



Exploring Speeding Up a Linux Kernel Build with LLVM Tools and Technologies

Nathan Chancellor, ClangBuiltLinux Meetup
2020



Speaker introduction

- External contributor to ClangBuiltLinux since 2018.
- Started exploring kernels on Android in 2016.
- Started working with Clang on the Pixel 2. Did independent backports to 3.18 for Pixel 1.
- Been interested in toolchains since the beginning (obsessed with being on the latest, fixing warnings, and finding other bugs).



Building LLVM versus GCC

- Building GCC feels convoluted (highly opinionated).
 - autoconf is generally seen as not user friendly.
 - Just building GCC is not too bad but adding in all of the other pieces like binutils, libc, and isl can feel overwhelming.
 - Things like crosstool-ng exist to try and make this simpler but they do not always succeed.
 - For the kernel, Segher Boessenkool's [buildall script](#) works best in my experience.
- Building LLVM is objectively much easier
 - Download and install CMake and ninja and they take it from there.
 - `cmake -G Ninja -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_PROJECTS="clang;compiler-rt;lld" ../llvm && ninja`
 - Adding stages and technologies like LTO and PGO only require extra CMake flags.



Beginning to look at performance

- When initially exploring Clang, research showed that Clang should have been faster than GCC but it wasn't (building with GCC often took 20 seconds less).
- As I understand it, GCC got a lot of improvements to performance in response to Clang and Clang never responded.
- Started reading about compiler optimizations from places like [Gentoo](#).
- Stumbled across a [Chromium revision](#) and [bug report](#) talking about building Chromium's clang with LTO and PGO, thought it would be interesting to see what impact it would make on the kernel.



Link Time Optimization

- Program is compiled into LLVM bitcode then the linker can go through the entire program and perform things like dead code elimination.
- Makes program faster by eliminating unnecessary branches and optimizing memory usage.
- Takes a long time because it is basically compiling the program twice and it uses a ton of memory
- ThinLTO is multithreaded and uses less memory while not sacrificing much optimization. Use ThinLTO if some sort of LTO is required.




Profile Guided Optimization

- Requires an instrumented compiler to generate profiles during run time. You take this compiler, run it against your workload, then use those profiles to build an optimized compiler.
- Typically see this in a three stage configuration
 - Build a small stage 1 compiler to build stage 2
 - Use stage 2 to run your benchmark
 - Use stage 1 to build stage 3 with the profiles from stage 2
- Takes time to run because of the stages plus profile collection BUT the speedup seen at compile time is in the range of 15% to 20%.




`-march=native`

- By default, Clang and GCC will only emit code that is able to run on any processor within the host's instruction set architecture (x86_64, aarch64, etc). "Base" instructions so to say.
- `-march=native` tells the compiler that it is allowed to emit code that will only be able to run on the host processor (or newer).
- Allows the compiler to use things like AVX and SSE for better code generation.
- Often seen as a good quick way to get some sort of performance boost.




Building LLVM with LTO and run time impact

- Building LLVM with LTO is generally done in two stages so that the latest LTO improvements can be used.
- In my benchmarks, building with LTO takes about 7.5x as long as doing a regular two stage build, while building with ThinLTO is only a 3.25x increase. Full LTO is really not worth the extra speed up.
- However, at run time, the kernel builds are only about 3% faster than with a regular two stage compiler ([link](#)).
- Only worth if you are not time constrained.



Building LLVM with PGO and run time impact

- As explained before, generally requires three stages plus time to collect profiles, which is done by just building Linux kernels.
- [tc-build](#) does this automatically with the `--pgo` flag.
- The three stages and profile collecting only results in a 2.5x increase over a regular two stage build.
- Run time performance shows that it is a range of 15% to 18% speed up during run time. Much more visible with LTO ([link](#)).
- When LTO is added to the mix, can go as high as 24%.
- TL;DR: If you are time constrained, just build with PGO. If you can spare the cycles, add ThinLTO into the mix.



Building LLVM with `-march=native` and run time impact

- Results in no visible speed up in the best case, results in 12% performance regression in the worst case ([link](#)).
- Use of AVX-512 causes the performance regression ([LLVM commit dealing with it](#)) so some CPUs are unaffected (like AMD).
- Might be useful in some workloads but it does not appear that just running the compiler to compile the kernel is enough to benefit from these optimizations.



Clang vs. GCC currently

- As of LLVM at f85d63a558364 and GCC at 2d33dcfe9f049, GCC beats a standard LLVM two stage build on all architectures, ranging from 7% (arm64) to 28% (x86_64) [[link](#)].
- PGO alone is enough to get within 10% usually.
- PGO + LTO beats GCC on arm64, arm32, and powerpc32 but fails on powerpc64le and x86_64.
- Some architectures appear to be very optimized in GCC (powerpc64le, x86_64) whereas others are not (arm64).
- Slow down might be due to C++ ? Other architecture changes? GCC is just that good?



Linking the kernel with `ld.lld`

- Two ways to view linking impact: clean builds versus incremental builds.
- Changing one thing then recompiling is mostly the linking because the compiler only runs for the single changed translation unit.
- On clean allyesconfig builds, `ld.lld` results in a:
 - 3% increase on arm32
 - 11% increase on arm64
 - 2% increase on x86_64
- The incremental links are much more telling:
 - 26% increase on arm32
 - 62% increase on arm64(!)
 - 13% increase on x86_64
- [Full results](#)



Questions?

Website: <https://nathanchance.dev>

Github: <https://github.com/nathanchance>

Twitter: <https://twitter.com/nathanchance>