Micro-benchmarks:

A comparison of C++, Rust and Zig

Fucă Răzvan-Ionuț

Technical University of Cluj-Napoca

Table of contents:

1. Introduction........................................................................................
   1. Context……………………………………………………..
   2. Language selection…………………………………………
   3. Objectives…………………………………………………..
   4. Hypothesis………………………………………………….
2. Bibliographic Study
3. Analysis
4. Design
5. Implementation
6. Testing and validation
7. Conclusions
8. Bibliography

Introduction

Context

The purpose of this project is to conduct a comprehensive benchmark of the execution time of various processes in three distinct programming languages: C++, Rust, and Zig. These languages have been chosen due to their classification as "systems programming languages," which means they offer direct access to hardware while also providing a robust, OS-agnostic API. Notably, they all support multi-threading, making them suitable for low-level and high-performance tasks. This project will delve into the nuances of these languages, shedding light on their capabilities and performance characteristics.

The primary motivations for this benchmark are to gain insights into how different systems programming languages handle critical tasks, such as memory allocation, memory access, and multi-threading. By analyzing their performance in these areas, we can understand the trade-offs and advantages offered by each language. This knowledge can be valuable for selecting the most suitable language for specific projects or optimizing existing code-bases.

Language selection

The languages chosen for this project are:

* **C++** - A widely used systems programming language that offers extensive control over memory management and hardware interactions. C++ has a long history and a rich ecosystem of libraries and tools.
* **Rust** - A modern systems programming language known for its strong focus on memory safety without sacrificing performance. Rust's unique ownership system and borrowing rules set it apart from traditional languages.
* **Zig** - An up-and-coming systems programming language that combines a C-like syntax with modern features and a focus on compile-time safety. Zig's custom allocators and comptime code are key features that distinguish it.

It's worth noting that while Go (Golang) is also a compiled, statically linked language with garbage collection, it does not support traditional threads but rather relies on goroutines, making it unsuitable for direct comparison in this project.

Objectives

The primary objectives of this project are:

* Develop sample programs in each of the selected languages that are specifically designed to test the performance of critical processes, including:
  + Memory allocation
  + Static memory access
  + Dynamic memory access
  + Thread creation
  + Thread context switch
  + Thread migration
* Execute the sample programs in order to collect and interpret the data.

Hypothesis

The hypothesis for this project is that the results of the benchmarks will demonstrate close performance characteristics among the selected languages. This expectation is based on the fact that all three languages ultimately compile down to assembly instructions, but differences will arise from:

* Compiler Optimizations: Each language's compiler applies its own set of optimizations, affecting execution speed.
* Programming Paradigms: The choice of language influences the programming paradigms and patterns used, which can impact performance
* Memory Management Strategies: The languages vary in their approach to memory management, such as C++ relying on manual memory management, Rust utilizing it's borrow checker, and Zig offering custom allocators
* The most significant differences will result from The optimizations of the compiler and from the programming paradigms, and memory management strategies that each language encourages. (C++ manual mememory management, Rust's borrow checker, Zig custom allocators)

Bibliographic study

Introduction to the languages

The programming languages selected for this micro-benchmarking study are C++, Rust, and Zig. These languages have the ability to provide low-level control over hardware resources and memory management. In this section, we provide an overview of these languages and their relevance to our benchmarking project.

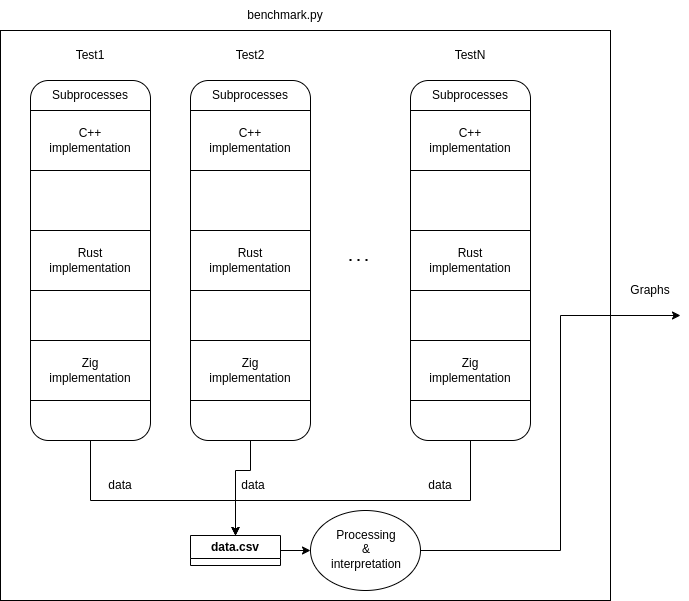
* C++ is a well-established systems programming language known for its versatility and extensive control over memory management and hardware interactions. It has a rich history, starting as an extension of C in 1979.
* Rust is a modern systems programming language that has garnered attention for its emphasis on memory safety without compromising performance. It distinguishes itself with its ownership system and borrowing rules, that force the programmer to adopt the RAII (resource acquisition is initialization) programming pattern which eliminates common sources of memory-related bugs [2]. Rust’s development started in 2006.
* Zig is an emerging systems programming language that combines a C-like syntax with modern features. It places a strong focus on compile-time safety, which aligns well with our objective to assess language-level safety and performance. Notably, Zig offers features like custom allocators and compile-time code that set it apart from traditional systems programming languages [3]. Zig first appeared in 2016, being the newest but also the least mature language used in this project.

Visualisation tools

To effectively present and analyze the data collected from our benchmarking programs, we will utilize Matplotlib, a widely-used Python library for creating high-quality data visualizations. Matplotlib provides a versatile and user-friendly platform for generating graphs and charts, making it an ideal choice for representing the performance metrics obtained during our micro-benchmarking study. Matplotlib has also been chosen because of its ease of processing external data such as csv files.

Analysis & Design

The programs will be CLI apps obeying \*NIX philosophy that will perform a series of algorithms designed to benchmark the aforementioned processes and will dump the results to a file in a plot-friendly format (CSV) that will later be processed by a script that will prune and interpret the data. Figure 1 showcases the general flow of data in our project.

Figure 1

The type of benchmarks that will be performed can be grouped quite easily into two main categories:

* Memory benchmarks
* Threading benchmarks

Memory benchmarks

For the memory benchmarks, the programs will perform a series elementary operations on increasingly larger common data structures. A complete list of actions to be performed that I believe will encompass most common applications is:

* Allocating dynamic arrays
* Creating linked lists (or similar dynamic data structures)
* Performing operations such as sorting or traversals on the created data structures

Threading benchmarks

For the threading benchmarks the measuring process will be a little more difficult as there is no explicit way of directly accomplishing our desired results, however there are workarounds that will force the OS to perform operations such as context switching and migration. A list of actions that I believe will adequately measure the desired processes consists of:

* Calling an empty function in a synchronous and asynchronous way and taking the difference between the two in order to gather quantitative data regarding thread creation
* Writing and reading bytes of data from a pipe in order to force 2 thread context switches
* Setting thread CPU affinity using functions like “pthread\_set\_affinity\_np” or similar depending on the platform in order to force thread migration

All of these benchmarks will be time-based, using each of the language’s built-in time measuring method (C++ std::chrono::high\_resolution\_clock, Zig’s std.time.Timer, Rust’s std::time::Instant)

Bibliography

[1] “C++ language refernce” [Online]. Available: <https://en.cppreference.com/w/cpp/language>

[2] “Rust language refernce” [Online]. Available: <https://doc.rust-lang.org/stable/reference/>

[3] “Zig language refernce” [Online]. Available: https://ziglang.org/documentation/master/

[4] “Matplotlib documentation” [Online]. Available: <https://matplotlib.org/stable/index.html>