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, or else they will be unable to be cooled. Yet the number of transistors onchip continues to double every two years as predicted 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 anticipate, find, and prevent. These include deadlock, livelock, and data races. 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.

Rust is a new systems programming language that facilitates writing safe concurrent programs through an advanced type system [2]. 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 C which primarily relies on lock-based synchronization, a greater degree of programmer expertise, and often project-specific conventions. This creates pitfalls for programmers and proliferates common and often subtle concurrency bugs. An unresolved question, however, is whether Rust's strict compile-time rules and "extra runtime bookeeping" has an adverse effect on metrics such as performance, power and programmer productivity and how severe are these effects [2].

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 performance 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, 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 parallelized 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 safely be attributed to each language's implementation of concurrency. Our benchmarks will be run in a multicore simulator. We will vary parameters such as input size and the number of threads running and collect data on 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 [3] on the Linux compute servers hosted at Tufts University.

Timeline

April 15: Completed creation of benchmarks in C and Rust and setup of 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] 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/`.