

Micro-benchmarks: A comparison of C++, Rust, and Zig

Fucă Răzvan-Ionuț
Technical University of Cluj-Napoca

Contents

1	Introduction	3
1.1	Context	3
1.2	Language selection	3
1.3	Objectives	3
1.4	Hypothesis	3
2	Bibliographic Study	4
2.1	Introduction to the languages	4
2.2	Visualization tools	4
3	Analysis & Design	5
3.1	Memory benchmarks	5
3.2	Threading benchmarks	5
4	Implementation	7
4.1	Memory tests, dynamic data	7
4.2	Memory tests, static data	7
4.3	Threading tests, thread creation	7
4.4	Threading tests, thread context switch	7
4.5	Threading tests, thread migration	7
4.6	Benchmarking script	7
5	Bibliography	8

1 Introduction

1.1 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.

1.2 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.

1.3 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.

1.4 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 its borrow checker, and Zig offering custom allocators.

2 Bibliographic Study

2.1 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, forcing the programmer to adopt the RAII (resource acquisition is initialization) programming pattern, eliminating common sources of memory-related bugs.
- 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.

2.2 Visualization 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.

3 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.

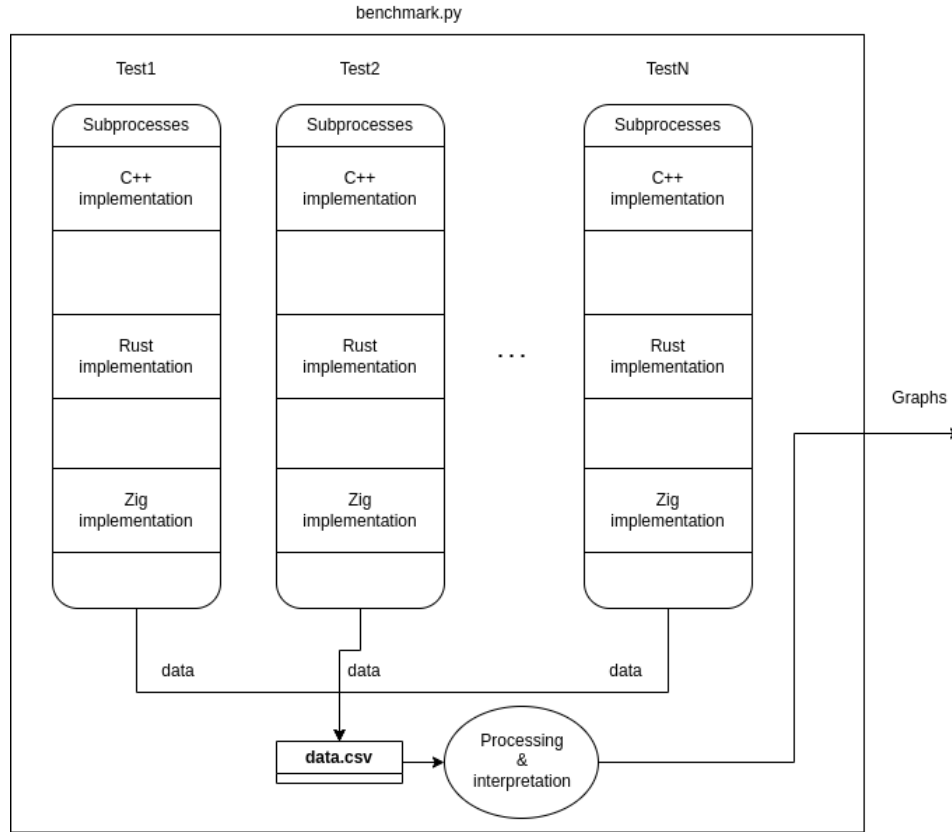


Figure 1: General flow of data in the project.

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

- Memory benchmarks
- Threading benchmarks

3.1 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

3.2 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 computing the difference between the two to gather quantitative data regarding thread creation
- Writing and reading bytes of data from a pipe to force two thread context switches
- Setting thread CPU affinity using functions like `pthread_set_affinity_np` or similar depending on the platform to force thread migration

All of these benchmarks will be time-based, using each of the language's built-in time measuring method that will yeild results in nanoseconds:

- C++ `std::chrono::high_resolution_clock`
- Zig's `std.time.Timer`
- Rust's `std::time::Instant`

4 Implementation

4.1 Memory tests, dynamic data

The first step of running our benchmarks is implementing the necessary data structures we need to test. The most notable of the data structures is a generic singly linked list that will showcase each language's lower-level features (manual memory allocation, the use of pointers and nullable/optional data types, etc.), as well as higher-level features (OOP, static and dynamic dispatch, or compile-time code), and the programming paradigm they encourage.

The implementation details of each are described below:

- **C++** – Singly linked list using templates
- **Zig** – Compile-time function that returns a new type
- **Rust** – Generic list that uses the 'Box' smart pointer for dynamic memory allocation

Another test relies on the dynamic allocation of large arrays on the heap using each of the languages specific constructs

- **C++** – Using the new operator
- **Zig** – Using different allocators provided by the standard library
- **Rust** – Using `Vec::with_capacity` as it is the closest analogue to allocating a dynamic array of a known size in rust

4.2 Memory tests, static data

Each program will allocate an array on the stack of size 100000 and then traverse it, setting the value of each entry in the array to their index.

4.3 Threading tests, thread creation

Each program will run a function that returns as soon as possible. At first by launching it in a thread, and secondly by running it directly. We will measure the time difference between these two scenarios in order to quantify thread creation.

This will be accomplished using:

- **C++** – `std::thread`
- **Zig** – `std.Thread`
- **Rust** – `std::thread`

4.4 Threading tests, thread context switch

The programs will create a child thread and ping_pong a single byte between parent and child. This will force two thread context switches per read/write

4.5 Threading tests, thread migration

The programs will leverage `pthread_set_affinity_np` in order to force the tested thread to run on a particular processor. Following that the affinity will be set to a single separate processor. We will force the thread to migrate from processor one to processor two using the linux system call `sched_yield`.

4.6 Benchmarking script

The script that will run and aggregate the data generated by the programs is `benchmark.py`. This benchmarking script relies on a couple of files

- **config.yaml** Where the benchmarks parameters will be defined. Ex (Size of array, number of runs, weather or not to recompile the programs) etc.
- **data.csv** where all the programs will dump their results

The script also provides a GUI in if the user prefers to use it instead of a config file.

5 Bibliography

1. *C++ language reference*. [Online]. Available: <https://en.cppreference.com/w/cpp/language>
2. *Rust language reference*. [Online]. Available: <https://doc.rust-lang.org/stable/reference/>
3. *Zig language reference*. [Online]. Available: <https://ziglang.org/documentation/master/>
4. *Matplotlib documentation*. [Online]. Available: <https://matplotlib.org/stable/index.html>
5. *Rust by example: RAI*. [Online]. Available: <https://doc.rust-lang.org/rust-by-example/scope/raii.html>