Lecture 19 — Optimizing the Compiler

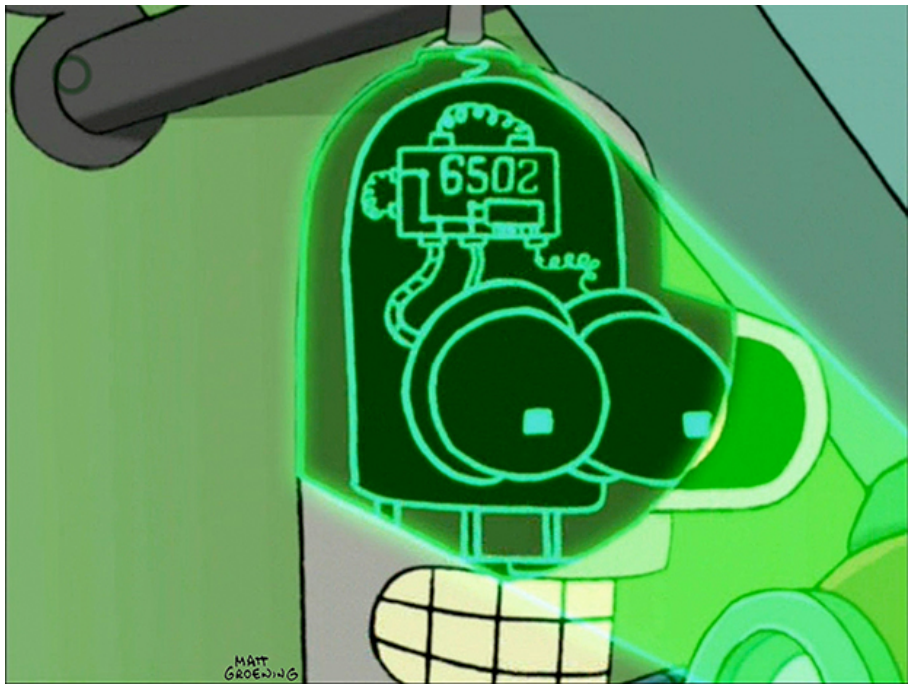
Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

The Reverse?

Last time, we talked about optimizations that the compiler can do.



This time, we'll switch our focus to optimizing the compiler itself.

Dr. Nicholas Nethercote (author of Valgrind and of its associated PhD thesis) has written a serious of blog posts describing how to improve the compiler itself.

Good News Everyone!

An observation back from 2016:

> *Any time you throw a new profiler at any large codebase that hasn't been heavily optimized there's a good chance you'll be able to make some sizeable improvements.*

No PhD in Compilerology Required

Nicholas Nethercote is a self-described non-rustc expert.

Good news for you: you, too, can improve the performance of systems that you didn't design and don't maintain.

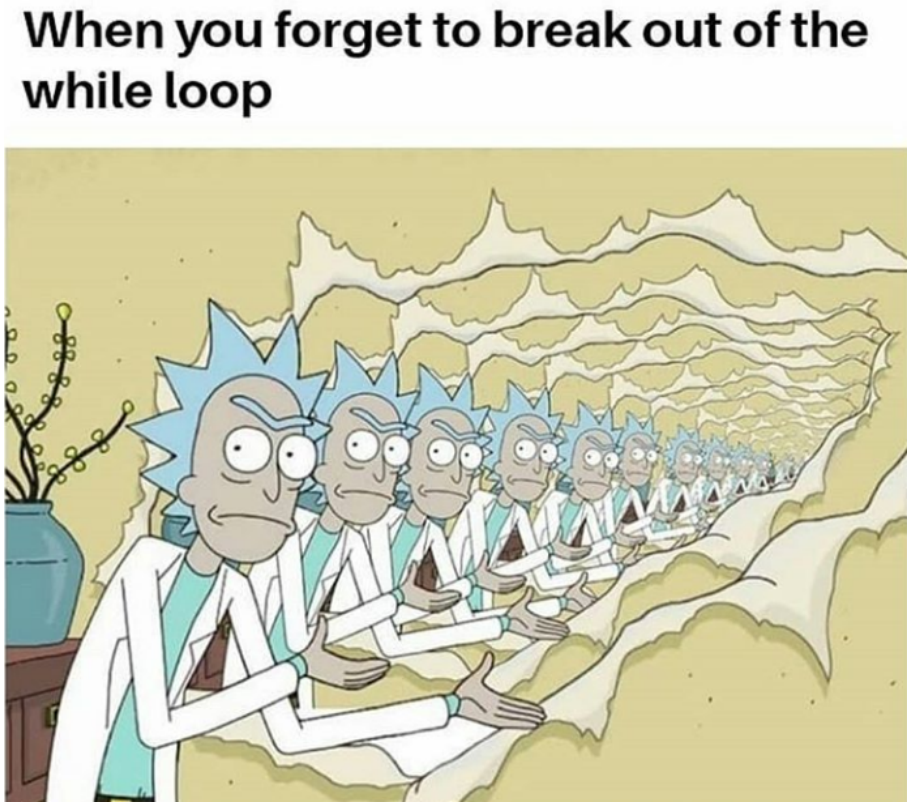
His approach has been to use a benchmark suite (`rustc-perf`) and profilers to find and eliminate hotspots.

The benchmark suite is run on the `https://perf.rust-lang.org/` machine, which is a continuous integration server that publishes runtimes.

Hurry up and wait...

The feedback loop through the remote server is too slow.

You will have a better experience if you run your benchmarks locally on a fast computer.



Make a potential change, benchmark it. Did it improve anything?

Rust Improvements?

Altogether, between January 2019 and July 2019, the Rust compiler reduced its running times by from 20% to 40% on the tested benchmarks.

From November 2017, the number is from 20% to 60%.

Note that a 75% time reduction means that the compiler is four times as fast.

Benchmark Selection

It’s also important to have representative benchmarks.

There are three categories:

- “Real programs that are important”
- “Real programs that stress the compiler”
- “Artificial stress tests”

See Runtimes

You can see the benchmark runtimes for the most recent run at `https://perf.rust-lang.org/status.html`.

> *The test harness is very thorough. For each benchmark, it measures Debug, Opt, and Check (no code generation) invocations. Furthermore, within each of those categories, it does five or more runs, including a normal build and various kinds of incremental builds. A full benchmarking run measures over 400 invocations of rustc.*

Data Collection

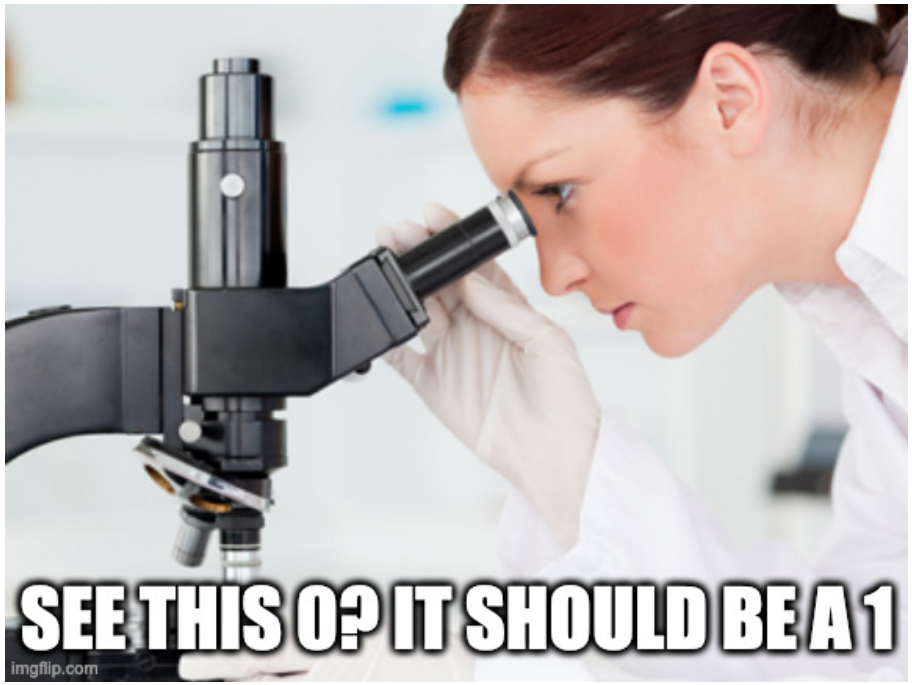
The main event uses `perf-stat` to measure compiler runtimes.

As we know, this tool produces various outputs (wall-clock time, CPU instructions, etc) and the site can display them.

We'll be talking about some profiling tools again soon!

`println!` debugging is a valid option too.

Let’s look at a couple of micro-optimizations.



Each of these is an example of looking through the profiler data, finding a hotspot, making a change, and testing it.

We’ll talk about what we learn from each specific case as well.

Do less Work: memcpy

Do less work: reduced the size of hot structures below 128 bytes, at which point the LLVM backend will emit inline code rather than a call to `memcpy`.

Fewer bytes to move, and one less function call.

Type Sizing

Reducing the size types take up can lead to meaningful speedup:

- ObligationCauseCode, 56 bytes to 32 bytes: speedup up to 2.6%.
- SubregionOrigin, 120 bytes to 32 bytes speedup less than 1%.
- Nonterminal, 240 bytes to 40 bytes: speedup up to 2%.

Interesting question: what is the perf tradeoff involved with boxing?

Compiler, I Command Thee

Manually specifying inlining, specialization, and factoring out common expressions:

- Manually inline a hot function: up to 2%.
- Factor repeated expressions, use iterators: up to 1.7%.
- Specialize a function for a hot calling pattern: up to 2%.
- Inline functions on metadata: 1-5% improvement.

Who Wrote This?

Sometimes addition by subtraction: removing a bad API helps too!

Do Less Work

Some ways in which work was removed:

- Change a hot assertion to run in debug builds: 20% on one workload.
- Is it a keyword? Up to 7%.
- Prefix and suffix matching improvement: up to 3%.
- Only generate one of the bitcode and object code.

Not Everything Works

Some ideas that failed:

- ‘I tried drain_filter in compress. It was slower.”
- Invasive changes in data representation.
- Change to u8to64_le function – simpler but slower.

Architecture Level Changes

These are small items so far – a few percent improvement.

Let’s talk about a couple of larger-scale changes.

One of them works and the other one doesn’t.

Pipelined Compilation

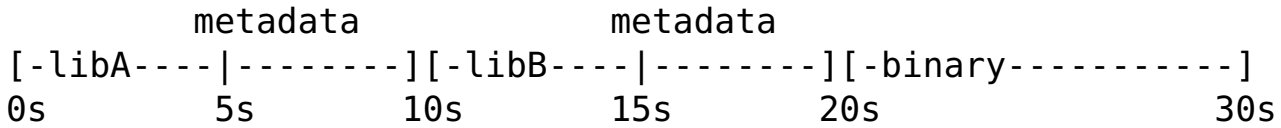
We discussed the idea previously of pipelining for completing tasks.

Here we are talking about compiling multi-crate Rust projects.

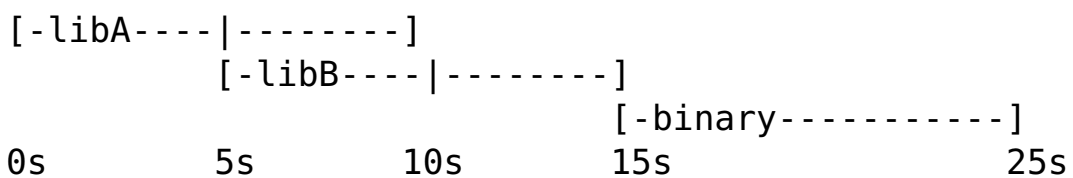
There is a dependency graph, and `cargo` already launches a dependency after all its requisites are met.

What's important here is that there is a certain point before the end at which `rustc` finishing emitting metadata for dependencies.

Here's a picture without pipelining:



and with:



Pipelining Success

The authors were not sure if this would work. But:

- No reproducible regression
- “Build speeds can get up to almost 2x faster in some cases”
- “Across the board there’s an average 10% reduction in build times.

Linking

“I found that using LLD as the linker when building rustc itself reduced the time taken for linking from about 93 seconds to about 41 seconds.”

But that change is blocked by not-`rustc` design problems...

A useful tip for these two optimizations just above: “Don’t have tunnel vision”.