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. CPU clock speeds cannot increase any further, else they will be unable to be cooled. Yet the number of transistors onchip continues to double every two years as dictated by Moore's Law, so the question to architects is this: what can we spend additional transistors on, if not clock speed? The industry has answered this question by converging on multicore computers: computers that integrate multiple processor cores into one chip [1]. The industry is forced to favor increased parallelism over increased speed.

This shift has huge ramifications for programmers. A program's performance improves automatically as clock speed increases, but programs need to be written specially to take advantage of hardware parallelism. As of now, multicore machines put the burden of utilizing parallel cores on the programmer, and writing parallel programs is difficult. It is hard for most people to decompose a program into parallel workloads and think about all of the interactions that can occur as they are executing at once. This makes the new types of errors that arise with concurrency that much harder to find, anticipate, and prevent. These include deadlock, livelock, and pointer aliasing, to name a few. It's important that programmers start writing parallel programs in order to take advantage of all that Moore's Law offers. But for this to happen, programmers need to feel confident in their ability to write safe, parallel code.

Highlight how writing concurrent programs is still difficult, prone to error. Rust is a new systems programming language that attempts to make writing concurrent programs easier for programmers through its type system.

Rust is a new systems programming language which guarantees memory safety through a complex type system.

Discuss Rust's concurrency constructs. Rust uses certain types to perform "extra runtime bookkeeping" on shared values, which we believe could negatively impact performance [2]. Discuss pthreads in C.

Goal and Description

Rust's new concurrency constructs make writing concurrent code safer and easier by preventing concurrency errors at compile time. We are interested in comparing the performance of threading in Rust and in C to see if Rust's new constructs affect its peformance in anyway. In other words, Rust helps programmers write concurrent code, but at what cost to performance?

What metrics are we interested in?

Methodology

Rust and C are inherently different in many ways unrelated to their implementations of concurrency, so we will need to isolate how each one's implementations 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 parallelized. We will implement a serial and multithreaded version of each benchmark in both C and Rust. We will use the single-threaded benchmarks to establish a baseline comparison between the two languages. Then any further differences in performance that arise during the multithreaded tests can be conclusively attributed to each language's implementation of concurrency.

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 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 [3] on the Linux compute servers hosted at Tufts University.

Timeline

References

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

[2] The rust programming language. `https://www.rust-lang.org`.

[3] The sniper multi-core simulator. `http://snipersim.org/`.