Rust vs. C: A Performance Breakdown

Cayden Lund (u1182408)
13 December 2024

Contents

1. Introduction

In the realm of systems programming, C and Rust represent two significant paradigms of low-level language design. While C has been a cornerstone of systems programming for decades, Rust emerges as a modern alternative promising memory safety without sacrificing performance.

Recently, Rust is often noted for its strong type system, memory safety guarantees, and modern concurrency features, yet it's generally thought to be slightly slower than C. This paper aims to provide a comprehensive comparative analysis of their performance differences. Understanding the performance slowdowns is particularly relevant to modern operating systems; Linux and Windows have been integrating Rust code into their kernels, several new operating systems in Rust are quickly gaining popularity, and world governments are pressuring companies to transition to memory-safe languages in the next few years. Therefore, having a good understanding of the performance challenges that Rust is facing, and especially their causes, is a relevant and important issue.

2. Language Fundamentals

2.a. C: The Classic Systems Language

C emerged as a revolutionary programming language developed by Dennis Ritchie at Bell Labs in the early 1970s, fundamentally transforming systems programming paradigms. The language was purposefully designed to provide programmers with unprecedented low-level control over system resources, enabling direct interaction with hardware at a granular level. Its design philosophy centers on three primary principles: minimal runtime overhead, maximal programmer control, and efficient resource utilization.

The language's architectural characteristics reflect a deep commitment to performance and system-level manipulation. C provides direct memory management, allowing programmers to explicitly allocate and deallocate memory resources, which enables fine-grained control but simultaneously introduces significant potential for memory-related errors. Unlike higher-level languages, C does not incorporate inherent memory safety mechanisms, placing the entire responsibility of memory integrity on the programmer's expertise and diligence.

Key characteristics of C include its procedural programming paradigm, which emphasizes a structured approach to code organization through functions and modular design. The language's minimal abstraction overhead ensures that compiled code remains extremely close to machine instructions, resulting in exceptional runtime performance. Furthermore, decades of compiler development have produced mature optimization technologies that can transform C code into highly efficient machine code, making it a persistent choice for performance-critical systems such as operating systems, embedded systems, and low-level system utilities.

2.b. Rust: Modern Systems Programming

Rust represents a paradigm shift in systems programming, developed by Mozilla Research as a response to the long-standing challenges of memory safety and concurrent programming. Unlike traditional systems languages, Rust introduces a revolutionary ownership and borrowing memory model that provides memory safety guarantees at compile-time, effectively eliminating entire classes of programming errors that have historically plagued systems development.

The language's core innovation lies in its ownership system, which enforces strict rules about memory access and lifecycle management during compilation. This approach allows Rust to provide memory safety and thread safety guarantees without incurring the runtime performance penalties associated with garbage collection. By implementing these checks statically, Rust ensures that common programming errors such as null or dangling pointer dereferences, buffer overflows, and data races are prevented before the program even executes.

Rust's design philosophy embraces the concept of zero-cost abstractions, meaning that high-level programming constructs compile down to efficient machine code with no additional runtime overhead. The language features a sophisticated type system with strong compile-time checks that go beyond traditional type safety, enabling developers to express complex invariants and system constraints directly in the type system. This approach allows for more robust and self-documenting code while maintaining performance characteristics nearly as fast as other low-level languages like C. The Rust compiler does well at its goal of zero-cost abstraction—and, subjectively, the abstraction is well worth the cost—but it doesn't seem to quite compare to C's raw performance.

The modern type system in Rust supports advanced features such as algebraic data types, pattern matching, and comprehensive generics, providing developers with powerful tools for expressing complex computational logic. Unlike C, which relies on programmer discipline for safe code, Rust's compiler actively prevents entire categories of potential errors, shifting the burden of safety from runtime checks to compile-time verification.

3. Performance Benchmarking Methodology

3.a. Benchmark Categories and Rationale

I employed a multi-dimensional benchmark approach designed to capture the performance characteristics of Rust and C across different domains. The selected benchmark categories were chosen to provide a holistic assessment of language performance:

1. **Memory Management Performance**
   This category evaluates the efficiency and overhead associated with dynamic memory allocation and deallocation. The implementation of this benchmark is in **`malloc-c`** and **`malloc-rs`** for C and Rust, respectively.
2. **Computational Efficiency**
   Computational benchmarks assess the raw processing capabilities of both languages across different challenges:
   - Numerical computing performance through matrix-matrix multiplication in **`matrix-c`** and **`matrix-rs`**
   - Algorithmic complexity handling through different sorting algorithms in **`sort-c`** and **`sort-rs`**
3. **Concurrency Performance**
   Given the increasing importance of parallel computing, this category examines thread creation and management overhead in **`concurrency-c`** and **`concurrency-rs`**.

3.b. Experimental Setup and Methodology

3.b.i. Hardware Configuration

My benchmark infrastructure utilized a laptop computer with the following specifications:

- **Processor:** AMD Ryzen 5 5500U
  - 6 physical cores (12 threads)
  - Base clock: 2.1 GHz
  - Turbo boost up to 4.0 GHz
- **RAM:** 16 GB DDR4 2400 MT/s
- **Storage:** Toshiba KXG5AZNV512G NVMe SSD
- **Operating System:** Linux 6.6.63 (Kernel), NixOS distribution, with performance governor enabled

3.b.ii. Compiler Configuration

To ensure fair and representative comparisons, I employed the following compiler versions:

- C Compiler: Clang (LLVM C frontend) version 18.1.8
  - Configured with full optimization support
- Rust Compiler: Rust 1.83.0 Stable Release
  - Default stable channel
  - Configured with release optimization profile

3.b.iii. Benchmark Optimization and Configuration

- Optimization Level: `-O3` for both languages
  - Enables maximum compiler optimizations
  - Allows aggressive inlining and code generation
- Compilation Flags:
  - C: `-march=native -mtune=native -flto`
  - Rust: `-C target-cpu=native lto=true`

3.b.iv. Reproducibility and Transparency

All benchmark source code, raw data, and detailed methodology are publicly available in the associated GitHub repository to ensure complete transparency and independent verification.

4. Benchmark Results

4.a. `malloc-c` and `malloc-rs`

4.a.i. C:

Number of allocations	Malloc runtime (ns)	Free runtime (ns)
10	282	149
100	2087	1296
1000	187746	49918
10000	2965090	1407816

4.a.ii. Rust:

Number of allocations	Malloc runtime (ns)	Free runtime (ns)
10	456	171
100	5031	1488
1000	782390	160985

Number of allocations	Malloc runtime (ns)	Free runtime (ns)
10000	10270961	2961104

4.b. `matrix-c` and `matrix-rs`

4.b.i. C:

Matrix dim.	Runtime (ns)	
40	33537	80
281912	160	2807064
320	23935532	

4.b.ii. Rust:

Matrix dim.	Runtime (ns)	
40	49020	80
413897	160	3187199
320	25249436	

4.c. `sort-c` and `sort-rs`

4.c.i. C:

Array length	Insertion sort runtime (ns)	Merge sort runtime (ns)
Radix sort runtime (ns)	Validation runtime (ns)	
10	168	688
1843	21	100
2963	8984	5295
126	1000	135559
96818	39920	982
10000	14868382	1112055
392420	9463	

4.c.ii. Rust:

Array length	Insertion sort runtime (ns)	Merge sort runtime (ns)
Radix sort runtime (ns)	Validation runtime (ns)	
10	116	490
1478	16	100
1688	5516	3155
55	1000	106577
68256	21687	452
10000	9510456	813836

Array length	Insertion sort runtime (ns)	Merge sort runtime (ns)
Radix sort runtime (ns)	Validation runtime (ns)	
221366	4489	

4.d. `concurrency-c` and `concurrency-rs`

4.d.i. C:

Number of threads	Matrix size	Runtime (ns)
1	10	41852
2	10	54338
4	10	92583
6	10	183286
8	10	227091
1	100	657328
2	100	400184
4	100	251194
6	100	264599
8	100	286981
1	400	46047054
2	400	24367671
4	400	12156049
6	400	9415956
8	400	9658438

4.d.ii. Rust:

Number of threads	Matrix size	Runtime (ns)
1	10	39372
2	10	55271
4	10	85267
6	10	133467
8	10	162697
1	100	923054
2	100	552538
4	100	307088
6	100	309694
8	100	313766
1	400	51287927
2	400	26605527

Number of threads	Matrix size	Runtime (ns)
4	400	13590651
6	400	11203079
8	400	13722652

5. Incomplete Work

I ran out of time to do a binary assembly analysis of the output C and Rust programs. I still intend to do this in the coming weeks, just for fun! But unfortunately, despite all the time I spent on coming up with benchmarks, I am not able to identify specific, measurable reasons why Rust programs are *generally* slower than C programs.