Cranelift code generation comes to Rust

By Daroc AldenMarch 15, 2024

Cranelift is an Apache-2.0-licensed code-generation backend being developed as part of the Wasmtime runtime for WebAssembly. In October 2023, the Rust project made Cranelift available as an optional component in its nightly toolchain. Users can now use Cranelift as the code-generation backend for debug builds of projects written in Rust, making it an opportune time to look at what makes Cranelift different. Cranelift is designed to compete with existing compilers by generating code more quickly than they can, thanks to a stripped-down design that prioritizes only the most important optimizations.

Fast compiler times are one of the many things that users want from their programming languages. Compile times have been a source of complaints about Rust (and other languages that use LLVM) for some time, despite continuing steady progress by the Rust and LLVM projects. Additionally, a compiler that produces code quickly enough is potentially viable in applications where it currently makes more sense to use an interpreter. All of these factors are cause to think that a compiler that focuses on speed of compilation, rather than the speed of the produced code, could be valuable.

Cranelift's first use was as the backend of Wasmtime's just-in-time (JIT) compiler. Many languages now come equipped with JIT compilers, which often use specialized tricks to quickly compile isolated functions. For example, Python recently added a copy-and-patch JIT that works by taking pre-compiled sections of code for each Python bytecode and stitching them together in memory. JIT compilers often use techniques, such as speculative optimizations, that make it difficult to reuse the compiler outside its original context, since they encode so many assumptions about the specific language for which they were designed.

The developers of Cranelift chose to use a more generic architecture, which means that Cranelift is usable outside of the confines of WebAssembly. The project was originally designed with use in Wasmtime, Rust, and Firefox's SpiderMonkey JavaScript interpreter in mind. The SpiderMonkey project has since decided against using Cranelift for now, but the Cranelift project still has a design intended for easy incorporation into other programs.

Cranelift takes in a custom intermediate representation called CLIF, and directly emits machine code for the target architecture. Unlike many other JIT compilers, Cranelift does not generate code that relies on being able to fall back to using an interpreter in case an assumption is invalidated. That makes it suitable for adopting into non-WebAssembly-related projects.

Cranelift's optimizations

Despite its focus on fast code generation, Cranelift does optimize the code it generates in several ways. Cranelift's optimization pipeline is based on equality graphs (or E-graphs), a data structure for representing sets of equivalent intermediate representations efficiently. In a

traditional compiler, the optimizer works by taking the representation of the program produced by parsing and then applying a series of passes to it to produce an optimized version. The order in which optimization passes are performed can have a large impact on the quality of code produced, since some passes require simplifications made by other passes in order to apply. Choosing the correct order in which to apply optimizations is called the phase-ordering problem, and has been the source of a considerable amount of academic research.

In Cranelift, the part of each optimization that recognizes a simpler or faster way to represent a particular construct is separated from the part that chooses what representation should ultimately be used. Each optimization works by finding a particular pattern in the internal representation, and then annotating it as being equivalent to some simplified version. The E-graph data structure represents this efficiently, by allowing two copies of the internal representation to share the nodes that they have in common, and to allow nodes in CLIF to refer to equivalency classes of other nodes, instead of referring to specific other nodes. This produces a dense structure in which adding an alternate form of a particular section of the program is cheap.

Because optimizations run on an E-graph only add information in the form of new annotations, the order of the optimizations does not change the result. As long as the compiler continues running optimizations until they no longer have any new matches (a process known as equality saturation), the E-graph will contain the representation that would have been produced by the optimal ordering of an equivalent sequence of traditional optimization passes — along with many less efficient representations. E-graphs are more efficient than directly storing every possible alternative (taking O(log n) space on average), but they still take more memory than a traditional intermediate representation. Depending on the program in question and the set of optimizations employed, a fully saturated E-graph could be arbitrarily large. In practice, Cranelift sets a limit on how many operations are performed on the graph to prevent it from becoming too large.

E-graphs pay for this simplicity and optimality when it comes time to extract the final representation from the E-graph to use for code generation. Extracting the fastest representation from an E-graph is an NP-complete problem. Cranelift uses a set of heuristics to quickly extract a good-enough representation.

Trading one NP-complete problem (selecting the best order for a set of passes) for another may not seem like a large benefit, but it does make sense for a smaller project. The order of optimization passes is largely set by the programmers who write the optimizations, because it requires domain knowledge to pick a reasonable sequence. Extracting an efficient representation from an E-graph, on the other hand, is a generic search problem that can have as much or as little computer time applied to it as the application permits. Cranelift's heuristics don't extract the most efficient representation, but they do a good job of quickly extracting a decent one.

Representing optimizations in this way also makes it easier for Cranelift maintainers to understand and debug existing optimizations and their effects, and makes writing new optimizations somewhat simpler. Cranelift has a custom domain-specific language (ISLE) that is used internally to specify optimizations.

While Cranelift does not organize its optimizations in phases, it does have ten different sets of related optimizations defined in their own ISLE files, which allows for a rough comparison with GCC and LLVM. LLVM lists 96 optimization passes in its documentation, while GCC has 372. The optimizations that Cranelift does have include constant propagation, bit operation simplifications, vectorization, floating-point operation optimizations, and normalization of

comparisons. Dead-code elimination is done implicitly by extracting a representation from the E-graph.

A paper from 2020 showed that Cranelift was an order of magnitude faster than LLVM, while producing code that was approximately twice as slow on some benchmarks. Cranelift was still slower than the paper's authors' custom copy-and-patch JIT compiler, however.

Cranelift for Rust

Cranelift may have been designed with the aim of being an alternate backend for Rust, but actually making it usable has taken significant effort. The Rust compiler has an internal representation (IR) called mid-level IR that it uses to represent type-checked programs. Normally, the compiler converts this to LLVM IR before sending it to the LLVM code-generation backend. In order to use Cranelift, the compiler needed another library that takes mid-level IR and emits CLIF.

That library was largely written by "bjorn3", a Rust compiler team member who contributed more than 3,000 of the approximately 4,000 commits to Rust's Cranelift backend. He wrote a series of progress reports detailing his work. Development began in 2018, and kept pace with Rust's own rapid development. In 2023, the backend was considered stable enough to ship as part of Rust nightly as an optional toolchain component.

People can now try the Cranelift backend using `rustup` and `cargo`:

```
$ rustup component add rustc-codegen-cranelift-preview --toolchain nightly
$ export CARGO_PROFILE_DEV_CODEGEN_BACKEND=cranelift
$ cargo +nightly build -Zcodegen-backend
```

The given `rustup` command adds the Cranelift backend's dynamic library to the set of toolchain components to download and keep up to date locally. Setting the `CARGO_PROFILE_DEV_CODEGEN_BACKEND` environment variable instructs `cargo` to use Cranelift for debug builds, and the final `cargo` invocation builds whatever Rust project lives in the current directory with the alternate code-generation backend feature turned on. The latest progress report from bjorn3 includes additional details on how to configure Cargo to use the new backend by default, without an elaborate command-line dance.

Cranelift is itself written in Rust, making it possible to use as a benchmark to compare itself to LLVM. A full debug build of Cranelift itself using the Cranelift backend took 29.6 seconds on my computer, compared to 37.5 with LLVM (a reduction in wall-clock time of 20%). Those wall-clock times don't tell the full story, however, because of parallelism in the build system. Compiling with Cranelift took 125 CPU-seconds, whereas LLVM took 211 CPU-seconds, a difference of 40%. Incremental builds — rebuilding only Cranelift itself, and none of its dependencies — were faster with both backends. 66ms of CPU time compared to 90ms.

Whether Cranelift will ameliorate users' concerns about slow compile times in Rust remains to be seen, but the initial signs are promising. In any case, Cranelift is an interesting showcase of a different approach to compiler design.
