Distributing PGO'ed Toolchains For Great Good! 🎉

gburgessiv
(he/him)

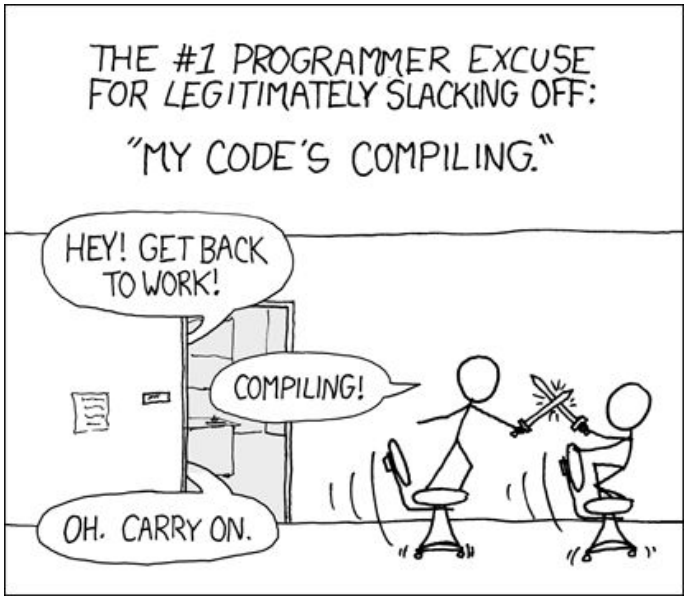
Intro

- I like it when developers are happy
  - They sometimes feel more fulfilled
  - They sometimes use smiley faces in their messages :)
  - They sometimes merge my PRs
- What makes developers unhappy?
  - A lot

Intro

- I like it when developers are happy
  - They sometimes feel more fulfilled
  - They sometimes use smiley faces in their messages :)
  - They sometimes merge my PRs
- What makes developers unhappy ***and is within our scope***?
  - A lot, but also slow builds

Proposal: make builds go faster 🏎️🔥



Spoilers

- In general, PGO + ThinLTO = ~13%-25% build time improvements
  - (In both wall and total CPU time, since `--jobs N` is a thing)
  - (Most often, we expect in the 15-20% perf improvement range)

What is ThinLTO?

- Does whole-program optimizations on C, C++, etc code
  - Inlining and optimization across C/C++ files (not just headers!)
  - Can optimize some `virtual` trickery in C++
- LTO but Scalable™
  - (No, really, it's much faster on a machine with more than one core)

How does one ThinLTO in general?

- Make sure you're using a toolchain which fully supports ThinLTO
  - e.g., Clang with LLD as its linker
  - `CPPFLAGS+=-flto=thin` + `LDFLAGS+=-flto=thin`
- 🪄 You're done ✨

What is PGO?

- Many optimizations rely on guesses about how code will execute to be effective
- These guesses are generally correct
- PGO: "Why make the compiler guess when you can simply *tell it*?"

How do I PGO in general?

- Assuming you want to PGO `${binary}`:
  - Build `${binary}` with instrumentation enabled (`CPPFLAGS+=-fprofile-generate`)
  - Run `${binary}` on a "representative" workload
  - Postprocess the profiles that `${binary}` generated (using `llvm-profdata`)
  - Build `${binary}` with the profile applied (`CPPFLAGS+=-fprofile-use=/path/to/pro.file`)
- 🪄 You're done ✨

But wait, there's more!

Clang makes this all easier

- ThinLTO: `cmake ... -DLLVM_ENABLE_LTO=Thin ...`
- PGO has a few hammers:
  - Profile collection:
    - llvm/utils/collect_and_build_with_pgo.py
    - stage2 PGO with cmake
    - If you want to build a profile, `cmake ... -DLLVM_BUILD_INSTRUMENTED=${profile_type} ...`
  - Profile application:
    - `cmake ... -DLLVM_PROFDATA_FILE=/path/to/profile.prof ...`

One last bit

- How do I find a "representative workload"?

What does Google do for it?

- Different teams do different things
  - Android builds+links a few medium-sized targets in their codebase
    - Varies between ThinLTO/vanilla linking
    - Varies between host (x86-64) and target (arm32/64) binaries
  - Chrome OS builds+links all of Chrome
  - Chrome builds a single 11MB `.ii` file
  - Server toolchain builds a few small targets + protobuf files
- All of these approaches seem quite successful

So... How do I pick the benchmarks to train on?

- Good enough is good enough
  - If you have known-hot targets and want to put effort in, great
  - If not, `ninja check-clang check-llvm` is good enough
    - 15% improvement now > 18% improvement never

Summary

- Please consider PGO'ing and ThinLTO'ing your compilers!
  - Faster builds for users, and for your future builds of future compilers 🎉
  - Happier developers worldwide = my PRs are more likely to be merged!!!
- We try to make it easy, and are open to ways to making it easier
  - For now, it's just a few different commands

Bonus: combining optimizations = number go **more** up 😎

I realize I mixed ThinLTO & PGO for perf numbers. For those interested:

- PGO provides the majority of the benefit (60%-85%)
- ThinLTO is the remainder
- The whole is greater than the sum of its parts
  - ThinLTO, being an optimization, makes smarter decisions with PGO data
  - PGO = 15%
  - ThinLTO = 5%
  - PGO + ThinLTO = 23%
  - 15% + 5% != 23% ($p < 0.05$)

Thanks for your time!