Comparing RUST and C++ Performance Metrics

Kushagra Srivastava

University of Massachusetts Amherst

Hons. 499T Proposal | Faculty Sponsor: Prof. Meng-Chieh Chiu

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 at various stages of computation: from low-level systems that communicate directly with the hardware, to high-level code such as Artificial Intelligence or Webpages on the Internet. Decades of contributions to all these different levels of computation have rendered the systems that we use daily. My research project attempts to dive into developing an intricate understanding of the nature of Computing, specifically lower-level code in the context of different Programming Languages.

The following Research Proposal attempts to compare between two 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++ was an evolution of C which was popularized during the 80s and 90s, and RUST which was released in 2013. As a result, 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 the risk of breaking the system.

I want to compare Performance Differences between the two languages 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. Majority of the low-level code today is running on C++, mainly due to the decades of contributions that have gone into developing such systems. Should an undertaking be made to shift most of the code from C++ to RUST, this project would give developers an adequate, hardware-agnostic idea of the performance benefits that RUST provides for their specific use-case.

1.1: Scientific Problem

The following Research Proposal draws inspiration from the paper, "Real-Time Program-Specific Phase Change Detection for Java Programs", authored by Prof. Meng-Chieh Chiu, Prof. Eliot Moss, and Prof. Benjamin Marlin. I use similar methodology described in the paper to aid in the comparison of performance metrics between RUST and C++, in a hardware-agnostic manner.

During the compilation of a program in each programming language, the program goes through different phases, such as: 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 consists of a set of specific instructions known to the CPU, and any program in the world can be written as a combination of these instructions. The CPU then goes sequentially through these instructions, and uses a certain amount of resources to execute each of them. These resources, also known as Clock Cycles, 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 different programming languages, we can notice how each of them generate their respective Assembly files, which is then executed by the CPU. 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 respective logic. For example, if we write the same program on RUST and C++, and the RUST version of the program has 5 lesser instructions in the Assembly for the CPU to execute while achieving the same output, we can say that RUST performs better than C++ in this specific case. Executing and analyzing multiple files such as this may give a comprehensive view of how optimized each Programming Language is. Moreover, this approach also circumvents potential issues such as hardware requirements, operating systems, and configuration of the machine or the state the program was run on, and instead focuses solely on the logic of each of the programming languages in use.

I plan to conduct this research on the x86 Architecture, popularized by Intel in consumer machines. The x86 Architecture uses a version of Assembly that consists of a set of 81 instructions, and 6 registers. This architecture has seen immense development in both C++ and RUST, and the hardware has matured to a stable point. Unlike novel architectures as ARM, I believe that the x86 Architecture would minimize potential setbacks to conduct such an experiment. For example, Apple's ARM Architecture, popularized in the M1 Processors and above, rely on a lot of x86 Architecture virtualization. That is, these chips run x86 programs in a software-defined x86 environment, thus potentially leading to false numbers. However, I believe that the approach and tooling developed as part of this research can be extended to other architectures as well, such as RISC-V or ARM, given certain precautions are taken.

1.2: 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. This is 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 servers and infrastructure 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. 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, thus deterring a lot of novice programmers into entering this field. Moreover, being an incremental upgrade to an already old language, C++ can sometimes not be the most efficient language out there, especially when it comes to compiling big programs, such as the Linux Kernel.

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. However, being a recent programming language, RUST has not seen as widespread adoption as anticipated initially.

I believe that these shortcomings can be chalked up to a lack of knowledge on how performant RUST can be, as compared to C++. Companies and Developers mainly fund C++ development, because the cost of refactoring systems to an entirely new programming language is a massive undertaking, and there does not exist enough knowledge on the Return on Investment that such an undertaking would provide. Thus, showcasing Performance Metrics between the two languages may give an overview on the Return on Investment for shifting to RUST from C++, 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[1], 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

[1] https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/

than the electricity used in about 119 countries of the world[2], 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.

RUST would enable the same infrastructure to server exactly the bandwidth it is serving currently, while automatically providing back electricity worth of powering such a high number of countries. Moreover, the same infrastructure can also be utilized into serving an even higher bandwidth without putting new hardware into place, thus providing immense future-proofing and reliability.

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, which is a result of the time that they were created in. Development of the language started at a time when hardware architectures were not standardized. As a result, C behaves differently on different kernels and hardware, even today. Moreover, there are currently tremendous amounts of versions of C in circulation, which can become confusing for a novice in the field. On the other hand, RUST was developed from day 1 as an open-source project, placing proper standardization across platforms. RUST behaves the same, on every single piece of hardware, which lowers the barrier of entry incredibly. Paired with incredible documentation, and a unified compilation process on every system, RUST makes low level development significantly easier to novices and professionals alike.

[2] https://en.wikipedia.org/wiki/List_of_countries_by_electricity_consumption

Literature Review

The key literature review for this project included Documentation for the RUST and C++ Compilers, which would give me an insight 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. Lastly, I also consulted documentation on the x86 Architecture which I am using to conduct my research, and on technologies that are used in the RUST and C++ compilers, such as the GCC and LLVM toolchains.

Professor Chiu's paper, "Real-Time Program-Specific Phase Change Detection and Prediction[3]," sets to explore the Phase Changes for the compilation of Programs written in 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. 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 phase consists of (which would tell how similar or different a given phase is). These metrics would then be analyzed using the Gaussian Mixture Model (GMM): which is a "probabilistic model generalizing k-means clustering to incorporate information about the covariance structure of the clusters." (Chiu et al., 2016). This approach gives a deep insight into the methodology that is used to derive performance metrics for a given programming language, and is done in a hardware-agnostic manner. Moreover, this project by Professor Chiu also involved tracing Python Bytecode, which is a similar low-level representation of Python Programs. However, Professor Chiu's paper deals with only Java, Python, C, and with Compilation Metrics. The project does not make comparisons between the programming languages, but gives insight into the workings of each of the languages mentioned above. Regardless, Professor Chiu's Python Bytecode Tracer became the

[3] https://scholarworks.umass.edu/cgi/viewcontent.cgi?article=2422&context=dissertations_2

groundwork for my x86 Assembly Tracer to compare Performance Metrics between RUST and C++, a product which I will be expanding upon in 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[4]" by Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti. The 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 that 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. RUST, notably has a much lower entry barrier, with its optimized object-oriented approach to programming that, according to the paper, is missing in both C and Fortran. The paper mentions the benefits RUST has over C and Fortran, running equally well on every hardware and handling concurrency and parallel processing much better than C/Fortran. Moreover, the paper also lauds the ownership functionality of RUST, which prevents edge cases such as Race Conditions, and ensures that multithreading does not pose any risks to program logic or data.

However, the case study focuses mainly on High Performance Computing, and determines success if the Performance between RUST and C/Fortran is roughly equal, and the ease of code is easier. Crucially, this approach misses the mark on conveying performance differences to everyday developers, who are writing code that functions on consumer machines and solve everyday tasks. 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 to C; keeping the barrier to low-level systems programming for novices still alive.

[4] https://arxiv.org/abs/2107.11912

Another similar paper that I approached during my literature review was "RUST vs. C++, a Battle of Speed and Efficiency[5]" by Vincent Ng. 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 two programs, as RUST's ownership functionalities do not provide a direct comparison to C++. Moreover, just like the research project mentioned above, this method also relies heavily on the hardware configuration of the user, leaving very little room for executing nuanced programs that may use parallel processing, multithreading, or multicore execution. This approach also relies on Memory and CPU usage, metrics which can be easily circumvented by using more powerful hardware.

Thus, it can be noticed that while research in the topic of measuring RUST and C++'s performance differences are plentiful, they are all very reliant on the hardware configuration of the system that they were tested on, and not on the logic of the respective Compilers. By choosing to trace the Assembly executed by the respective Compilers, we can derive performance metrics between the two languages based on their inherent logic, and not on the capabilities of the hardware. The only reason I plan to use the x86 Architecture is to have a standardized platform for this comparison to take place, but the approach I plan to employ does not rely on measuring Memory or CPU utilization, as these metrics do not provide a comprehensive picture suited to developers from all backgrounds.

I also consulted documentation on the RUST and C++ Compilers, to understand how compilation takes place. For RUST, I referred to the RUST Compiler Development Guide[6], which provides extensive manuals into downloading the compiler, making changes to the source code, and testing them on a local

[5] https://www.researchgate.net/publication/370735143_Rust_vs_C_a_Battle_of_Speed_and_Efficiency
[6] https://rustc-dev-guide.rust-lang.org/

environment on any given computer. The RUST Compiler Guide also provides deep insight into the Low Level Virtual Machine (LLVM) architecture that is used to output the x86 Assembly, and how the whole pipeline from user-code to computer-program 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 Bytecode, and finally to Assembly.

On the other hand, the Clang Compiler User's Manual[7] 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 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[8]", which also goes over instructions on installing LLVM and executing your own LLVM IR on the platform. Thus, while my focus remains on

[7] https://clang.llvm.org/docs/UsersManual.html
[8] https://llvm.org/docs/

the x86 Architecture, I believe that using Clang, RUST, and LLVM may enable to conduct this research study in different architectures, such as ARM or RISC-V.

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 Bytecode, 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[9]", 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 the 81 instructions they can execute, and the purposes of the 6 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 thus far has also been the "x86 and amd64 Instruction Reference[10]" by Felix Cloutier. 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. Since I am trying to create a tool that traces the execution of Assembly, i.e. records what lines of Assembly are being executed at each point, I believe that this guide would provide a comprehensive guide on how much resources I have spent in executing the same program on either programming language, which is what lies at the crux of this Research Endeavour.

[9] https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[10] https://www.felixcloutier.com/x86/

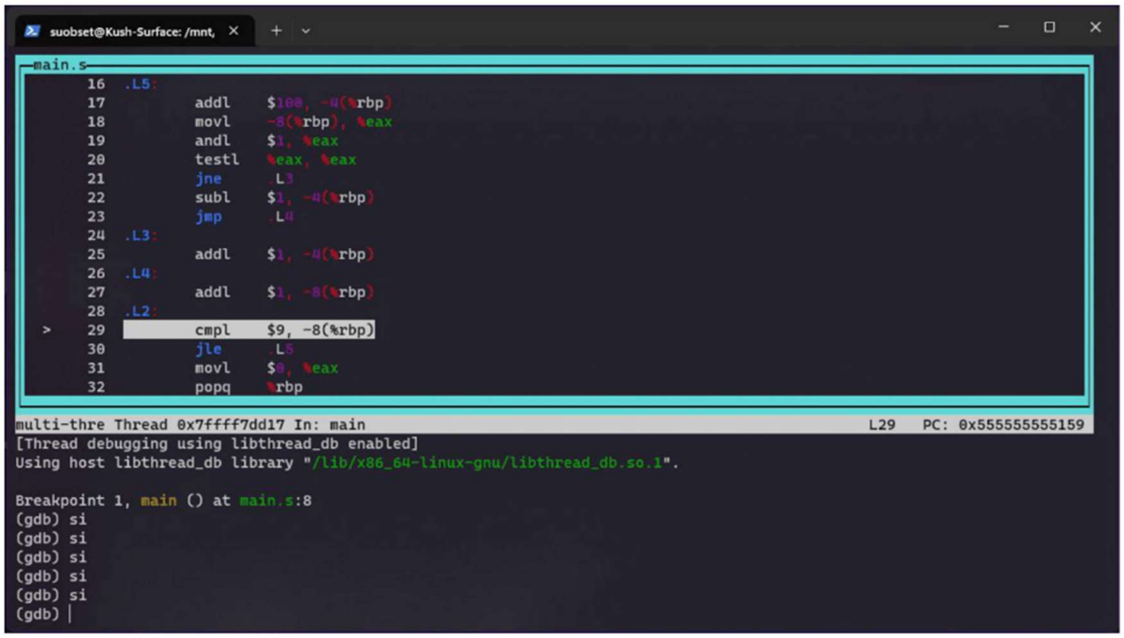
Research Methodology

The main intent behind this research project is to compare performance metrics between RUST and C++ in a hardware-independent manner. Specifically, I want to focus on the Assembly output that the two languages give, for the same exact programs as input. We then trace this Assembly file during execution, that is we create a log of what lines of this file were executed by the CPU, and what instructions were executed. When we have this log, we can retroactively calculate the amount of resources used by the CPU for achieving the desired output, i.e. successful execution of the code.

For example, let us say that we write a program on RUST or C++. We then get the Assembly from their respective Compilers, and we start executing it. During execution, we create a log (through software) of the instructions that are being executed in the Assembly, and the line numbers to keep track of loops. At the end of this run-through, we can analyze this log for the different programming languages. Since the Assembly is a standardized set of instructions dependent only on the CPU Architecture, we can calculate how many Clock Cycles were spent by the CPU to execute the same program on both languages. Thus, Performance Metrics for the two languages can be derived, as long as we do so on the same CPU Architecture.

The following research is divided into two main segments. The first part is building tracing tools for x86 Assembly that would create logs of Assembly execution. This part was achieved during the 499Y Honors Thesis in Fall 2023. I was able to create a “x86 Assembly Tracer”, which utilizes GNU GDB, a debugging platform, and Python, a high-level programming language. Essentially, GNU GDB can debug Assembly files, if we make sure the compiler provides debugging information in the metadata of the code. As a debugger, GNU GDB can derive execution information, such as the Assembly instruction currently being executed, memory status of the Assembly Stack, and so on. I was able to use Python’s

GDB Support, as outlined in the Python Developer's Guide[11], to act as an Application Programming Interface and create a log of different Assembly instructions.



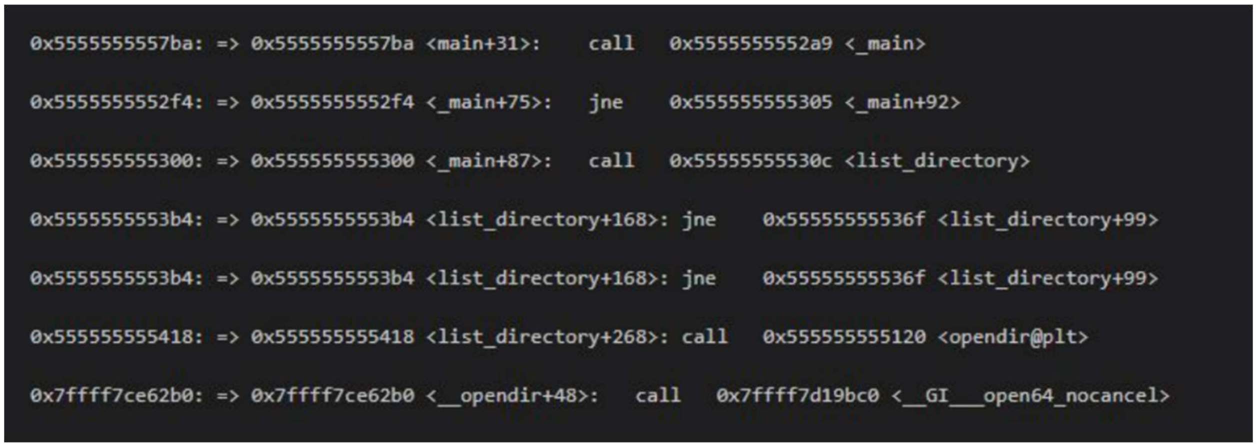
```
  .L5:
          addl    $100, -4(%rbp)
          movl    -8(%rbp), %eax
          andl    $1, %eax
          testl   %eax, %eax
          jne     .L3
          subl    $1, -4(%rbp)
          jmp     .L4
  .L3:
          addl    $1, -4(%rbp)
  .L4:
          addl    $1, -8(%rbp)
  .L2:
> 
          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) |
```

Figure 1: GDB Running Example x86 Assembly

There were failed approaches over the semester, which included making changes to the Compiler Source Code, making Changes to the outputted Assembly to output Print Statements as logs, and using online tools such as Compiler Explorer[12]. However, none of these approaches would yield positive results, as making any changes to the Compilers or Assembly outputs would render the research project not too useful. We want to measure performance differences for this tooling in the real-world, as it is provided to other people, which crucially means we cannot make any changes to the tooling either. Thus, using GDB would provide a "3rd Person View" into program execution, without interfering with any parameters that we are trying to derive performance metrics for. The combination of GDB and Python renders a single, standalone, compact utility that can be used to derive logs for Assembly

[11] https://devguide.python.org/development-tools/gdb/
[12] https://godbolt.org

execution, and be used by other developers to do similar tracing for their own code, as long as it outputs x86 Assembly.



```
0x5555555557ba: => 0x5555555557ba <main+31>:    call   0x5555555552a9 <_main>
0x5555555552f4: => 0x5555555552f4 <_main+75>:   jne    0x555555555305 <_main+92>
0x555555555300: => 0x555555555300 <_main+87>:   call   0x55555555530c <list_directory>
0x5555555553b4: => 0x5555555553b4 <list_directory+168>: jne    0x55555555536f <list_directory+99>
0x5555555553b4: => 0x5555555553b4 <list_directory+168>: jne    0x55555555536f <list_directory+99>
0x555555555418: => 0x555555555418 <list_directory+268>: call   0x555555555120 <opendir@plt>
0x7ffff7ce62b0: => 0x7ffff7ce62b0 <__opendir+48>:   call   0x7ffff7d19bc0 <__GI___open64_nocancel>
```

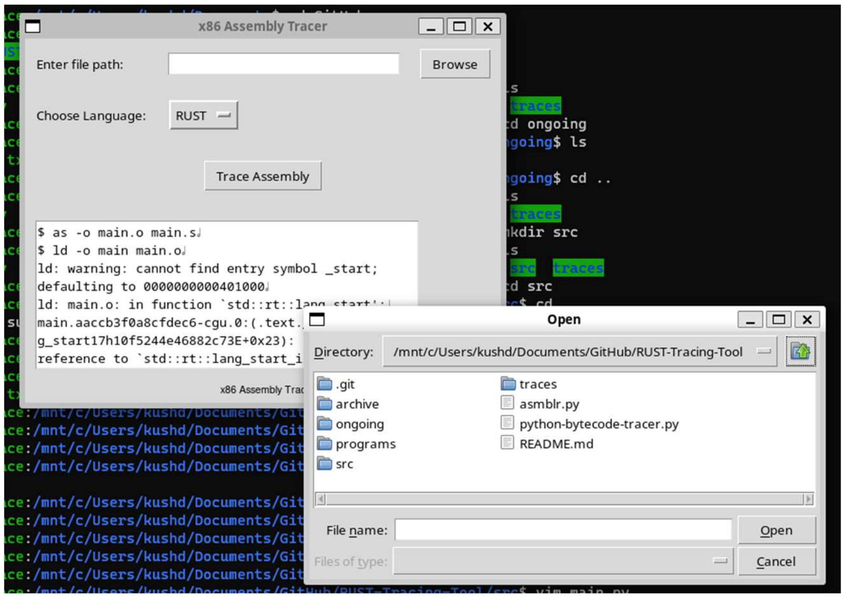
Figure 2: Assembly Trace on the "ls" UNIX Command

The second phase of the research includes running various programs on C++ and RUST, both self-written and ones used in the real-world, and obtain a huge set of these trace files to analyze. In parallel, the research would also require an analysis of how much resources each of these Assembly Instructions take in the CPU. Once a huge set of trace files have been achieved, we can analyze the amount of resources that would have been taken by the CPU to execute these programs, and this gives us insight into the Performance Metrics for the two programming languages on the x86 Architecture. This will be my plan of action over the Winter Break and the Spring 2024 semester.

We plan to parse through each of the Trace Files once achieved, and see how many instructions were executed for each. For specialized instructions involving comparisons and jumps, we would also look at the line numbers to determine if that specific instruction was executed or not. We would continue to collect metrics on such Assembly instructions throughout the execution of a program, and how they were executed (if jumps were executed or not), and then calculate the CPU Clock Cycles that were utilized in executing each of these instructions. This will be conducted via a Python Program which is in the works currently. Thus, we can see from an assembly-logic based view, if RUST or C++ used more CPU Clock Cycles to execute the same program, depending on how the Assembly was executed.

The research will also have to account for RUST's added security measures, which may render additional Assembly Instructions on the low-level. To do so, we would also attempt to replicate similar security and ownership paradigms on C++, and try to compare the two. Similar to the RUST and C Comparison Paper for High Performance Computing (Costanzo *et. al*) mentioned above, we would also measure if the ease of code and security is a valuable tradeoff for the changes in bare performance, as these paradigms are not provided by C++ at all. The added value of our approach would include an inclusivity of all hardware, ranging from consumer laptops to high performance computing, as we focus solely on the logic of the compiler itself.

We plan to conduct this research with no specialized tooling. Mainly, I will be using my own Personal Computers, as well as the EdLab Computers provided by The University of Massachusetts Amherst to conduct this analysis. While the difference in hardware should not matter since the focus is on the logic of the Assembly, the EdLab Computers are high performance servers provided for various Systems related classes at UMass Amherst, and provide a suitable benchmark for a real-world configuration for software development.



Enter file path:
Choose Language: RUST

```
$ as -o main.o main.s
$ ld -o main main.o
ld: warning: cannot find entry symbol _start;
defaulting to 0000000000401000
ld: main.o: in function `std::rt::lang_start'
main.aaccb3f0a8cfdec6-cgu.0:(.text.
g_start17h10f5244e46882c73E+0x23):
reference to `std::rt::lang_start_i
```

Directory: /mnt/c/Users/kushd/Documents/GitHub/RUST-Tracing-Tool

```
.git          traces
archive       asmblr.py
ongoing       python-bytecode-tracer.py
programs      README.md
src
```

File name:
Files of type:

```
ce:/mnt/c/Users/kushd/Documents/Git
ce:/mnt/c/Users/kushd/Documents/Git
ce:/mnt/c/Users/kushd/Documents/Git

ce:/mnt/c/Users/kushd/Documents/Git
ce:/mnt/c/Users/kushd/Documents/Git
ce:/mnt/c/Users/kushd/Documents/Git
ce:/mnt/c/Users/kushd/Documents/Git
ce:/mnt/c/Users/kushd/Documents/Git
```

Figure 3: Alpha Version of the Tracing Tool, running GDB on the background

Evaluation and Communication

The main criteria for evaluation for the project would be two-fold: a Thesis describing the different findings and analyses of the project, as well as the tools that were developed to conduct x86 Assembly Tracing, and then Parsing the logs in the second phase of the research. Moreover, I also plan to meet Prof. Meng-Chieh Chiu and other Committee Members weekly for about 30 minutes to 1 hour, and give weekly updates on my progress. This would be done in the form of a small presentation and a live demo, either over Zoom or in-person. If required, I will also supplement the same with written progress reports. Moreover, the code for my tracing tool is already existent in a GitHub Repository[13], and I will continue to develop the software in the same repository moving forward.

By the end of the Winter Break, I am expected to clear all bugs from my x86 Assembly Tracer and the code that parses logs of the Assembly traces, so that we can seamlessly begin Phase 2 of the project in Spring 2024. I believe that by the end of the first month of Spring 2024, I have gathered logs for various programs on either languages, and that we can move on to analyzing the trace files in depth.

I would receive feedback from my Thesis Committee in my weekly meetings, during Zoom meetings where I lay out my progress report. Moreover, the committee will also provide me feedback on my code, and the tracing tool by trying to use it on their own computers, as well as by judging the outputs of the Assembly trace that are obtained live. If the project is not viable, I will discuss it with my thesis committee at the earliest to figure out alternative methods for the research work.

The committee and I will also communicate more frequently via email, and any discrepancies can be upheld via emails, or zoom/in-person meetings. Overall, my final products include a Thesis, and an Open Source Repository for my programs. Further materials include weekly reports, presentations, and live demos to ensure that the research stays on track.

[13] https://github.com/cics-syslab/RUST-TRACING-TOOL

Timeline

Outline

I have already completed the first phase of the research, and have made significant progress on the second phase. I believe that February 15, 2024 is a good date for the first review of the outline of my manuscript, as it gives me enough time over the Winter Break to draft it, and some time in the semester to receive feedback.

First Draft

I believe that by March 15, 2024, I would have gotten preliminary findings on the performance differences. Thus, we can tentatively aim for March 25, 2024 as a deadline for my first draft.

Second Draft

I would like to tentatively aim for April 15, 2024 as the deadline for my Second Draft. At this point, most of the work for the research should have been completed.

Final Manuscript and Artifacts

I would prefer to set May 5, 2024 as the deadline for the Final Manuscript. This would be about a week prior to the final day of classes, and everything should have wrapped up by now. Moreover, artifacts would include my Code Repository, and a clean version of it would be submitted alongside the Final Manuscript (as a URL).

In the final week of classes before the Final Exams, I would like to communicate and plan a public presentation for my work: with committee members and faculty sponsors. I intend to make it public to inspire more people to dive into low level systems programming, as this would be a showcase of how the barrier can be lowered into this field.

Bibliography

Chiu, M.-C. (2018). Run-Time Program Phase Detection and Prediction. *University of Massachusetts Amherst Graduate School*, 110.

Coloutier, F. (2023, March). *x86 and amd64 instruction reference*. Retrieved from https://www.felixcloutier.com/x86/

Energy Innovation. (2020, March 17). *How Much Energy do Data Centers Actually Use*. Retrieved from Energy Innovation: https://energyinnovation.org/2020/03/17/how-much-energy-do-data-centers-really-use/

Godbolt, M. (n.d.). *Compiler Explorer*. Retrieved from https://godbolt.org/

Intel. (2023, November 12). *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Retrieved from Intel Developer Guides: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

LLVM. (n.d.). *Clang Compiler User Manual*. Retrieved from https://clang.llvm.org/docs/UsersManual.html

LLVM. (n.d.). *The LLVM Compiler Infrastructure Documentation*. Retrieved from https://llvm.org

Manuel Costanzo, E. R. (2021). Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. *Cornell University arxiv*, 21.

Ng, V. (2023). Rust vs. C++, a battle of Speed and Efficiency. *ResearchGate*, 7.

The Python Programming Language. (n.d.). *GDB Support*. Retrieved from Python Developer's Guide: https://devguide.python.org/development-tools/gdb/

The RUST Programming Language. (n.d.). *RUST Compiler Developer Guide*. Retrieved from https://rustc-dev-guide.rust-lang.org/

Wikipedia. (n.d.). *List of Countries by Electricity Consumption*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/List_of_countries_by_electricity_consumption