by The White House advocating for RUST and its security features as described above. Lastly, I also consulted documentation on the x86 Architecture which I am using as a benchmark CPU architecture to conduct this research,. And on technologies that are used in the RUST and C++ compilers, such as the GCC and LLVM tool-chains.

Professor Chiu’s paper, “Real-Time Program Specific Phase Change Detection and Prediction” sets to explore and detect Phase Changes during compilation for programs written on Java and Python. It uses a Machine Learning Model that has been trained to explore when a program would switch phases during execution. More specifically, this would refer to how often a program would shift to executing a different segment of the code, in a hardware agnostic low-level manner. Phase Changes are detected through recording various time intervals between different phases, and to cluster them based on the similarity of their feature vectors: the number of similar properties that specific phases consist of (which would tell how similar of different a given phase is). These metrics were then analyzed using a Gaussian Mixture Model (GMM), a “probabilistic model generalizing k-means clustering to incorporate information about the covariance structure of the clusters” [1].

This research paper was crucial in expanding my understanding of how to conduct low-level systems research as an undergraduate, as well as gave me guidelines on how to make it such that we’re not incorporating any metrics into our evaluation that may be fixed using different specifications. For example, it was crucial for me to consider that the time required to successfully execute a program would not have been a robust metric, as this could be reduced by using more powerful CPUs. Moreover, this project by Professor Chiu involved tracing Python Bytecode, a low-level intermediate representation of Python code during compilation, similar to Assembly. However, Professor Chiu’s paper only deals with Java, Python, C, and with Compilation Metrics (as opposed to execution metrics). This project does not make comparisons between different programming languages, but gives insights into the worings of each of the languages mentioned above, and an excellent idea of how to approach problems in this given domain. Despite the shortcomings of the paper with regards to the aim of my research, Professor Chiu’s Bytecode Tracer became the groundwork for my x86 Assembly Tracer to compare Performance Metrics between RUST and C++, a product which I will be expanding upon this document soon.

I also referred to a case study, “Performance vs. Programming Effort between RUST and C on Multicore Architectures: Case Study in N-Body” by Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti [10]. This paper delves into the Performance metrics for RUST and C by conducting a comparative study of the two programming languages in high performance tasks. The study also investigates the Programming Effort went into creating similar programs across the two programming languages, noting it as a crucial aspect of the developer experience in getting optimized code out in the real-world. Notably, RUST has a lower entry barrier as compared to C with its optimized object-oriented approach to programming that, according to the paper, is missing in both C and Fortran (the latter being another Programming language used for High Performance Computing). The paper mentions further benefits that RUST beings to the table compared to C or Fortran: running equally as well on every piece of hardware, handling concurrency and parallel processing much better than C/Fortran, and preventing edge cases such as Race Conditions through ownership and inheritance paradigms ensuring that multithreading and parallel processing are safe.

However, this case study focuses mainly on High Performance Computing, and determines success for the performance of RUST being equal to C/Fortran for the given domain (with RUST having the easier coding experience as an edge). Thus, this does not convey enough information to incentivize developers to switch away from using C++ on everyday systems, and facilitate an investment to refactor from C/C++. This means that most systems continue to be written on C/C++, and keep the barrier of entry to low-level systems for novices high.

Another similar paper that I approached during the Literature Review was “RUST vs. C++: a battle of speed and efficiency” by Vincent Ng [11]. While this is a pre-published work, the approach taken in this paper is very similar to mine, with the author compiling the exact same programs in the two different programming languages. The author then measures the performance of these languages by measuring different metrics, such as Memory Usage, Execution Time, Compilation Time, and Lines of Code. However, this approach has shortcomings in measuring multithreading capabilities of the programs, as it is only based on adding layers and variables to single-threaded code to measure performance. This leaves very little to no nuance regarding multithreading, parallel processing, or multicore execution. Moreover, this paper relies heavily on the hardware configurations of the system: i.e. the metrics recorded were CPU and RAM consumption, which could be offset by using more powerful hardware. This leaves very little nuance in discussing about every hardware configuration, as it essentially finds the performance differences between the two languages in single-threaded programs on a specific specification.

Recently, a report published by The US White House Office of National Cyber Director (ONCD), urges towards a shift to RUST for creating more secure, robust, and efficient programs, as it has become a matter of national security [12]. The paper outlines how some of the biggest exploits of the Internet era have come from unsafe memory practices with languages that allow unsafe memory handling: such as C and C++. These languages require a programmer to manually allocate memory to different variables used in code, and deallocate them when no longer needed. While this offers a lot of flexibility and freedom, it opens up issues in two major categories: spatial and temporal.

Spatial memory errors happen when a programmer tries to read or assign data to arrays or variables outside of what has been allocated to a given memory space, i.e. an out of bounds error. Suppose that an array has been allocated 5 memory spaces. In a memory safe programming language (such as RUST), reading the 6th element of this 5-element array will throw an out-of-bounds error. However, in an unsafe language (such as earlier versions of C), the language lets the programmer read contiguous memory regions, regardless of what’s stored there.

On the other hand, temporal errors occur when a program tries to access memory that has already been deallocated. Once again, while a memory safe programming language would throw an appropriate error, unsafe languages may not do so and give a leeway for generating exploits. In the context of C, one of the examples mentioned are that of Pointers, a variable that stores the memory address of another variable. If the latter variable is deallocated, the pointer does not automatically get deallocated either. This pointer remains to point to the memory location, even if the same is now used for different purposes in the code. Accessing this memory location now by means of the pointer causes it to read whatever data was stored in this memory location for the new process (not what it was originally intended to read), causing a use-after-free bug.

The ONCD report advises programmers to use memory safe languages such as RUST. The report lays out the three requirements for a memory safe programming language, namely:

1. The language should allow the code to be close to the kernel to facilitate systems development that can integrate software and hardware.
2. The language must be deterministic in nature, so that outputs can be predicted and are consistent.
3. The language must not be able to override, the “garbage collector”: a function that reclaims memory allocated by the code no longer used.

C/C++, the most widely used systems programming languages, do not have these pre-requisites, which RUST does on the other hand. However, the report also mentions the adoption rates of RUST having been low as of yet. I believe that showcasing performance differences between RUST and C++, on top of the widely appealing reasons as listed by these various researchers, would make an appealing argument to favor a shift towards RUST in higher numbers.

For the purposes of this research, I also consulted documentation on the RUST and C++ Compilers, to understand how each of the compilers interacts with user-code and outputs them into Intermediate Representations, Assembly, any Object Files, Linking external files, and packaging it into an executable. I referred to the RUST Compiler Development Guide [13], which provides extensive manuals into downloading the compiler, making changes to the source code, and testing them on a local environment on any given computer. The RUST Compiler Development Guide also provides deep insight into the Low Level Virtual Machine (LLVM) toolchain that is used to output the x86 Assembly, and how the whole pipeline from user-code to an executable is laid out. This was a very crucial guide in understanding the intricate nature of RUST Code Compilation. The guide provided insights into the various stages of compilation: initialization, parsing, semantic analysis, optimization, and code generation. After these stages, the code is passed into the LLVM toolchain in the form of “LLVM Intermediate Representation,” from which the LLVM toolchain takes over and links it to LLVM Byte-code, and finally to Assembly.

On the other hand, the Clang Compiler User’s Manual [14] provided analogous insights into C++ compilation, emphasizing the commonality with RUST through the shared LLVM architecture for producing x86 Assembly. While both languages leverage LLVM architecture, the specifics of each compiler's implementation may lead to variations in the compilation pipeline. This resource enabled a detailed exploration of the stages involved in C++ compilation, ranging from source code to the creation of a computer program. However, despite the different specifications of the Clang Compiler, it also outputs LLVM IR, from which the LLVM toolchain takes over in a similar manner. By consulting these manuals, a holistic view of the compilation processes in RUST and C++ was achieved, allowing for a nuanced comparison between the two languages.

It is also crucial to know how LLVM bridges the gap between different CPU architectures. Different CPU architectures (such as x86, ARM, MIPS, etc.) have different Assembly instructions that they can parse and execute. However, the LLVM toolchain just requires the LLVM Intermediate Representation (LLVM IR) as its input, and can output Assembly for any architecture required. I was able to gain further insight into the LLVM toolchain from the “LLVM Compiler Infrastructure Documentation” [15], which also goes over instructions on installing LLVM and executing your own LLVM IR on the platform. Thus, while my focus remains on the x86 Architecture, I believe that using Clang, RUST, and LLVM may enable to conduct this research study in different CPU architectures as well.

Looking at the entire pipeline, roughly each compiler goes over the stages of initialization, parsing user code, semantic analysis, code generation, and outputs LLVM IR. The LLVM toolchain then takes this LLVM IR, and converts it to LLVM Byte-code, and then to the Assembly for any requested architecture. This gives us a compiler and hardware independent platform to compare the logic behind the two Programming Languages executing the same program.

I also consulted documentation on the Intel x86 Architecture, to gain an overview of the x86 Assembly that these processors can parse and execute. Specifically, I referred to the “Intel 64 and IA-32 Architectures Software Developer Manuals” [2], which give an in-depth look of the x86 Architecture, the registers and memory architecture built in, and the format of the Assembly that they can parse and execute, along with a glossary of all the instructions they can execute, and the purposes of the registers that they use to store information and data during runtime. The manual also provides in-depth information on resource optimization and clock cycles for each Intel Processor in production, and provides schematics and information on how each processor executes Assembly instructions.

A similar guide that has been crucial in my research work has also been the “x86 and amd64 Instruction Reference” by Felix Cloutier [16]. This guide provides a comprehensive list of all Assembly instructions used in the x86 Architecture in a concise manner, derived from the Intel Manuals mentioned earlier. Another similar resource by Professor Agner Fog, “Instruction Tables”, was equally crucial in deriving the amount of Clock Cycles and CPU resources used per x86 Assembly Instruction, as derived from various testing and experimentation [3]. The values of CPU instructions in these guides are derived from the Intel Manuals, as well as real-life testing of each instruction across various programs, and are what I use as my benchmark in the second half of deriving performance benchmarks from tracing x86 Assembly outputs. These two guides provide a comprehensive review of the amount of CPU resources that would have been spent in executing the Assembly output of the two Programming Languages, which is what lies at the crux of this research endeavor.

# Research Methodology

The main intent behind this Research Project is to compare Performance Metrics between RUST and C++ on the x86 Architecture, and in a hardware independent manner. Specifically, I want to create a pipeline that takes in a user program, and

##### References

1. *Meng-Chieh Chiu, “RUN-TIME PROGRAM PHASE CHANGE DETECTION AND PREDICTION: A Dissertation Presented by Meng-Chieh Chiu” (PhD Thesis, University of Massachusetts, 2018).*
2. Intel, “Intel® 64 and IA-32 Architectures Software Developer Manuals: Intel Developer Guides” (Intel, December 2023), <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
3. Agner Fog, “Software Optimization Resources,” November 2022, accessed February 16, 2024, <https://www.agner.org/optimize>.
4. Pryce, Dave (May 11, 1989). "80486 32-bit CPU breaks new ground in chip density and operating performance. (Intel Corp.) (product announcement) EDN" (Press release).
5. “Rosetta 2 on a Mac with Apple Silicon,” Apple Support, accessed April 27, 2024, <https://support.apple.com/guide/security/rosetta-2-on-a-mac-with-apple-silicon-secebb113be1/web>.
6. The Free Software Foundation, GNU Coreutils, n.d., accessed April 30, 2024, <https://github.com/coreutils/coreutils>.
7. Apple, Inc., GitHub - Apple-Oss-Distributions/Distribution-MacOS at Macos-144, n.d., accessed April 30, 2024, <https://github.com/apple-oss-distributions/distribution-macOS>.
8. Silvio Marcacci, “How Much Energy Do Data Centers Really Use?,” *Energy Innovation: Policy and Technology*, last modified March 17, 2020, accessed April 30, 2024, <https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/>.
9. Various Authors, “List of Countries by Electricity Consumption: Wikipedia, the Free Encyclopedia,” Wiki, December 11, 2023, <https://en.wikipedia.org/wiki/List_of_countries_by_electricity_consumption>.
10. Manuel Costanzo et al., “Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body” (Master’s Thesis, Universidad Nacional de La Plata, 2021), <https://arxiv.org/abs/2107.11912>.
11. Ng, Vincent. “Rust vs. C++: A Battle of Speed and Efficiency: A Research Paper on the Performance Differences between Rust and C++ and Answers Which Language Is Better for Performance-Based Applications.” Research Paper. Kuala Lumpur, Malaysia: St. Joseph International School, 2023. <https://doi.org/10.36227/techrxiv.22792553.v1>.
12. ONCD. “Press Release: Future Software Should Be Memory Safe.” The White House, February 26, 2024. <https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/>.
13. The RUST Compiler Developer Team. “RUST Compiler Developer Guide: The RUST Programming Language,” n.d. <https://rustc-dev-guide.rust-lang.org/>.
14. The Clang Team. “Clang Compiler User Manual: LLVM Documentation,” n.d. <https://clang.llvm.org/docs/UsersManual.html>.
15. The LLVM Compiler Infrastructure Developers. “The LLVM Compiler Infrastructure Documentation: Low Level Virtual Machine,” n.d. [https://llvm.org](https://llvm.org/).
16. Felix Cloutier. “X86 and Amd64 Instruction Reference,” May 2023. <https://www.felixcloutier.com/x86/>.