Comparing C++ and RUST Performance on the Intel x86 Architecture

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.**

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

I. 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.

II. Background

A. 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.

B. *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.

1) 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.

2) 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.

III. 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.

IV. 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 intended to create a pipeline that takes in a User Program, and outputs the amount of resources used in the execution of this program. Intermediate steps would include getting the Assembly for the x86 Architecture, automating the process of assembling it into an object file with certain tweaks that would aid in tracing what instructions were executed in the real-time run of this program.

As compared to the papers mentioned above, the added value of our approach would include an inclusivity of all hardware, as we are trying to compare the performance differences of the two languages based on the Assembly output from their compilers. Since we want a platform agnostic way to measuring languages (so nothing that can be solved by throwing more hardware at the problem), we trace the execution of this Assembly to find logical differences in how the two programming languages execute the same program. The assembly output for a given CPU architecture remains the constant benchmark throughout, and circumvents bottlenecks such as higher CPU Cores or RAM/Memory. Thus, if one language requires the CPU to execute more Assembly instructions than the other (given the code and logic is same), we can say that the language performs worse in this case.

This approach also reduces our reliance on non-deterministic benchmarks: such as execution time, memory usage at a certain point, or program execution fetching different results due to Operating System multi-threading or concurrency paradigms. The comparison is between two Programming Languages, not between two computers, which makes it crucial to decipher differences between the logic of the compiler's output: instead of the hardware benchmarks.

A. *Initial Approach and Roadmap*

My initial approach included compiling the exact same program on RUST and C++, generating their respective x86 Assembly outputs, and then selecting parameters to trace that would give insights into performance. This would give a benchmark into which programming language executed the same code in a more optimized manner. For a hypothetical example, if the same source code logic:

- RUST compiles a x86 Assemble that requires the CPU to do 5 comparisons and 7 jumps.
- C++ compiles a x86 Assembly that requires the CPU to do 2 comparisons and 4 jumps.
- Then C++ performed better than RUST in this case.

I had initially selected the Assembly instruction Jump: written as "jmp" and refers to Jump to a specific memory location (and variants of it, such as jne, jge, jg, jl, jle), and the instruction Compare: written "cmp" and used to compare the values stored in two memory locations, as the main instructions to trace. Mainly to serve as a proof of concept, I wanted to create an automated script which would add print statements directly into the Assembly before it was assembled into an object file and executed thereafter. Moreover, my initial choices were the RUST compiler, and the GNU C Compiler [17]. The latter was chosen as it remains to be one of the most widely used compilers for C++ development.

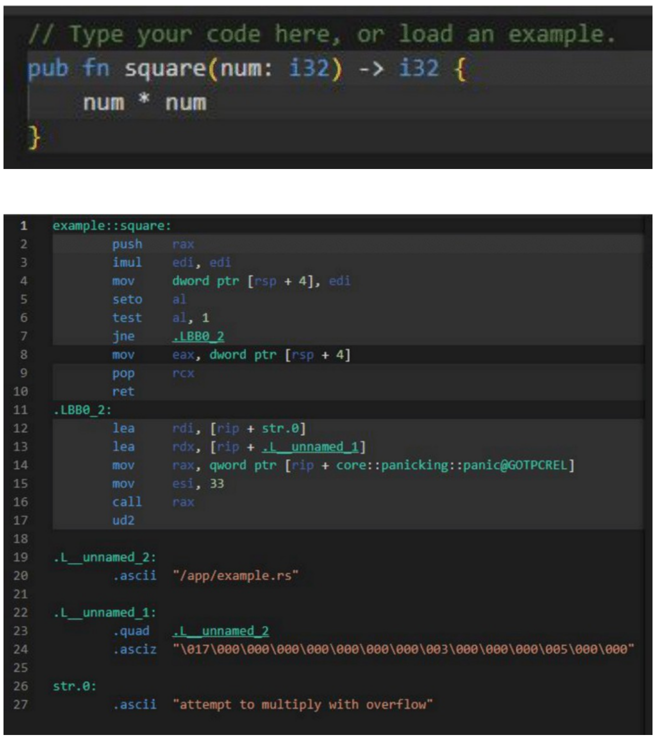
Thus, my goal was to just print on a terminal if a jump or a compare has occurred during the execution of a program, and where has it happened. Also, for variants of Jump that include a comparison (such as Jump if equal to: "je"), I wanted to print details on where has the jump taken place and what comparisons were made. This would provide a trace file with all the jumps and comparisons that have taken place throughout the execution of the program, which could then be analyzed to determine the amount of CPU resources used in total per program.

While the plan for tracing exactly the same programs on RUST and C++ in the Assembly level became the benchmark for this thesis, the Assembly modification approach towards tracing a certain subset of Instructions introduced a lot of variables that soon became an issue.

The first pitfall that I realized soon afterwards was the fact that the GNU GCC and RUST Compiler do not have enough in common to be a valid comparison. As mentioned earlier, the RUST compiler uses the LLVM toolchain to maintain compatibility across platforms, which the GCC does not. Rather, GCC relies on having been compiled for a target platform, which while may provide additional benefits, were not suitable for this test. The RUST Compiler relies on outputting the LLVM Intermediate Representation, which is then picked up by the LLVM toolchain. However, GCC outputs the x86 Assembly directly, thus not creating an equivalent benchmark where performance differences could be accurately measured. Moreover, both of these compilers behave differently with regards to compilation, assembling, and linking of header files: all variables that affect the representation and tracing of the x86 Assembly afterwards.

Essentially, I was unable to find a benchmark for the two languages where given all the variables remain the same, and only the compilers be different, I could trace the Assembly to find the differences in performance. I was able to find some other tools which would give me a comparable benchmark: however, they all fell short in some way.

One example of such a tool was Compiler Explorer: an online tool that lets a user enter any code, and the tool shows the Assembly output of the file along with graphs and visual indicators on which parts of the Assembly represent the code [18].



```rust
// Type your code here, or load an example.
pub fn square(num: i32) -> i32 {
    num * num
}
```

```asm
example::square:
        push    rax
        imul    edi, edi
        mov     dword ptr [rsp + 4], edi
        seto    al
        test    al, 1
        jne     .LBB0_2
        mov     eax, dword ptr [rsp + 4]
        pop     rcx
        ret
.LBB0_2:
        lea     rdi, [rip + str.0]
        lea     rdx, [rip + .L__unnamed_1]
        mov     rax, qword ptr [rip + core::panicking::panic@GOTPCREL]
        mov     esi, 33
        call    rax
        ud2

.L__unnamed_2:
        .ascii  "/app/example.rs"

.L__unnamed_1:
        .quad   .L__unnamed_2
        .asciz  "\017\000\000\000\000\000\000\000\003\000\000\000\005\000\000"

str.0:
        .ascii  "attempt to multiply with overflow"
```

Fig 1: Example Input Code and Output Assembly on the Compiler Explorer Web Tool

However, Compiler Explorer was built with a debugging and educational approach in mind, thus it omits a lot of crucial aspects of Assembly. Namely, low-level system files, or parts of Assembly that do not interact with the code directly itself but are still used in the execution are omitted. Moreover, header files and linked object files for a Programming Language are also omitted in Compiler Explorer, thus not making this a viable approach.

Another approach that I tried was to utilize the "unsafe" aspect of RUST. RUST provides the programmer with a feature that lets them override all of the safety paradigms of RUST, and insert any code of their liking, including x86 Assembly [19]. My approach with creating an equal benchmark was to get the Assembly output from GCC, insert that Assembly output into RUST as an unsafe chunk of code, and compile that (via LLVM) into a form of Assembly which would allow for me to compare between the two programming languages with a proper benchmark. However, this approach had major shortcomings as well, which were realized as I went further with this plan.

```
use std::arch::asm;

let i: u64 = 3;
let o: u64;
unsafe {
    asm!(
        "mov {0}, {1}",
        "add {0}, 5",
        out(reg) o,
        in(reg) i,
    );
}
assert_eq!(o, 8);
```

Fig 2: Example of RUST Inline Assembly

The biggest shortcoming for this approach was not comparing the two Programming Languages based on their compiler, but rather comparing two very different RUST programs (one safe and one unsafe) against each other. Since the comparisons at the end were being made by an output of the RUST compiler for both languages, it was questionable if such an approach was actually comparing programming languages. Moreover, I soon realized that RUST's inline Assembly does not change anything about the Assembly in itself. While initially perceived as a good thing, it was soon realized that this meant the C++ header files for these programs to function were missing and could not be linked at all, at least without tinkering to an extreme degree. RUST links it's own header files to an object file later in the compilation process, which would also clash with C++ header files should we try to brute force the same. Thus, the above approaches were not feasible for the purposes of this project.

I was able to soon find another approach which worked, Clang: a C++ Compiler that depends on LLVM in the exact manner that RUST does. Clang is a very popular C++ compiler, that was also built from the ground up to work on any platform, and outputs LLVM Intermediate Representation which then gets compiled to x86 Assembly.

B. The LLVM Approach: Rustc and Clang compilers

Low Level Virtual Machine, or LLVM, is a toolchain that was introduced to standardize compilation across different platforms. Essentially, The LLVM toolchain requires any compiler to output LLVM Bytecode (or LLVM BC), which is then handled by the LLVM toolchain to compile for any given platform. In between the LLVM IR and the executable, the LLVM toolchain also outputs LLVM Intermediate Representation (LLVM IR), and the x86 Assembly file, each of which can be output into a file for the purposes of debugging as well [15]. The LLVM IR could also be compiled into Assembly for other target platforms, such as ARM or PowerPC. From this point on, the Assembly is assembled into an Object file, header files are linked, and everything is packaged into an executable. This gives the compilers itself the leverage to be platform-agnostic, focus on the logic of the code instead, and levels the playing field for the purposes of this research thesis.

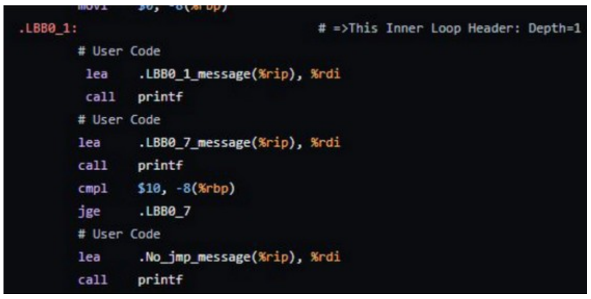
This meant that for comparing RUST and C++, if we used LLVM-compatible compilers, then the x86 Assembly output at the end would all be the same, with the only variable being different the compilers. The approach would give a benchmark for comparing the two languages, keeping every other variable the same, and still focusing on the Assembly output that helps us analyze and differentiate between the logic of the code output between the two compilers. Further research to ensure that Clang was a suitable compiler revealed that it is 1v1 binary compatible with GCC, which meant that everything that has been compiled in GCC will compile in Clang in the exact same manner, including external packages. Debugging tools meant for GCC would also continue to work on Clang (such as GDB or Valgrind), thus helping us compare programs that are already in use as well.

Thus, my new roadmap was to compile the same C++ and RUST programs to x86 Assembly using the LLVM-based compilers, trace the execution of this x86 Assembly, and compare between them. I was able to confirm at this point that for both RUST and C++, any x86 Assembly I output would be able to be further assembled and packaged into a working executable, given that the extra step of linking header files to the object files is taken with RUST. The latter is required as RUST links its own header files to the executable, something not required with C++. However, I believe that this is still a good benchmark for comparing both programming languages, as we are able to compile both to x86 Assembly using the same toolchains, and are able to assemble it into a working executable.

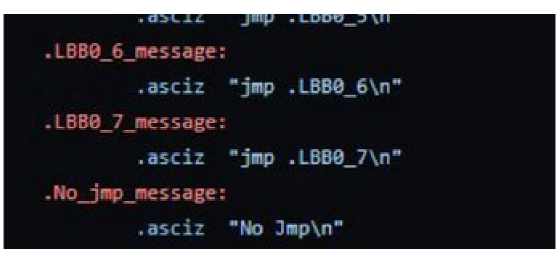
C. Initial Approaches in Tracing Assembly

Upon compiling similar RUST and C++ programs to x86 Assembly, the next step of the thesis was to be able to successfully trace them. There were three main approaches taken, which failed, before I was able to land on to an approach that worked.

My first approach herein was to edit the Assembly file itself (and later do so with an automated script), and add print statement before each Jump and Compare statement. As seen in the snippet below, each line that has the comment "User code" was added manually, and calls a function that pushes an ASCII string into the given register. Both, the ASCII string returning functions and the code interleaved with the main code were user written.



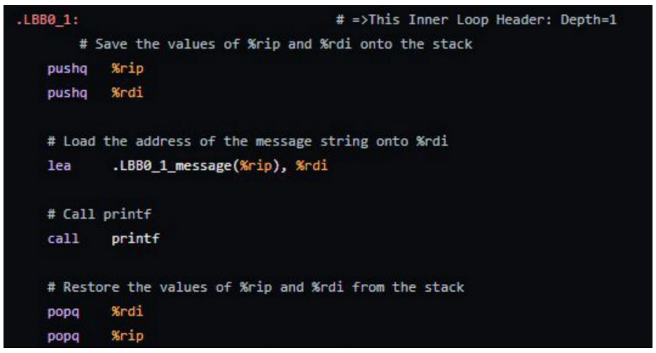
```
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        # User Code
         lea     .LBB0_1_message(%rip), %rdi
         call    printf
        # User Code
        lea     .LBB0_7_message(%rip), %rdi
        call    printf
        cmpl    $10, -8(%rbp)
        jge     .LBB0_7
        # User Code
        lea     .No_jmp_message(%rip), %rdi
        call    printf
```



```
.LBB0_6_message:
        .asciz  "jmp .LBB0_6\n"
.LBB0_7_message:
        .asciz  "jmp .LBB0_7\n"
.No_jmp_message:
        .asciz  "No Jmp\n"
```

Fig 3: Adding Print Statements to Assembly

However, the issue with this approach was using registers that may have already been in use, which would hinder the functioning of this program. The same was evident in most of the attempts to execute this program either resulting in gibberish results, or the program crashing altogether. Thus, I tried to iterate on this approach and save the current register states for the ones that I am using for printing into the current stack:



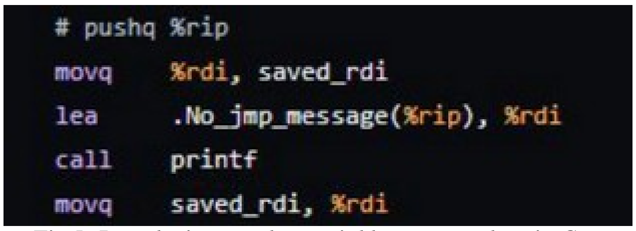
```
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        # Save the values of %rip and %rdi onto the stack
    pushq   %rip
    pushq   %rdi

    # Load the address of the message string onto %rdi
    lea     .LBB0_1_message(%rip), %rdi

    # Call printf
    call    printf

    # Restore the values of %rip and %rdi from the stack
    popq    %rdi
    popq    %rip
```

Fig 4: Saving Register Values into the Current Stack

Here, the values of the registers %rip and %rdi are stored into the Current Stack, then the registers are used to store and print the ASCII string denoting that a jump has taken place, and then the original values of the registers are popped from the stack. It is to be noted that the functions holding the ASCII strings remain the same as above, and thus have not been reproduced in the above figure.

While this approach provided some success in running and tracing programs that did not have many variables, it also crashed often. It is unknown as to why this approach was still non deterministic, but the outputs would often be random or gibberish (sometimes crashing the terminal session). I believe that there may have been an issue with the way I was manipulating the current stack, but regardless it was determined that this approach would not work as well.

A last approach I tried in terms of editing the Assembly itself was to introduce a variable call "saved_rdi" of my own, store the original value of %rdi into this variable instead of the current stack, and output the relevant ASCII string as it was. This approach also did not work, and kept crashing or outputting non-deterministic random outputs, ranging from the ASCII strings in a random order to unknown characters altogether. It was determined that the whole approach with editing Assembly was futile, and would not lead anywhere, and that the issue most likely lies in using the registers to print an ASCII string.



```
    # pushq %rip
    movq    %rdi, saved_rdi
    lea     .No_jmp_message(%rip), %rdi
    call    printf
    movq    saved_rdi, %rdi
```

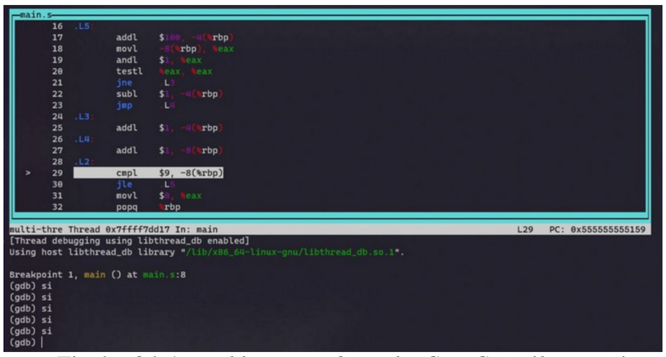
Fig 5: Introducing another variable to save values in Core Registers

It was further realized that editing the Assembly itself would introduce a plethora of other issues that were not considered altogether. It messes up current stacks, registers, and values stored within core registers, which is very memory unsafe and goes against core RUST principles. This results in a tremendous Segmentation Faults which cannot be debugged, and also do not allow for any third person to take my program and start analyzing their own code easily, or at all. I also realized that creating scripts that could reliably change Assembly files in hopes that the same would execute well. Moreover, this approach limits us to the tracing of one file at a time, which does not translate to real life programs that depend on hundreds of different files, ranging from ones that handle different aspects of the same program to system files that are part of the compiler, or the operating system. This approach breaks any interoperability between systems, and has too many moving parts to be considered a reliable benchmark. Most importantly, I realized that this approach could lead to the analysis of false benchmarks, as the print statements add user injected instructions that were not part of the original program, and do not represent the optimization and efficiency of either language.

D. A Debugging Approach to Tracing

The approaches above provided deep insight into some of the things we want to keep in mind while tracing the Assembly output for a given program. Namely, we want to be able to trace across multiple files, without breaking any interoperability or functionality. We also do not want to disturb any source code or compiler output, but rather have a "3rd person view" towards program execution. In essence, we want to just watch how the program executes, and note down the Assembly Instructions that were executed by the CPU during the run-time of the program, we would have obtained a complete trace for this program.

This was the approach that was taken towards the x86 Assembly Tracer for this thesis. Essentially, I used GNU GDB to trace the x86 Assembly that was output from LLVM. Since LLVM is 1v1 Binary Compatible with GCC, the tool-chain can also incorporate GDB Debugging flags during the assembling process into the object file [20]. I was able to successfully obtain x86 Assembly for a given C++ code, and for a given RUST code. Upon assembling this output into an object (and linking header files for RUST), I was also able to pass in GNU GDB Debugging compatibility using the -g flag, similar to GNU GCC.



```
main.s
    .L5:
            addl    $100, -4(%rbp)
            movl    -8(%rbp), %eax
            andl    $1, %eax
            testl   %eax, %eax
            jne     .L3
            subl    $1, -4(%rbp)
            jmp     .L4
    .L3:
            addl    $1, -4(%rbp)
    .L4:
            addl    $1, -8(%rbp)
    .L2:
>           cmpl    $9, -8(%rbp)
            jle     .L5
            movl    $0, %eax
            popq    %rbp
multi-thre Thread 0x7ffff7dd17 In: main                          L29   PC: 0x555555555159
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main.s:8
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) |
```

Fig 6: x86 Assembly output from the C++ Compiler running in GNU GDB. The process is similar for RUST as well.

Through this approach, I was able to successfully run x86 Assembly from any compiler inside of the GNU GDB debugger. I was also able to step inside of any function calls, debug across various assembly files (not constrained to just one file), and record every possible metadata for the execution, including memory, registers, and what registers are the assembly instructions acting on.

To record all the instructions executed during a run of the program, and also the corresponding metadata, I utilized Python's GDB Support. Python is a scripting and programming language used in various applications across data science and software development, and comes with GDB support for debugging purposes as well. Essentially, it is able to latch on to GDB during an active debugging session, and is able to record any data from GDB, including current lines of code/assembly, instructions executed, registers, memory, and countless others. I was able to create a simple script that would record all this data per instruction during the execution of a program, and save it to an external file.

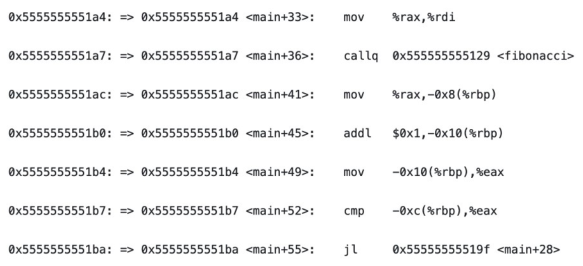
```
import gdb
import time

# Define the file path where you want to save the log
log_file_path = "./gdb_asm.txt"

# Define the maximum time to wait in seconds before
switching to "ni"
max_stuck_time = 10  # Adjust this value as needed

si = 0

max_si = 500

def log_asm_event(event):
    pc = int(gdb.parse_and_eval("$pc"))
    instruction = gdb.execute("x/i " + hex(pc),
to_string=True)
    with open(log_file_path, "a") as log_file:
        log_file.write(f"{hex(pc)}: {instruction}\n")

# Connect the stop event to the log_asm_event function
gdb.events.stop.connect(log_asm_event)

# Initialize timer
start_time = time.time()

# Stepping through the program continuously until it
terminates
while True:
    if si < max_si:
        gdb.execute("si")
        si += 1
    else:
        gdb.execute("ni")

    # Reset timer if execution is not stuck
    start_time = time.time()
    si = 0

    time.sleep(0.001)  # Add a small delay
```

Fig 7: Python Code to Record GDB Data

The above Python file records each Assembly instruction that was executed during the live run-time of the program, along with metadata on memory locations, register states, initial line number of the source file, the assembly instruction, the final line numbers, and the functions invoked (if any). The line numbers provide a crucial way to be able to determine if a Conditional instruction (such as "Jump if greater than") was executed properly or not. Function invocations provide a similar insight as well. Extenuating circumstances (such as Segmentation Fault errors) are also recorded.

The script also automatically keeps Stepping into each assembly instructions, in an attempt to be able to record Assembly instructions from different files as well (including header files). If GDB approaches a certain function that it cannot step into (such as a SysCall), the script reverts to "Next Instruction" as a fallback, before continuing to Step Into others. This way, we are able to trace all the Assembly file instructions for a given program, and record all data nicely into an external file, all without modifying the Assembly file or disturbing the runtime at all. Here's a snippet of the trace output for a given RUST program (in text and as a picture):

```
0x5555555551ba: => 0x5555555551ba <main+55>:       jl
0x55555555519f <main+28>

0x55555555519f: => 0x55555555519f <main+28>:       mov
-0x10(%rbp),%eax

0x5555555551a2: => 0x5555555551a2 <main+31>:       cltq
```



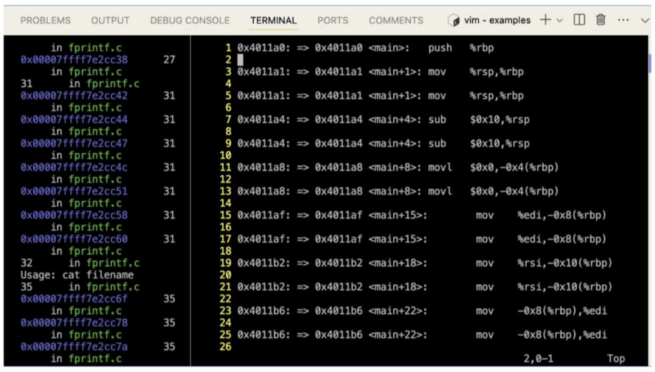
```
0x5555555551a4: => 0x5555555551a4 <main+33>:    mov    %rax,%rdi

0x5555555551a7: => 0x5555555551a7 <main+36>:    callq  0x555555555129 <fibonacci>

0x5555555551ac: => 0x5555555551ac <main+41>:    mov    %rax,-0x8(%rbp)

0x5555555551b0: => 0x5555555551b0 <main+45>:    addl   $0x1,-0x10(%rbp)

0x5555555551b4: => 0x5555555551b4 <main+49>:    mov    -0x10(%rbp),%eax

0x5555555551b7: => 0x5555555551b7 <main+52>:    cmp    -0xc(%rbp),%eax

0x5555555551ba: => 0x5555555551ba <main+55>:    jl     0x55555555519f <main+28>
```

Fig 7: Snippet for a x86 Assembly Trace Output

The first segment of each line shows the memory locations that are currently being manipulated. Line numbers are shown as "<main+33>", the term "main" here being the function currently executed (for line 45 in a function "fibonacci", the term would be "<fibonacci+45>"). We then showcase the Assembly instruction executed, following with the registers manipulated. Lastly, any register values or line numbers are shown as required. The GDB debugger can also jump across multiple Assembly files, including some system calls, and all are recorded in the trace file (including function and file names as applicable).

Thus, we are able to get an Assembly Trace for a given x86 Assembly Code executed by Clang: the C++ Compiler, or RUSTC: The RUST Compiler.

A last thing to note: I am also collecting every Assembly instruction in this main trace file now, instead of just Jumps and Compares. This is because with GDB, I can easily do so, and provide a more in-depth analysis of the amount of CPU resources used per program. Since GDB handles Multi-threaded programs and Parallel Processing very well, I am also able to get Trace outputs for programing involving the same.



```
    in fprintf.c
0x00007ffff7e2cc38      27
    in fprintf.c
31      in fprintf.c
0x00007ffff7e2cc42      31
    in fprintf.c
0x00007ffff7e2cc44      31
    in fprintf.c
0x00007ffff7e2cc47      31
    in fprintf.c
0x00007ffff7e2cc4c      31
    in fprintf.c
0x00007ffff7e2cc51      31
    in fprintf.c
0x00007ffff7e2cc58      31
    in fprintf.c
0x00007ffff7e2cc60      31
    in fprintf.c
32      in fprintf.c
Usage: cat filename
35      in fprintf.c
0x00007ffff7e2cc6f      35
    in fprintf.c
0x00007ffff7e2cc78      35
    in fprintf.c
0x00007ffff7e2cc7a      35
    in fprintf.c
```

```
0x4011a0: => 0x4011a0 <main>:   push   %rbp

0x4011a1: => 0x4011a1 <main+1>: mov    %rsp,%rbp

0x4011a1: => 0x4011a1 <main+1>: mov    %rsp,%rbp

0x4011a4: => 0x4011a4 <main+4>: sub    $0x10,%rsp

0x4011a4: => 0x4011a4 <main+4>: sub    $0x10,%rsp

0x4011a8: => 0x4011a8 <main+8>: movl   $0x0,-0x4(%rbp)

0x4011a8: => 0x4011a8 <main+8>: movl   $0x0,-0x4(%rbp)

0x4011af: => 0x4011af <main+15>:        mov    %edi,-0x8(%rbp)

0x4011af: => 0x4011af <main+15>:        mov    %edi,-0x8(%rbp)

0x4011b2: => 0x4011b2 <main+18>:        mov    %rsi,-0x10(%rbp)

0x4011b2: => 0x4011b2 <main+18>:        mov    %rsi,-0x10(%rbp)

0x4011b6: => 0x4011b6 <main+22>:        mov    -0x8(%rbp),%edi

0x4011b6: => 0x4011b6 <main+22>:        mov    -0x8(%rbp),%edi

                                                  2,0-1         Top
```

Fig 8: GDB Tracing a RUST Assembly File on the left, and the Vim text editor showcasing the active trace file being built on the right.

E. Analyzing the Trace Files

For the second part of the methodology, I have obtained a plethora of trace files across various programs on C++ and RUST, and we have to now analyze the same. My main approach here was to learn how many CPU Clock Cycles were executed per Assembly Instruction. To understand what a CPU Clock Cycle is, we look at the Fetch, Execute, and Decode cycle.

Every CPU does a Fetch→Execute→Decode Cycle, or more deeply: fetches an Assembly Instruction, Decodes what the instruction wants to do, Executes that instruction, and (optionally) performs a Memory Read/Write as part of the execution. This cycle is continued indefinitely, until either the user sends an INTERRUPT System Call, or the program terminates. In simplistic terms, we can say that one clock cycle is one whole run of a Fetch→Execute→Decode Cycle.

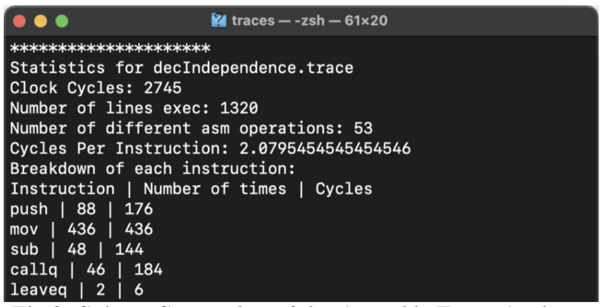
However, not all Assembly Instructions are created equal. Different instructions may use different amounts of clock cycles, and this is where the differentiation factor comes in between RUST and C++ comparisons. On an Assembly level, we can take an instruction, "jne" (Jump if not equal to), as an arbitrary example. This instruction has a comparison part ("Do x *if not equal to*), and a Jump part. Thus, this instruction actually uses 4 clock cycles on an average.

The values of the number of Clock Cycles per Assembly Instruction for the purposes of this thesis were taken from Professor Agner Fog's "Instruction Tables", a comprehensive guide of every single x86 Instruction, and the number of clock cycles they each use on a plethora of different CPUs. These clock cycles remain roughly the same across the many CPUs, unless we get to some super specific instructions that pertain directly to system calls, which becomes outside the scope of this thesis as those system calls cannot be traced using the methods described herein. Hence, for all the common instructions that we may come across, we have a stable benchmark of the number of clock cycles that any given x86 Assembly instruction uses.

At this point in the study, we have the same programs on RUST and C++, we have their respective compiled x86 Assembly files, and we have extracted the Assembly trace files by executing them and recording the instructions on a separate file. We also have a comprehensive list of the number of Clock Cycles for a given instruction on the x86 Architecture, and can calculate the number of Clock Cycles executed per program.

If for a given program, one language uses lesser clock cycles than the other to achieve the same output, we could say that the former program is the better performing one. To make this an even comparison: I also added additional metrics, such as number of lines executed, cycles per instruction (number of total clock cycles divided by number of lines executed, which gives us a ratio of the number of clock cycles executed per instruction), number of different Assembly instructions executed for the program, and a breakdown of each instruction (with how many clock cycles that instruction used, and how many times it was executed).

Through a Python program which takes an Assembly Trace File described above as an input, a user can get all these metrics as output. The script also takes into account other run-time factors for each instruction (for example: checking if a conditional jump has occurred or not).



```
*********************
Statistics for decIndependence.trace
Clock Cycles: 2745
Number of lines exec: 1320
Number of different asm operations: 53
Cycles Per Instruction: 2.0795454545454546
Breakdown of each instruction:
Instruction | Number of times | Cycles
push | 88 | 176
mov | 436 | 436
sub | 48 | 144
callq | 46 | 184
leaveq | 2 | 6
```

Fig 9: Snippet Screenshot of the Assembly Trace Analyzer

This completes the pipeline from User Code in C++ or RUST, to Compiling to Assembly, to Linking Header Files and Passing Debugging Flags, to Tracing the Assembly, to finally analyzing the Assembly Traces and compare the programming languages.

V. Results and Discussions

For the analysis of RUST and C++, I ran the tracer analyzer toolchain (herein referred to as *tra86*), on the same program across RUST and C++, with minor tweaks for each runtime. The programs run can be divided into two main segments:

- Linux System Programs and Commands
- Self-written Programs to test constraints.

The first segment deals with programs that are bundled as part of the Linux Command Line, such as "ls" (list directories), or "cat" (read and print contents of a file into the command line). These are simple programs, yet crucial ones used in everyday life. Their vast availability and simplicity makes them one of the best things to target, as these tiny programs are what collectively make up the Operating System. In the given time constraints for this research paper, comparing Linux system programs would become the closest analog to comparing what a full Operating System written on the two programs may look like.

The second segment deals with programs that communicate with the Internet. These are mainly scripts or small Command Line Utilities, ranging all the way from simple ones like Ping, to complex ones such as WGET (a tool that helps a user download resources from a URL). While not bundled as part of the Operating System, these scripts can help decipher more real-life usage data for the languages. Some of the scripts (such as CURL) have been simplified to allow for the scope of this project as well.

As a side note: the following traces were obtained from a GitHub Codespace, a cloud-based virtual machine running Ubuntu 22.04 on the Intel Xeon Platform (a x86 CPU), and traces were also obtained from a Desktop PC running on an Intel Core i5 CPU, and a MacBook Pro running on an Intel Core i3 CPU. Since we are dealing with the Assembly outputs of the compilers and tracing that, hardware specifications should not make a difference at all, which was proven using these different configurations. The GDB Debugger on every platform ran the program in the exact same manner, and there were absolutely no differences between hardwares, which ensures that this test is hardware agnostic in real life conditions as well.

A. Linux Command Line Utilities

For the first part of the programs, I chose the Command Line Utilities "ls", "cat", and "cp". The main files for C++ were taken from *Dirent* [22], a lightweight implementation of these functions made for use on embedded systems. RUST implementations of the given functions were a direct implementation from the C++ versions as part of *Dirent*, with absolutely zero change to the logic of the code. The RUST Versions of *Dirent* code can be found on the *tra86* repository, as part of this Honors Thesis as well.

The graphs shown below are for Cumulative Clock Cycles, and the Clock Cycles Per Instruction respectively, for the given programs with different arguments.

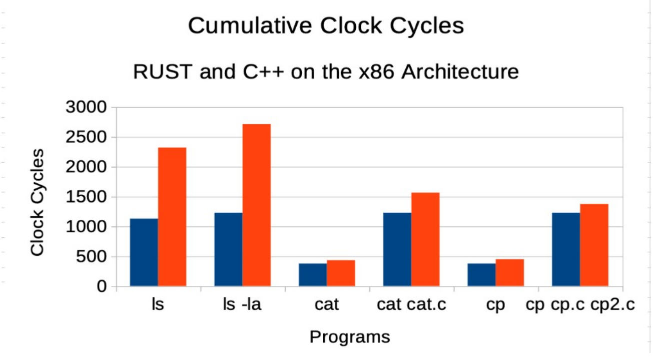


Fig 10: Cumulative Clock Cycles for different Command Line Programs (BLUE: Rust, RED: C++)

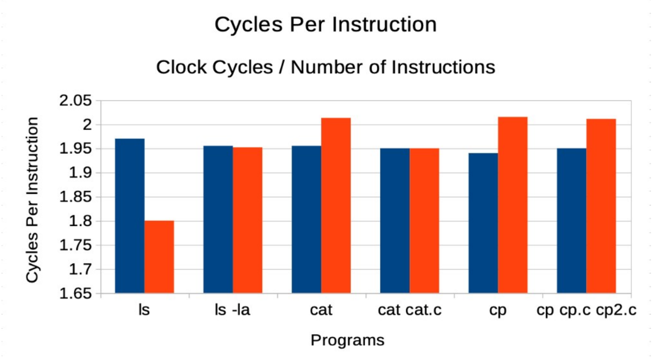


Fig 11: Clock Cycles Per Instruction for different Command Line Programs (BLUE: Rust, RED: C++)

Across the board, RUST used less raw CPU power (in the form of cumulative clock cycles) to perform and achieve the same tasks. However, both RUST and C++ used roughly an equal amount of Cycles Per Instruction, with RUST fairing a bit worse for more object-oriented tasks (such as ls), as well as executing more lines at times due to the higher security paradigms. Thus, we could at least consider RUST at par with C++ in terms of Cycles Per Instruction, noting that the language provides higher security and easier debugging.

Another experiment that I tried was to use one program, ls, and increase the complexity of the program at every point. I ran ls as is, ls with the -a flag (shows hidden files), ls with the -la flag (shows hidden files and extra information on read/write and ownership information per file), and -la -r flags (recursively displays all files). The following graph was the result of increasing complexity on the same program:

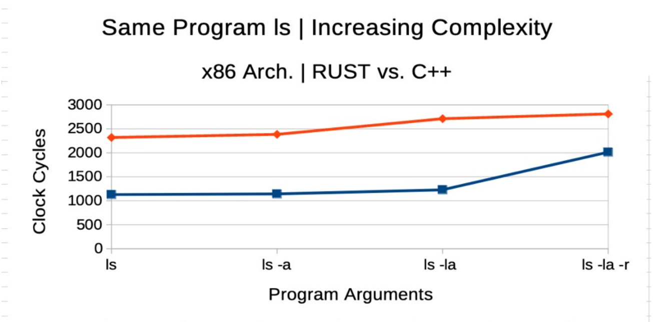


Fig 12: Increasing Complexity on the ls Command Line Program (BLUE: Rust, RED: C++)

While the complexity of the RUST and C++ program increase accordingly, it can be seen that for the same program with the same logic, C++ uses more CPU Clock Cycles than RUST throughout all the possible combinations. Thus, while RUST may use more Cycles Per Instructions in some cases, C++ uses overall higher amounts of Clock Cycles, albeit not by much. This makes the performance differences between RUST and C++ almost equivalent at worst, and RUST's raw performance better in an ideal case.

B. Self-written Programs to Test Constraints

The second group of programs consisted of self-written programs, ranging mainly in the realm of Data Structures and Algorithms. These programs were written as a means to test the Performance Metrics of RUST and C++ given any overhead that RUST may have owning to the Code Security features that it exhibits (inheritance, ownership, and protection).

A set of 5 different algorithms were written on RUST and C++, following the exact same logic and computation power. We coded the Fibonacci Sequence, Reversal of a Linked List, Breadth First Search (BFS), Depth First Search (DFS), Bubble Sort on an arbitrary array, and Merge Sort on an Arbitrary array. The implementations of the algorithms on both languages can also be seen under the *tra86* repository.

The graphs shown below are respectively the Cumulative Clock Cycles, and the Cycles Per Instruction for running the following programs on the two languages, keeping every other aspect the same.

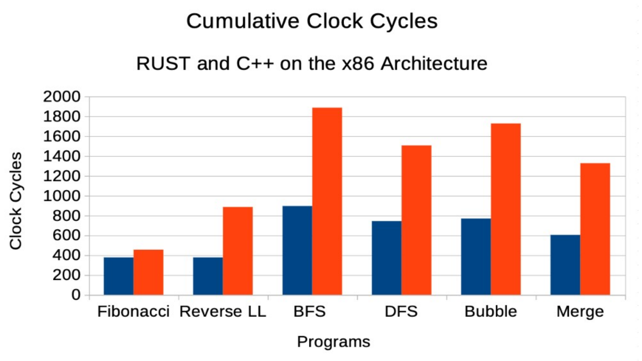


Fig 12: Cumulative Clock Cycles for self-written Algorithms (BLUE: Rust, RED: C++)

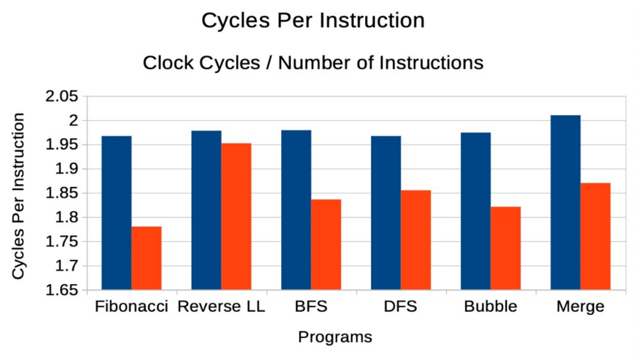


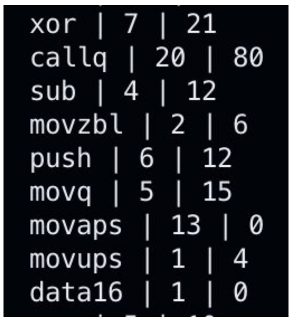
Fig 13: Clock Cycles Per Instruction for self-written Algorithms (BLUE: Rust, RED: C++)

It is for self-written programs that RUST still uses lesser Cumulative Clock Cycles, but we see a different outcome for Cycles Per Instruction. The results were taken over multiple runs of the programs on each language as well (about 200 runs), and yielded the same results throughout.

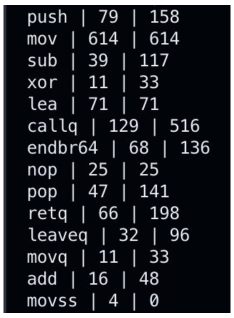
From the Cumulative Clock Cycles graph, it can still be inferred that RUST uses lower raw performance overall, and conducts the same computation that both languages had to. Moreover, RUST does so while providing the various code safety features discussed throughout the paper, which are not available in C++. Thus, RUST is still performing better in this case.

The results get interesting when we analyze the Cycles Per Instruction. These subset of programs were the only ones that resulted in a higher Cycles Per Instruction for RUST than for C++, i.e. the notion that the CPU had to use more resources per instruction for a RUST program. It is interesting to note that despite the higher amount of resources used by the CPU, the total resources used are still lesser than C++, still making RUST more performant than C++.

I speculate that one of the results for this effect is the fact that RUST uses more obscure x86 instructions than C++ for compilation, which result in more clock cycles used per instruction. The Instruction Tables I am using [3] contain many variations of the same instructions with different operands, as well as contain complex instructions that are combinations of multiple simple instructions (such as *mov* and *movq2d2*), which may use a higher number of clock cycles per instruction.



```
xor | 7 | 21
callq | 20 | 80
sub | 4 | 12
movzbl | 2 | 6
push | 6 | 12
movq | 5 | 15
movaps | 13 | 0
movups | 1 | 4
data16 | 1 | 0
```



```
push | 79 | 158
mov | 614 | 614
sub | 39 | 117
xor | 11 | 33
lea | 71 | 71
callq | 129 | 516
endbr64 | 68 | 136
nop | 25 | 25
pop | 47 | 141
retq | 66 | 198
leaveq | 32 | 96
movq | 11 | 33
add | 16 | 48
movss | 4 | 0
```

Fig 14: Example snippets of RUST (left) and C++ (right) analyzer outputs. RUST uses more complex forms of the "mov" instruction, thus maybe resulting in more Cycles Per Instruction.

Another limitation that I speculate in the following research can limitations that surround the GNU GDB debugger, the software that forms the basis of the tracer. GDB can only trace and record Assembly Instructions if all the files that it has to trace have been compiled with debugging flags (something that can be only done with availability of all source code). There have been certain header files that GDB has been unable to trace, due to the fact that GDB cannot read the file itself since it was not compiled for debugging purposes by the original author.

While GDB was able to record assembly instructions for all header files in Group 1 (which were mainly C-based Operating System files facilitating I/O, memory read/write, etc.), GDB was not able to trace many of the RUST header files that were invoked during the execution of the RUST version of the algorithms I wrote. On the other hand, while GDB was unable to open some C++ header files as well, it was still able to trace the instructions executed while that header file was active. This may mean that RUST results yield differently because a lesser number of lines were able to be traced, thus resulting in a skewed Cycles Per Instructions metric.

Regardless of the Cycles Per Instruction Metric, it can still be concluded that RUST uses less raw resources based on the Cumulative Clock Cycles graph.

VI. Conclusions and Future Scope

Through a comprehensive analysis between RUST and C++ through an Assembly Trace and Analysis, it can be inferred that RUST performed better than C++ across the board based on the cumulative number of CPU clock cycles used per program. In fact, RUST is about 40% faster than C++ for the same program executed on both languages, conducted on tests that have been hardware agnostic and at the Assembly level.

I do believe that more robust results can be extracted using the *tra86* toolchain, if all the header files and supplemental files are also compiled with debugging flags such that GDB can read them and proper instructions be traced. This may eliminate any inconsistencies and speculations regarding the Cycles Per Instruction metric for self-written code on RUST.

I plan to conduct further studies on this subject matter beyond the Honors Thesis, especially with creating Virtual Environments from the ground up wherein GDB has access to every file. Moreover, I also plan to conduct this study with a wider range of programs: including programs that involve networking, asynchronous processes, multithreading, and locks/mutex. Moreover, the idea of tracing Assembly files can be ported to other CPU architectures as well, which I plan to do and test RUST and C++ on (specifically ARM).

I also believe that the *tra86* toolchain can be improved upon by gathering more datapoints and having more transparent access to every single file that it needs to trace.

Despite the limitations of the *tra86* toolchain, the results of the *tra86* toolchain still lean heavily towards RUST performing much better than C++ across the board, and that the *tra86* toolchain will prove to be a valuable tool for other developers to also test their C++ and RUST programs and find performance differences on the Assembly Level.

VII. Acknowledgements

I would like to hereby acknowledge Professor Meng-Chieh Chiu for his support and guidance throughout the Honors Thesis, as well as Professor Timothy Richards for acting as my supplemental advisor. I would also like to thank Professor Philip Sebastian Thomas, the Honors Program Director for Computer Science, to guide me through the process of writing an Honors Thesis and guiding through the process.

I would like to thank the Integrated Concentration in STEM program at UMass Amherst, for their invaluable guidance through the process of writing a thesis, communicating the results, as well as for invaluable opportunities such as talking about this research work at the Massachusetts Undergraduate Reseachers' Conference, and the iCons Senior Expo. Specifically, I would like to thank Professor Scott Auerbach for his guidance and input in the project and it's communications to a wide range of audiences.

Lastly, I would like to thank Anvitha Ramachandran, undergraduate Computer Science and Computer Engineering major at UMass Amherst, for their incredibly valuable insights on low-level architectures, as well as honing in on the methodology to trace Assembly files in a manner which is comprehensive, and encompasses all edge cases and possibilities.

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.

[16] Felix Cloutier. "X86 and Amd64 Instruction Reference," May 2023. https://www.felixcloutier.com/x86/.

[17] "GCC, the GNU Compiler Collection - GNU Project," accessed May 2, 2024, https://gcc.gnu.org/.

[18] Matt Godbolt, *Compiler Explorer*, Web/UNIX (Illinois, 2023), https://godbolt.org.

[19] "Inline Assembly - Rust By Example," accessed May 2, 2024, https://doc.rust-lang.org/rust-by-example/unsafe/asm.html.

[20] "GDB: The GNU Project Debugger," accessed May 2, 2024, https://www.sourceware.org/gdb/.

[21] "GDB Support," *Python Developer's Guide*, accessed May 2, 2024, https://devguide.python.org/development-tools/gdb/.

[22] Toni Rönkkö, *Dirent: Programming Interface for Retrieving Information about Files and Directories in C and C++ Languages.*, n.d., https://github.com/tronkko/dirent.

[23] Kushagra Srivastava, "LLVM Compilation Performance Metrics," accessed May 3, 2024, https://tra86.skushagra.com/.