Comparing Performance Differences Between RUST and C++

Kushagra Srivastava
Manning CICS
University of Massachusetts
Amherst, MA, U.S.A.
ksrivastava [at] umass [dot] edu

Prof. Meng-Chieh Chiu
Manning CICS
University of Massachusetts
Amherst, MA, U.S.A.
joe [at] umass [dot] edu

Prof. Timothy Richards
Manning CICS
University of Massachusetts
Amherst, MA, U.S.A.
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 execution, 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/.**

***Index Terms*—compilers, programming languages, assembly, systems, performance**

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.

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 reseach 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 a 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 [**?**], 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 [**?**]. Similarly, macOS being a UNIX variant, and the Windows NT Kernel are mainly C-based as well [**?**]. 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$_variable concurrently, leading to a race condition where the final value is unpredictable. However, C will let us run this with no issu$

```
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ gcc race.c
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1138441
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1339034
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1070599
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1103020
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1073974
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1233980
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1403249
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1200399
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1036593
suobset@Kush-Surface:/mnt/c/Users/kushd/Documents/GitHub/temp$ ./a.out
Shared variable: 1115206
```

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-secbb113be1/web.

[6] Linus Torvalds and The Linux Foundation, Linux Kernel, n.d., accessed April 30, 2024, https://github.com/torvalds/linux.

[7] The Free Software Foundation, GNU Coreutils, n.d., accessed April 30, 2024, https://github.com/coreutils/coreutils.

[8] 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.