Rust and LLVM in 2021
Progress and Challenges in Code Generation
Personal Background

- Have been working on Rust and with Rust since before its release in 2010
- Am a Rust core team alumnus
- Wrote the initial LLVM-based code generator for Rust, as well as the first self-hosting version of the typechecker and name resolver
- Have done lots of work with Rust (graphics, concurrency, etc.) as well as on the compiler itself
- Was formerly at Mozilla, now at Facebook
- Have been working on and off with LLVM for over a decade now
Agenda

- New Features
- Improvements and Fixes
- Future Challenges
- Wrap-up
New Features

Bringing LLVM Enhancements to Rust
Stack Clash Protection
Background

• *Stack Clash* was a 2017 attack that defeated guard pages with allocations so large that they jumped the guard page

• GCC had implemented a defense; Clang/LLVM didn’t have such a feature at the time

• Rust already had a partial countermeasure on x86 that used LLVM’s segmented stack feature to implement stack probes, but this was inelegant
Stack Clash Protection
Better Stack Probes

• Rust worked with upstream to implement the feature properly in LLVM and Clang
• Led to the discovery that stack alignment requirements could jump guard pages as well
• Firefox’s test suite was used to verify
• Clang and Rust are now using the new inline stack probes feature, replacing the old __rust_probestack
ThinLTO

Problems With Monolithic LTO

• Link-time optimization brings significant runtime performance benefits
• But traditional monolithic LTO has operated by combining all compilation units into a single LLVM module and optimizing that
  • This presents serious scalability problems
    • Not all passes are linear-time
    • Memory usage explodes
• Few packages used LTO in practice, despite Cargo (build system) support
ThinLTO
CodeGen Units

- As a separate but related problem, Rust compilation units tend to be very large
  - The compilation model has traditionally been entire-package-at-a-time
  - All `.rs` files that make up a crate (package) are concatenated into one LLVM module
- To work around the resulting compile time problems, Rust has a `codegen units` feature in which the Rust compiler automatically divides up a crate into smaller LLVM modules and compiles them in parallel
  - Sacrifices interprocedural optimization opportunities
The solution to both problems: ThinLTO, introduced in 2016

- LLVM emits compact summaries of each module on the side to perform global interprocedural optimizations, without needing to parse bitcode
- Only functions likely to be inlined are imported into other modules
- Rust adopted ThinLTO to make codegen units feature more viable for production builds, not just debug builds as it was previously
- Shipped alongside Rust incremental compilation, which caches compilation artifacts on a fine-grained level
Profile-Guided Optimization

Basic Use

• The Rust compiler has full support for LLVM’s profile-guided optimization
• The compile flags are modeled after those of Clang
• Rust packages LLVM tools so that they can be installed just like the compiler
• Example of use:
  
  $ rustc -Cprofile-generate=/tmp/pgo-data -O ./main.rs
  $ ./main mydata1.csv
  $ llvm-profdata merge -o ./merged.profdata /tmp/pgo-data
  $ rustc -Cprofile-use=./merged.profdata -O ./main.rs
Profile-Guided Optimization
Application to the Rust Compiler

• Idea: Why not use PGO on the Rust compiler itself?
  • Compile time is very important to Rust
  • Clang saw improvements of up to 20% using PGO
• rustc is written in Rust, so enabling PGO for LLVM and enabling PGO for the Rust compiler are two separate problems
  • Fortunately, the same version of LLVM need not be used for the C++ compiler and Rust compiler, as long as we’re careful
Profile-Guided Optimization
Rust Compiler Results

- 10%-16% reduction of build times
- Probably dominated by improved cache effects
- Unfortunately, PGO on rustc may be difficult to deploy
- Rust development moves too quickly to build with PGO in continuous integration

<table>
<thead>
<tr>
<th>Cargo-Opt</th>
<th>Avg: -16.7%</th>
<th>Min: -16.7%</th>
<th>Max: -16.7%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Futures-Opt</td>
<td>Avg: -16.7%</td>
<td>Min: -16.7%</td>
<td>Max: -16.7%</td>
</tr>
<tr>
<td>Regex-Opt</td>
<td>Avg: -16.6%</td>
<td>Min: -16.6%</td>
<td>Max: -16.6%</td>
</tr>
<tr>
<td>Serde-Opt</td>
<td>Avg: -16.5%</td>
<td>Min: -16.5%</td>
<td>Max: -16.5%</td>
</tr>
<tr>
<td>Packed-Simd-Check</td>
<td>Avg: -16.4%</td>
<td>Min: -16.4%</td>
<td>Max: -16.4%</td>
</tr>
<tr>
<td>Serde-Debug</td>
<td>Avg: -16.4%</td>
<td>Min: -16.4%</td>
<td>Max: -16.4%</td>
</tr>
<tr>
<td>Regex-Debug</td>
<td>Avg: -16.4%</td>
<td>Min: -16.4%</td>
<td>Max: -16.4%</td>
</tr>
<tr>
<td>Serde-Check</td>
<td>Avg: -16.3%</td>
<td>Min: -16.3%</td>
<td>Max: -16.3%</td>
</tr>
<tr>
<td>Piston-Image-Opt</td>
<td>Avg: -16.3%</td>
<td>Min: -16.3%</td>
<td>Max: -16.3%</td>
</tr>
<tr>
<td>Regression-31157-Debug</td>
<td>Avg: -16.3%</td>
<td>Min: -16.3%</td>
<td>Max: -16.3%</td>
</tr>
<tr>
<td>Ripgrep-Check</td>
<td>Avg: -16.2%</td>
<td>Min: -16.2%</td>
<td>Max: -16.2%</td>
</tr>
<tr>
<td>Regex-Check</td>
<td>Avg: -16.0%</td>
<td>Min: -16.0%</td>
<td>Max: -16.0%</td>
</tr>
<tr>
<td>Futures-Check</td>
<td>Avg: -16.0%</td>
<td>Min: -16.0%</td>
<td>Max: -16.0%</td>
</tr>
<tr>
<td>Syn-Debug</td>
<td>Avg: -15.9%</td>
<td>Min: -15.9%</td>
<td>Max: -15.9%</td>
</tr>
<tr>
<td>Futures-Debug</td>
<td>Avg: -15.9%</td>
<td>Min: -15.9%</td>
<td>Max: -15.9%</td>
</tr>
<tr>
<td>Clap-Rs-Opt</td>
<td>Avg: -15.8%</td>
<td>Min: -15.8%</td>
<td>Max: -15.8%</td>
</tr>
<tr>
<td>Syn-Check</td>
<td>Avg: -15.8%</td>
<td>Min: -15.8%</td>
<td>Max: -15.8%</td>
</tr>
<tr>
<td>Encoding-Check</td>
<td>Avg: -15.7%</td>
<td>Min: -15.7%</td>
<td>Max: -15.7%</td>
</tr>
<tr>
<td>HelloWorld-Check</td>
<td>Avg: -3.3%</td>
<td>Min: -3.3%</td>
<td>Max: -3.3%</td>
</tr>
<tr>
<td>Unify-Linearly-Debug</td>
<td>Avg: -3.3%</td>
<td>Min: -3.3%</td>
<td>Max: -3.3%</td>
</tr>
<tr>
<td>Helloworld-Opt</td>
<td>Avg: -2.9%</td>
<td>Min: -2.9%</td>
<td>Max: -2.9%</td>
</tr>
<tr>
<td>Helloworld-Debug</td>
<td>Avg: -2.6%</td>
<td>Min: -2.6%</td>
<td>Max: -2.6%</td>
</tr>
<tr>
<td>Token-Stream-Stress-Debug</td>
<td>Avg: -2.2%</td>
<td>Min: -2.2%</td>
<td>Max: -2.2%</td>
</tr>
</tbody>
</table>
Source-Based Code Coverage
Overview

• Clang has long since had a source-based code coverage feature
  • Implemented via the *InstrProf* feature in LLVM
  • `llvm-profdata` and `llvm-cov` tools
• Instrumentation data generated in the front-end, on the MIR intermediate representation
• Allows for precise code coverage measurement at a granularity smaller than a single source line
Improvements and Fixes
Improving LLVM for Everyone
Infinite Loops

The Problem

• Very roughly, infinite loops (\texttt{\{\} and similar in Rust) are undefined behavior in versions of LLVM prior to LLVM 12
  
  • Rust must have no undefined behavior absent \texttt{unsafe}, so this is a problem

• Rust landed a workaround for trivial cases in 2020 (PR \#77972), but it failed to catch anything but trivial cases out of compile time concerns
  
  • A short test case: (0..).\texttt{sum()} raised SIGILL

• This has been the subject of several LLVM mailing list threads over the years, with no clear consensus on what to do until 2020
Infinite Loops
Old LLVM Forward Progress Semantics

• Prior to LLVM 12, LLVM was allowed to assume that any thread will eventually do one of the following:
  1. Terminate.
  2. Make a call to a library I/O function.
  3. Access or modify a volatile object.
  4. Perform a synchronization operation or an atomic operation.
• Models C++‘s forward progress assumption
In 2019, LLVM introduced a new function attribute, `willreturn`:
- Indicates that a function must return eventually.

And in 2021, LLVM introduced a related attribute, `mustprogress`:
- Indicates that the function follows C++ forward progress requirements.
- Unlike C++, Rust imposes no obligations on functions to return or make forward progress, so Rust never uses either of these two tags.
- Thus LLVM 12 should fix the problem automatically.
Aliasing Guarantees
Rust Pointer Types

- Rust has a very different set of pointer semantics than C, C++, or even managed languages do.
- To a first approximation, there are three kinds of pointers:
  1. *Immutable reference* (&T)—value can be freely aliased but is immutable.
     - Different from C\texttt{const T*}—referent is immutable for the lifetime of the pointer.
  2. *Mutable reference* (&mut T)—pointer has exclusive access, like C\texttt{restrict}.
  3. *Unsafe reference* (*T*)—no restrictions, not even “strict-aliasing” TBAA.
• Rust references on function arguments would seem to map fairly cleanly to LLVM function argument attributes
  • Immutable references map to `readonly`
  • Mutable references would seem to map to `noalias`, but in practice this has exposed miscompilations in LLVM
    • Use of `noalias` was disabled in `rustc` in 2018
    • LLVM fixes landed in 2021; plan is to reenable in `rustc` soon
    • Rust developers helped to isolate and prepare patches for the primary issue here, related to loop unrolling
Managing Regressions
Finding and Fixing LLVM Bugs

- Rust has surfaced some correctness and compile-time regressions that have been fixed
  - Enabled use of MemorySSA by default in `memcpy` optimization
  - Correctness fixes to loop strength reduction, alias analysis, scalar evolution, `memcpy` optimization
  - Compile time improvements to scalar evolution and instruction combining
- Nikita Popov has been driving much of this work
Missed Optimizations
Improvements to Upstream LLVM

• Rust has also helped find and fix missed optimizations in LLVM
• Examples with Rust issue numbers:
  • #48627—missed constant folding opportunity (fixed by Nikita)
  • #73827—missed bounds check optimization (also fixed by Nikita)
  • #74938—another missed bounds check optimization (fixed by Xavier Denis)
Future Challenges
Opportunities Going Forward
Richer Aliasing Guarantees

Background

• As mentioned before, Rust potentially has strong guarantees around aliasing
• However, the details are tricky
  • How, and in which scope, do unsafe pointers interact with references?
  • Is type-based alias analysis applicable to Rust?
  • What do Rust programmers intuitively expect? Will their code break if we implement aggressive optimizations?
Richer Aliasing Guarantees
Potential Future Opportunities

• The Rust compiler doesn’t currently use the full set of LLVM alias metadata available to it, e.g. on load and store instructions
  • Miscompilations may happen if this infrastructure is relied on more
  • Are these optimizations best done in rustc or in LLVM?
    • Few languages have as strong guarantees around memory as Rust has
    • May be difficult to implement in LLVM’s type system
    • Even if implementable, compile time concerns may make some optimizations not worthwhile in LLVM
Compile Time
An Ongoing Challenge

• Compile times remain a challenge, especially at -O0.
  • GlobalISel may help here
  • Historically we haven’t been able to use FastISel much due to exceptions
• Alternate backends are being explored for debug mode
• However, LLVM is not that easy to outperform
Minor Platforms

Three Issues

Three simultaneous facts create an interesting situation:

1. As Rust gets more uptake, other open source projects are starting to take it on as a dependency
   - Examples: Firefox, librsvg, Python cryptography package

2. Rust currently has only one major complete implementation, which depends on LLVM
   - Though mrustc is close

3. LLVM is fairly conservative about adding support for new architectures
Minor Platforms
Linux Distro Impact

- Rust has brought out some difficult decisions for distros that support less popular architectures unsupported by LLVM
- Alpine Linux, Gentoo
- DEC Alpha, HP PA-RISC, Intel Itanium, IBM System/390
Minor Platforms
Upstreaming New Architectures

• Interesting outcome: Rust specifically was the motivation to get the out-of-tree Motorola 68000 backend upstream

• John Paul Adrian Glaubitz has been doing the work

• Looks promising so far
Wrapping Up

• LLVM has served Rust’s needs well over the years
• In fact, it’s been key to Rust’s success
• Working with upstream LLVM has been part of the Rust development culture from the beginning
• We’re looking forward to continued collaboration in the future!
Thank you!

Patrick Walton
@pcwalton
Fifth LLVM Performance Workshop at CGO
February 28, 2021

Artist credits:
“Ferris and the LLVM Wyvern”—Anonymous, used with permission