# Proposal: A Comparison of Concurrency in Rust and C

Josh Pfosi Riley Wood Henry Zhou

April 6, 2016

## Background

Computer architects today are faced with a power wall which limits the trend of rising clock frequency seen over the past two decades. Yet the number of transistors on chip continues to double every two years as predicted by Moore's Law, architects are left with this question: How can we continue to improve processor performance without scaling clock speed? The industry has answered with multicore processors [1]. The industry has begun to favor parallelism over clock frequency. This shift has significant ramifications for programmers. A program's performance scales trivially with clock frequency, whereas parallel programs oftentimes require significant design effort on the part of the programmer to scale well. As of now, multicore machines burden the programmer with efficient usage of the parallel cores [2]. It is difficult for most programmers to decompose an algorithm into parallel workloads and to anticipate all of the interactions that can occur. This proliferates new kinds of errors that are much harder to anticipate, find, and prevent. These include deadlock, livelock, and data races. It is important that programmers start writing parallel programs in order to scale software performance with hardware. To this end, programmers should feel confident in their ability to write safe, parallel code. The question becomes: Which tools currently available are best suited for this task?

Rust is a new systems programming language that facilitates writing safe concurrent programs through an advanced type system [3]. It introduces the idea of data ownership where variable bindings "own" the data to which they point. This and other statically enforced safety features reduce the burden of concurrent programming on the programmer. Contrast this with other standard concurrent programming models such as OpenMP, `pthreads`, and C++11's `std::threads`, which are difficult to learn and error-prone, Rust's approach is novel.

To investigate the research question, we have selected Rust and C's `pthreads` as representatives for contemporary approaches to parallelism. Namely, Rust, which relies on static analysis, and `pthreads`, which relies on programmer expertise, project-specific conventions, and manual synchronization. The specific contribution made by this research will be to glean whether Rust's strict compile-time rules and "extra runtime book-keeping" [3] have an adverse effect on performance.

## Goal and Description

Rust's concurrency constructs and paradigms make writing concurrent code safer and easier by preventing code with common concurrency errors from compiling. We are interested to see if Rust's new constructs affect its performance in a significantly detrimental way when compared with C, a widely-used concurrency-capable language. Metrics we are interested in include program execution time, power consumption, and idle time. We will present our findings through a presentation and written report.

Our findings should be useful to systems programmers who are evaluating languages for their next project. With our results, they should be able to make a more educated decision about which systems language, C or Rust, provides a better trade-off between programmer productivity and performance.

## Methodology

Rust and C are inherently different in many ways, so we will need to isolate how each one's implementation of concurrency impact performance. We will do this in the following way:

We will choose a set of benchmarks that can be run both serially and concurrently and implement them in C and Rust. We will compare the performance of the two languages on the single-threaded benchmarks to establish a baseline for how Rust and C differ. Then any further differences that arise in our multi-threaded tests can roughly be attributed to each language's concurrency model. Our benchmarks will be run in a multithreaded-capable simulator. We will vary parameters such as input size and the number of threads and collect metrics such as total execution time, CPU idle time, and power consumption.

Furthermore, we plan on exploring the differences in performance observed using different hardware. For this section, we will execute concurrent benchmarks at different thread counts with different hardware settings such as L1/L2 cache size and cache associativity. We seek to examine if changes in relative performance between C and Rust arise when running benchmarks on different hardware.

## Equipment and Tools

We will be writing code in C and Rust. For more convincing analysis, we will compile our C code using both clang and gcc. Rust will be compiled with the standard rustc compiler. We are still in the process of selecting a suite of concurrent benchmarks to use. Once we have chosen a set of benchmarks and ported them to Rust, we will run our simulations in the Sniper Multi-Core Simulator [4] on the Linux compute servers hosted at Tufts University.

## Timeline

| | |
|---|---|
| April 11 | Select benchmarks. |
| April 15 | Implement benchmarks in C and Rust and setup testing environment. |
| April 20 | Project status discussion with Dr. Hempstead |
| April 22 | Completed all benchmarks and collect all data |
| April 27 | Project Oral Presentation |
| May 3 | Final Report Submission |

## References


[1] J. Larus, "Spending moore's dividend," *Communications of the ACM*, pp. 62–69, may 2009.

[2] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, pp. 54–62, sep 2005.

[3] "The rust programming language," https://www.rust-lang.org.

[4] "The sniper multi-core simulator," http://snipersim.org/.