ECE459: Programming for Performance Winter 2024

Lecture 18 — Optimizing the Compiler

Patrick Lam *2024-09-13*

Optimizing the `rustc` Compiler

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 series of blog posts describing his work speeding up the Rust compiler [Net20, Net19b, Net19c, Net19a]. These serve as a good case study for careful optimization work of a large system. There are also posts from 2016 and 2018 but we'll focus on the later posts. For some context, from the first 2016 post [Net16]:

> Rust is a great language, and Mozilla plans to use it extensively in Firefox. However, the Rust compiler (rustc) is quite slow and compile times are a pain point for many Rust users. Recently I've been working on improving that.

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.

Measurement Infrastructure

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`[1]) and profilers (the Rust heap allocation profiler DHAT, or Dynamic Heap Allocation Tool [2] as well as `perf-record` and `Cachegrind`) to find and eliminate hotspots.

The benchmark suite is run on the `https://perf.rust-lang.org/` machine, which is basically a continuous integration server that publishes runtimes. That server is essential for tracking performance regressions.

However, 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 (which is acceptable for this use case: the machinery is representative of typical use). The optimization workflow is to make a potential change and then benchmark it. Acceptable changes don't blow up runtimes for any benchmarks and reduce runtimes by a couple of percents for at least a few benchmarks.

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. We can look at the description of the Rust perf benchmarks[3] as of this writing. There are three categories:

[1] `https://github.com/rust-lang-nursery/rustc-perf`
[2] `https://blog.mozilla.org/nnethercote/2019/04/17/a-better-dhat/`
[3] `https://github.com/rust-lang/rustc-perf/tree/master/collector/benchmarks`

- "Real programs that are important:" The suite includes 15 programs that the community cares about. Recall that we are testing compiler performance here. These benchmarks include features that are "used by many Rust programs", including the commonly-used futures implementation, regular expression parser, and command-line argument parser. So even though the command-line parser only affects a negligible part of total Rust programs' runtime, it comes up in compiling many of these programs. The hello world benchmark is not a real program but it is an important lower bound. The web renderer is important to an important Rust stakeholder.
- "Real programs that stress the compiler:" Here we have 8 edge cases that perhaps don't behave like most Rust programs but are important points in its performance envelope. For instance, `encoding` contains large tables, `keccak` contains many local variables and basic blocks, and `ucd` contains large statics that test the non-lexical lifetimes implementation.
- "Artificial stress tests:" Many of these 18 small programs are performance regression tests for specific issues that have occured in the past (and are sometimes named after the specific issue, e.g. `issue-46499`.) Quadratic or exponential behaviour are problematic if n can grow, and many of the programs here reference the problem.

You can see the benchmark runtimes for the most recent run at `https://perf.rust-lang.org/status.html`. The machine also has a dashboard, and graphs over time (so one can track which commits are responsible for a regression) and the ability to compare two runs. The CI machine also runs various builds starting with different incremental and cache states; to quote:

> 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.

`rustc` itself includes some per-pass profiling, enabled with `-Ztime-passes`. This is coarse-grained information and not so helpful for finding smaller hotspots. Cachegrind's instruction counts are most useful, and point at `malloc` and `free`, whence DHAT.

Briefly about DHAT [Sew10]: it aims to find pathological memory uses. Either repeated `malloc`s of blocks that live briefly, or `malloc`s of blocks that live for the entire program lifetime (and are perhaps never used). Understanding DHAT is beyond the scope of this lecture; see Lecture 28.

I'll also point out that custom `println!` statements can help for all debugging, including performance debugging, and the blog posts mention that in passing, although not specific examples. He post-processes the print results, so I assume that this is a lot of counting of events.

Case Studies: Micro-optimizations

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.

memcpy removal. In the vein of doing less work, there were a number of changes which 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`. So there are two improvements as a result: fewer bytes to move, and one less function call. Nethercote tracked hot calls to `memcpy` by modifying DHAT to find `memcpy`.

Type sizes. You can read about the general strategy of reducing type sizes at `https://nnethercote.github.io/perf-book/type-sizes.html` and some specific examples here:

- "#64302: This PR shrank the ObligationCauseCode type from 56 bytes to 32 bytes by boxing two of its variants, speeding up many benchmarks by up to 2.6%."
- "#64394: This PR reduced the size of the SubregionOrigin type from 120 bytes to 32 bytes by boxing its largest variant, which sped up many benchmarks slightly (by less than 1%). If you are wondering why this type caused memcpy calls despite being less than 128 bytes, it's because it is used in a BTreeMap and the tree nodes exceeded 128 bytes."
- "#67340: This PR shrunk the size of the Nonterminal type from 240 bytes to 40 bytes, reducing the number of memcpy calls (because memcpy is used to copy values larger than 128 bytes), giving wins on a few benchmarks of up to 2%."

You tell me: what is the perf tradeoff involved with boxing, i.e. when does it succeed and when does it fail?

Manual application of compiler techniques. Manually specifying inlining, specialization, and factoring out common expressions:

- "#64420: This PR inlined a hot function, speeding up a few benchmarks by up to 2.8%. The function in question is indirectly recursive, and LLVM will normally refuse to inline such functions. But I was able to work around this by using a trick: creating two variants of the function, one marked with #[inline(always)] (for the hot call sites) and one marked with #[inline(never)] (for the cold call sites)."
- "#64500: This PR did a bunch of code clean-ups, some of which helped performance to the tune of up to 1.7%. The improvements came from factoring out some repeated expressions, and using iterators and retain instead of while loops in some places."
- "#67079: Last year in #64545 I introduced a variant of the shallow_resolved function that was specialized for a hot calling pattern. This PR specialized that function some more, winning up to 2% on a couple of benchmarks."
- "#69256: This PR marked with #[inline] some small hot functions relating to metadata reading and writing, for 1-5% improvements across a number of benchmarks."

Specialization comes up a bunch of other times too. If the compiler can do it, it's better, but sometimes it can't.

Removing bad APIs. #60630: disallows slow symbol-to-string comparisons and forces the use of more consistent symbol-to-symbol comparisons.

Doing less work. A key optimization technique.

- #58210: This PR changed a hot assertion to run only in debug builds, for a 20%(!) win on one workload; `https://github.com/rust-lang/rust/pull/61612` removed unnecessary "is it a keyword" calls for a 7% win on programs with large constants.
- "#70837: There is a function called find_library_crate that does exactly what its name suggests. It did a lot of repetitive prefix and suffix matching on file names stored as PathBufs. The matching was slow, involving lots of re-parsing of paths within PathBuf methods, because PathBuf isn't really designed for this kind of thing. This PR pre-emptively extracted the names of the relevant files as strings and stored them alongside the PathBufs, and changed the matching to use those strings instead, giving wins on various benchmarks of up to 3%."

Another example of doing less work:

> Then Alex Crichton told me something important: the compiler always produces both object code and bitcode for crates. The object code is used when compiling normally, and the bitcode is used when

compiling with link-time optimization (LTO), which is rare. A user is only ever doing one or the other, so producing both kinds of code is typically a waste of time and disk space.

The solution turned out to be somewhat complicated and I won't summarize it here, but yielded improvements of up to 18%. Note the comment: "When faced with messy code that I need to understand, my standard approach is to start refactoring."

Negative Results

Sometimes it's more fun to look at things that don't work. Here are a couple of cases.

- "I tried drain_filter in compress. It was slower."
- "I tried several invasive changes to the data representation, all of which ended up slowing things down."

That last change is a case of making the code more complicated and slower. Obviously not a win. Note also the following report:

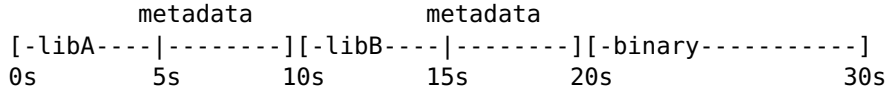
- "#69332: This PR reverted the part of #68914 that changed the u8to64_le function in a way that made it simpler but slower. This didn't have much impact on performance because it's not a hot function, but I'm glad I caught it in case it gets used more in the future. I also added some explanatory comments so nobody else will make the same mistake I did!"

Sometimes making the code simpler is, on second thought, also not a win.

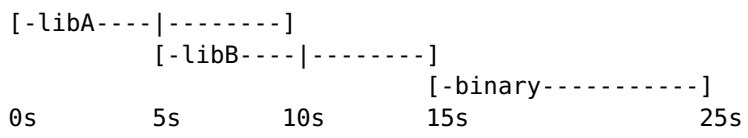
Architecture-level changes

Most of the changes we've discussed so far have been local changes that get a couple of percent speedup. Let's talk about a couple of larger-scale changes. One of them works and the other one doesn't, as of writing.

Pipelined compilation. In Lecture 10 we talked about the pipeline-of-tasks design. Compilers, too, can use this design; the reference is [Net19a]. 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. Once that is complete, then dependencies can proceed in parallel with the requisite. That constitutes pipelining. Here's a picture without pipelining:



and with:



The authors of the patch weren't sure whether it was actually helpful on others' code so they crowdsourced the evaluation of a preliminary version[4]. Here's what they found from the reports (quoting from the crowdsource call):

[4] https://internals.rust-lang.org/t/evaluating-pipelined-rustc-compilation/10199/47

- "Across the board there appears to be no regressions. The reductions in build time here I can't reproduce locally and may have been PIPELINING/PIPELINED confusion (sorry!) and may also just be normal variance. In either case there hasn't yet been a reliably 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. The standard deviation though is pretty and it seems to confirm that you're either seeing large-ish reductions or very little."

Specifically, the compiler-side changes required were[5]: (1) ability to produce an rlib (archive with object code, compressed bytecode, and metadata) with only meta files for dependencies (as opposed to when producing a full linkable artifact); and (2) ability to signal to cargo that meta file is created (implemented using a JSON message from `rustc` to `cargo`)[6]

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." I'm not sure of the total time here, but it looks like linking takes 20 minutes, or 1200 seconds, on the CI infrastructure. A 50 second change is about 2.5%, so a reasonable proportion of the total; it is a larger proportion for incremental builds, and described as "a huge % of the compile". But that change is blocked by not-`rustc` design problems[7], especially across platforms (there is no macOS lld linker backend that works). And the linker on Linux/Unix has to be invoked through the system C compiler, and specifying a specific linker is only possible for some C compilers (if gcc, then at least 9).

What am I saying here? A useful tip for these two optimizations just above: "Don't have tunnel vision", i.e. look at the broader context and where you can really make a difference.

References

[Net16] Nicholas Nethercote. How to speed up the Rust compiler, Oct 2016. Online; accessed 2020-11-06. URL: https://blog.mozilla.org/nnethercote/2016/10/14/how-to-speed-up-the-rust-compiler/.

[Net19a] Nicholas Nethercote. How to speed up the Rust compiler in 2019, Jul 2019. Online; accessed 2020-11-04. URL: https://blog.mozilla.org/nnethercote/2019/07/17/how-to-speed-up-the-rust-compiler-in-2019/.

[Net19b] Nicholas Nethercote. How to speed up the Rust compiler some more in 2019, Oct 2019. Online; accessed 2020-11-04. URL: https://blog.mozilla.org/nnethercote/2019/10/11/how-to-speed-up-the-rust-compiler-some-more-in-2019/.

[Net19c] Nicholas Nethercote. The Rust compiler is still getting faster, Jul 2019. Online; accessed 2020-11-04. URL: https://blog.mozilla.org/nnethercote/2019/07/25/the-rust-compiler-is-still-getting-faster/.

[Net20] Nicholas Nethercote. How to speed up the Rust compiler in 2020, Apr 2020. Online; accessed 2020-11-04. URL: https://blog.mozilla.org/nnethercote/2020/04/24/how-to-speed-up-the-rust-compiler-in-2020/.

[Sew10] Julian Seward. Fun 'n' games with DHAT, Dec 2010. Online; accessed 2020-11-06. URL: https://blog.mozilla.org/jseward/2010/12/05/fun-n-games-with-dhat/.

[5] #58465

[6] PS: more detailed design discussion here: https://rust-lang.github.io/compiler-team/working-groups/pipelining/NOTES/.

[7] https://github.com/rust-lang/rust/issues/39915#issuecomment-618726211