Making LLVM faster with build time optimization technologies

Nathan Chancellor, ClangBuiltLinux Meetup 2023

Current situation

- Most builds of Clang are objectively slower than their GCC counterpart
- Quick test with `ARCH=arm64 virtconfig` with Arch Linux packages
    - Clang 16.0.6: 6m
    - GCC 13.2.0: 4m 30s (25% faster!)
- This adds more friction to getting maintainers to test with Clang because it adds too much to their already potentially lengthy build times (Linus, PeterZ, Boris/Ingo)
- A couple of known reasons
    - Time spent in frontend (lexing, parsing, etc)
    - Dynamic linking against LLVM libraries (for example, Fedora's policy mandates this)

Improving the situation for “free”

- LLVM has several optimization techniques that can be applied at build time to improve the runtime performance of an application.
- FDO/PGO (Feedback-driven Optimization / Profile-guided Optimization): Uses data gathered from running an instrumented application to make optimization decisions at build time.
- LTO (Link Time Optimization): Linker performs optimization based on full program analysis, such as inlining and dead code elimination. LLVM has two types:
  - “full” LTO (serialized but more optimization potential)
  - ThinLTO (parallel)
- BOLT (Binary Optimization and Layout Tool): Similar to FDO/PGO but happens post-link so that the application being optimized is the one that will actually be run.
  - Prefers data from perf / LBR but offers an instrumented mode as well.

Trade Offs

- Certain technologies require more build investment up front.
  - For example, performing PGO on LLVM requires an additional LLVM build stage to get the instrumented compiler and it has to be run against code that it will be building on the regular.
- The up front build cost should be contrasted against the speed ups at run time in combination with number of people potentially benefiting from it.
  - Probably does not make sense to apply all technologies to a toolchain that will only be used by one person infrequently, as the savings at application run time probably does not outweigh the extra application build time cost.
  - However, a minute saved per application build time across fifty developers across a month might be much more significant than the extra compiler build time.
- Requires every distributor to adopt these technologies for everyone to benefit (most developers want to use packages from their distribution, not a standalone tarball)

Brief introduction to tc-build

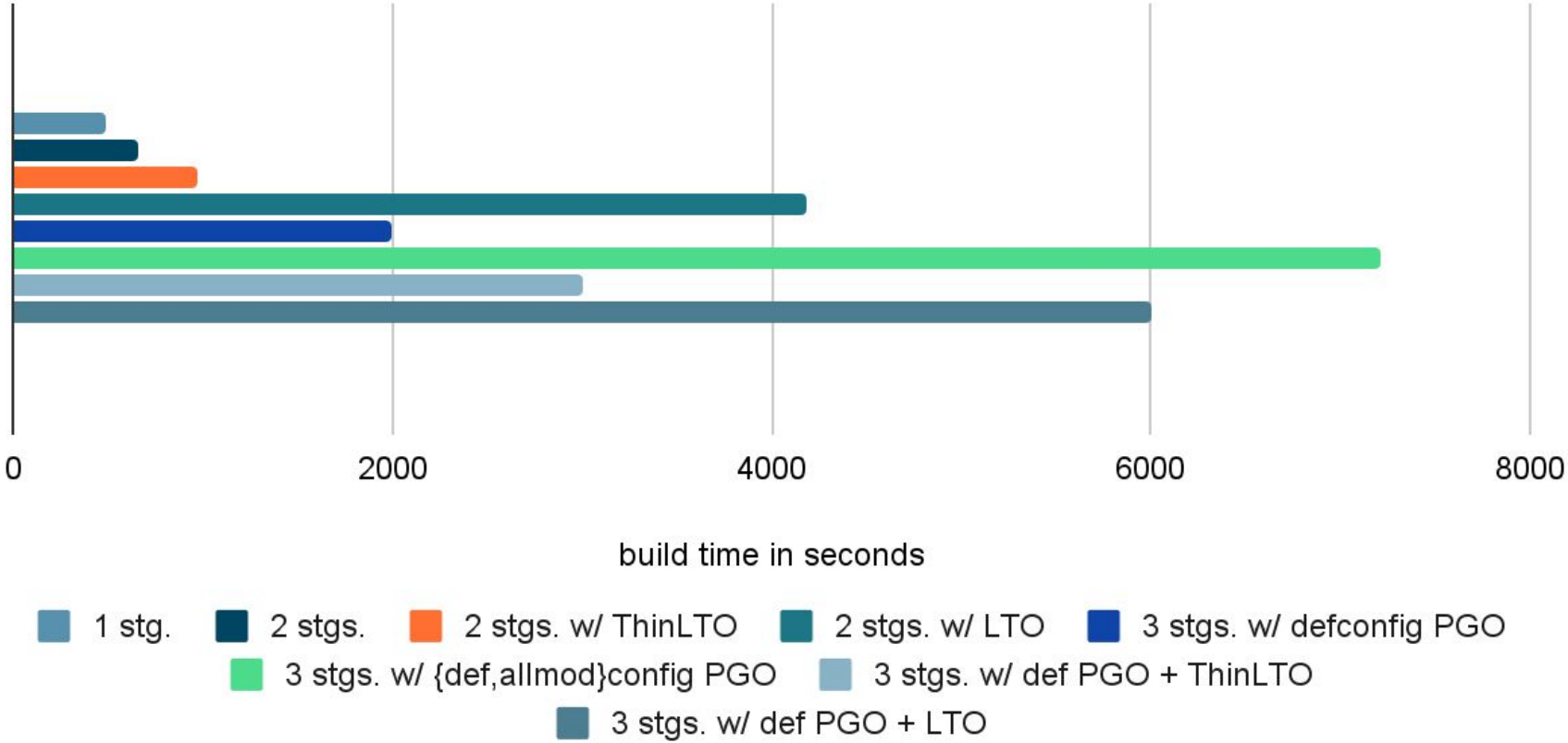
- An LLVM toolchain build script with several options for easily applying these technologies in an automated fashion.
- Can automatically build kernels as PGO benchmark (so building kernels with final toolchain is faster).
- Allows easy testing and comparison of these various technologies.

Benchmarks

- LLVM build benchmarks: Exploring how applying these technologies impacts how long it takes to produce an optimized LLVM toolchain.
  - Only built backends that would be used for profiling, which keeps PGO benchmarking stage slimmer as well
  - Stages are different LLVM builds. By default, tc-build does at least a two stage build (bootstrapping), where the host environment's compiler builds a smaller, host-only toolchain that builds the final toolchain (to take advantage of the most recent compiler improvements).
- Kernel build benchmarks: Exploring how applying these technologies to the toolchain impacts how long it takes to build various Linux kernel configurations.
- Machine configurations:
  - aarch64: Ampere Altra Q80-30 (80 cores @ 3.0GHz), 256GB of RAM, NVMe SSD
  - x86_64: AMD EPYC 7513 (32 cores/64 threads @ 2.6GHz [base], 3.65GHz [boost]), 256GB of RAM, NVMe SSD
- Raw results on GitHub

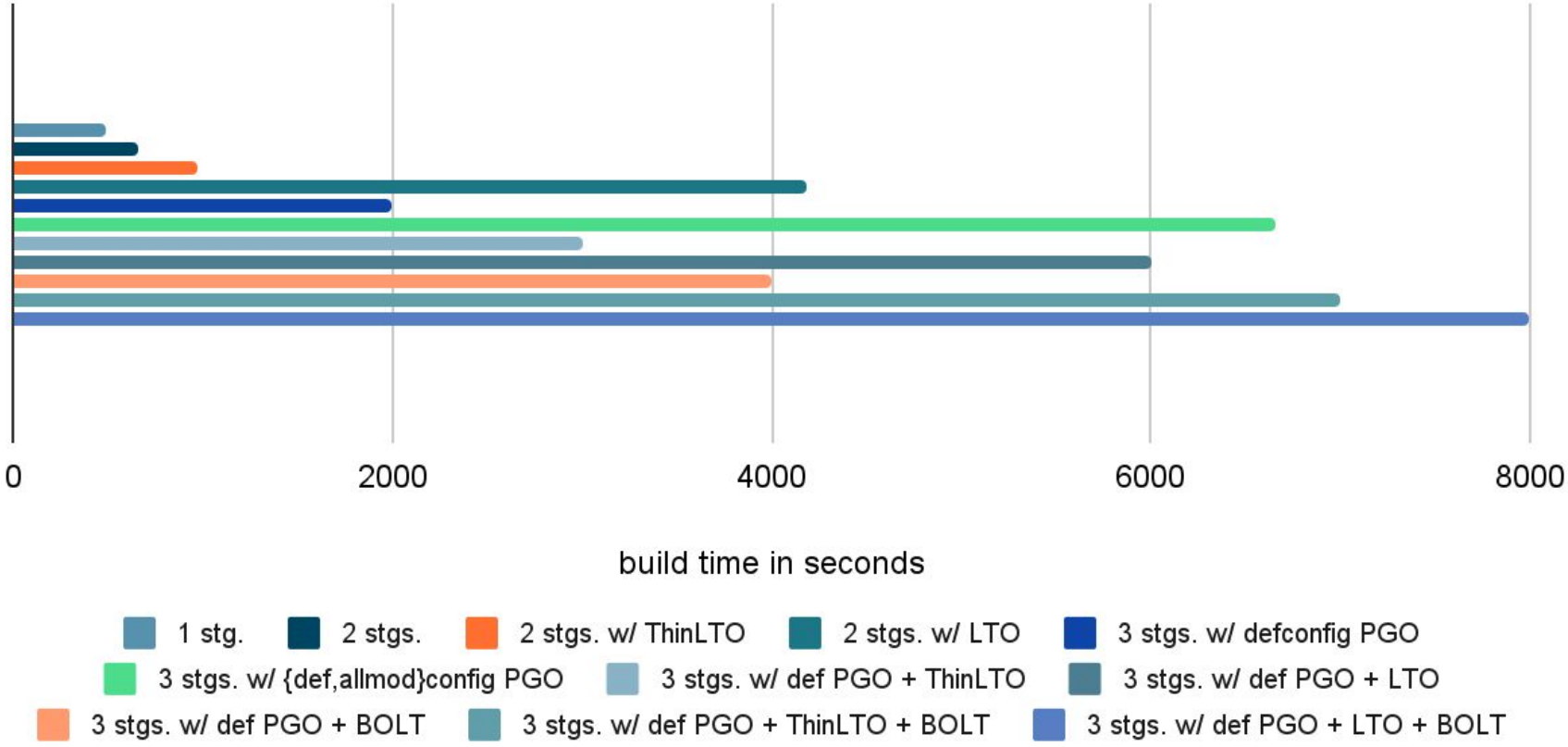
LLVM build times (aarch64)

Averaged across five runs



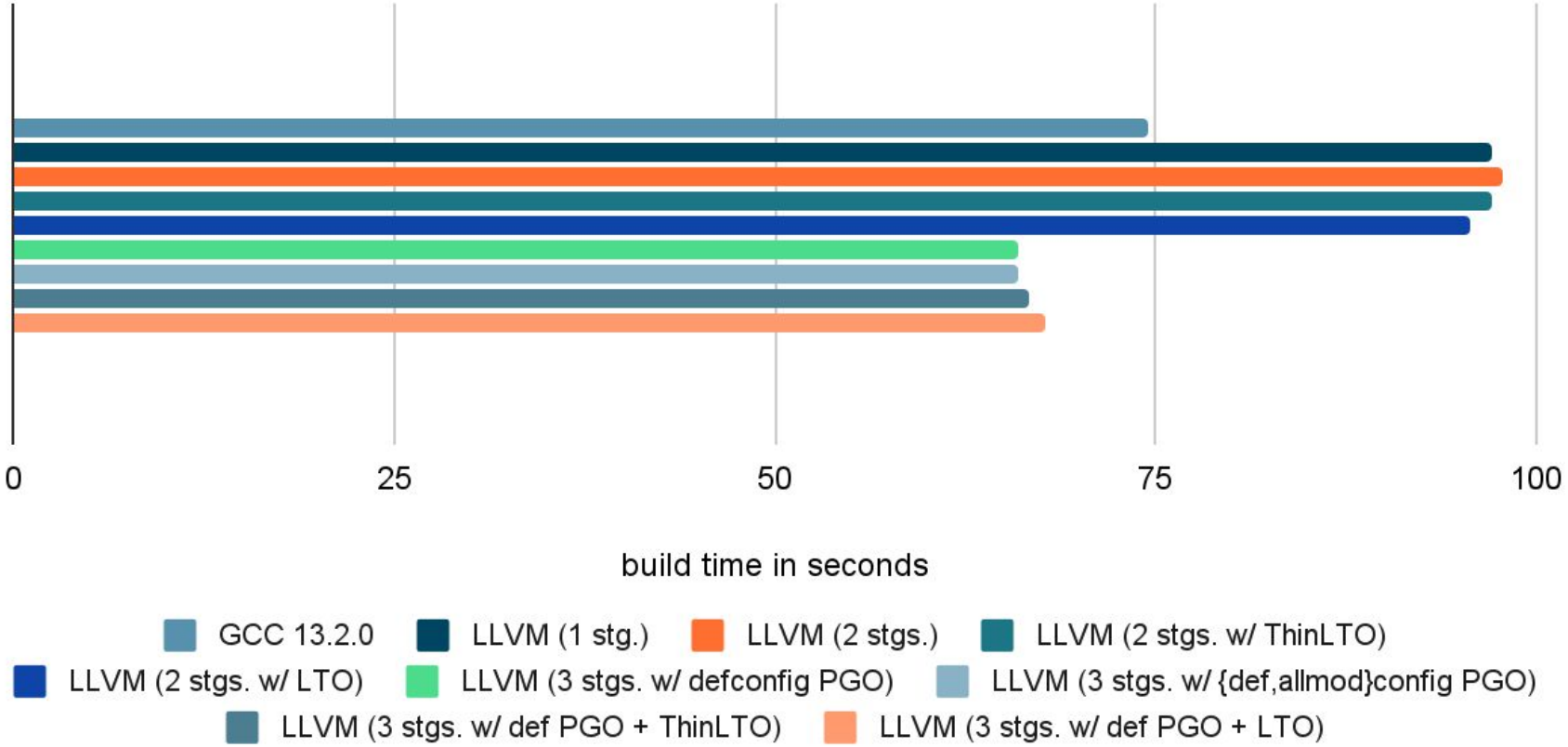
LLVM build times (x86_64)

Averaged across five runs



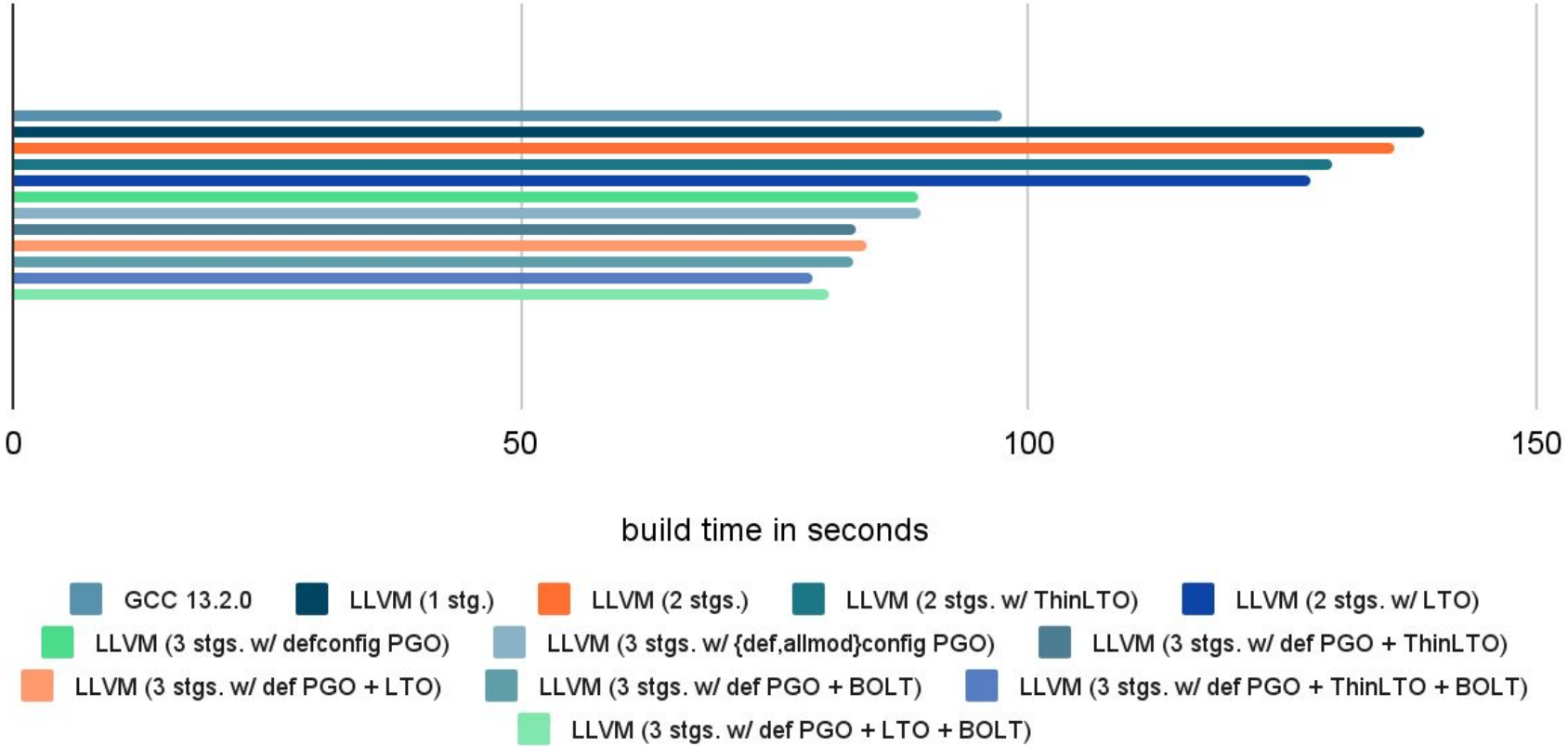
ARCH=arm defconfig (aarch64)

Averaged across ten runs



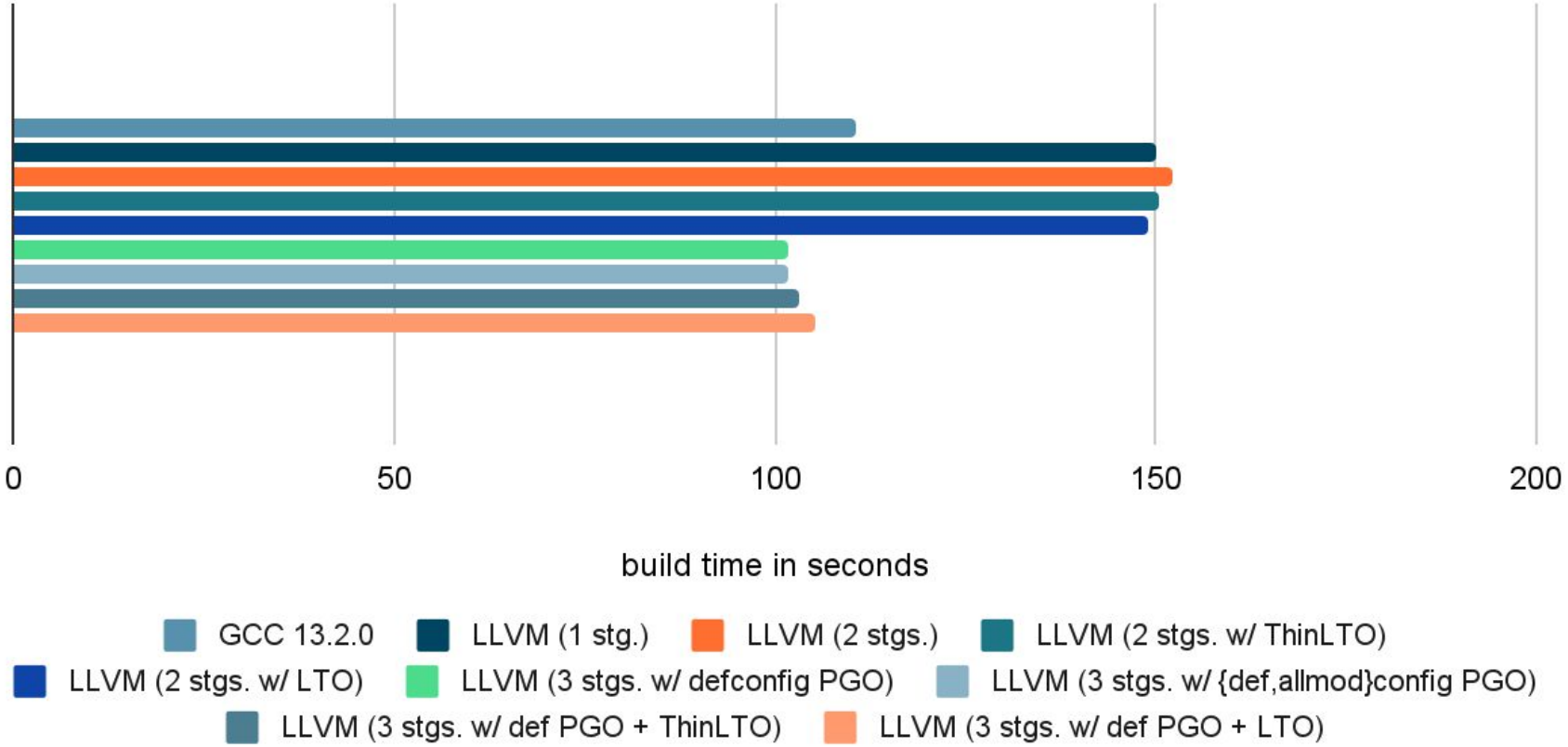
ARCH=arm defconfig (x86_64)

Averaged across ten runs



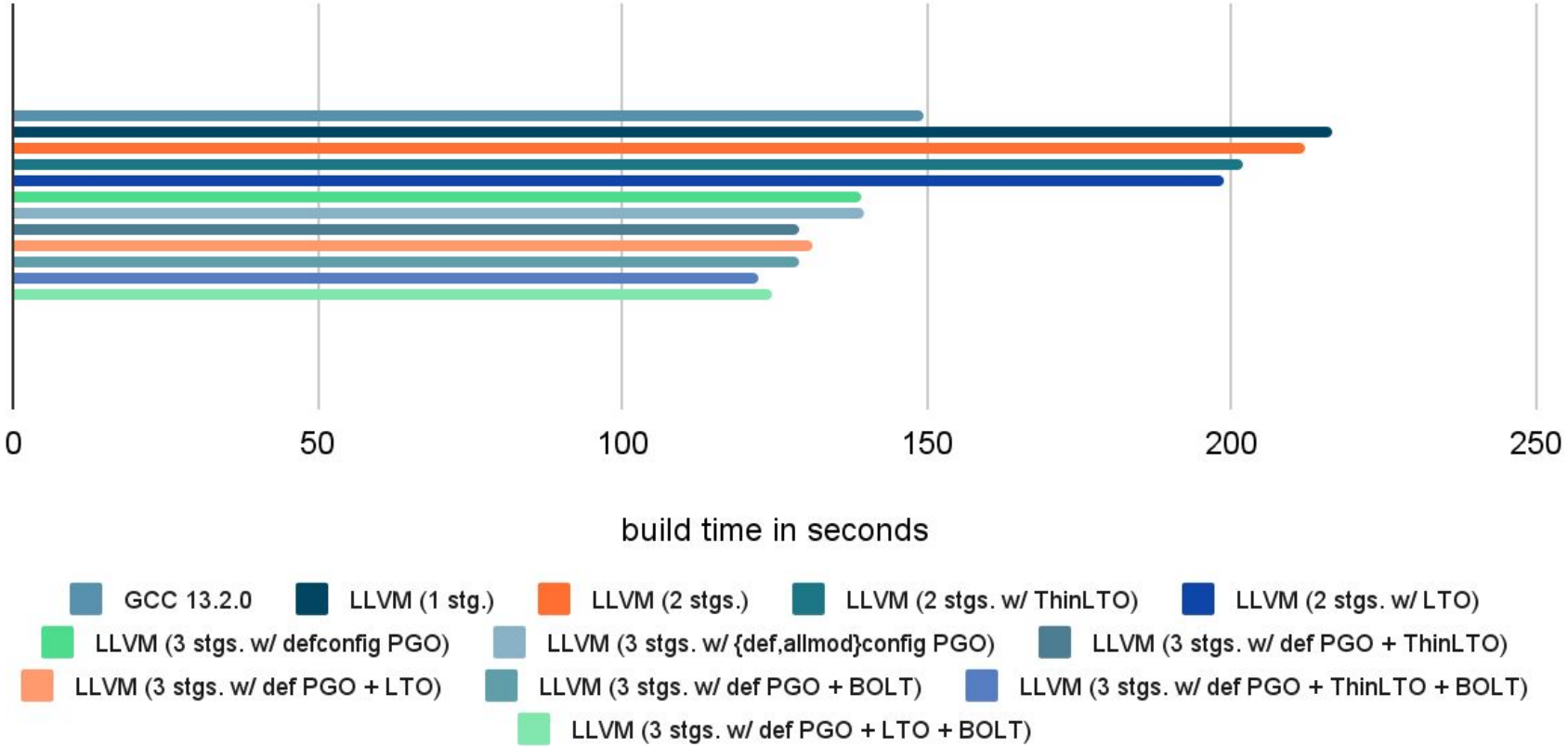
ARCH=arm64 defconfig (aarch64)

Averaged across ten runs



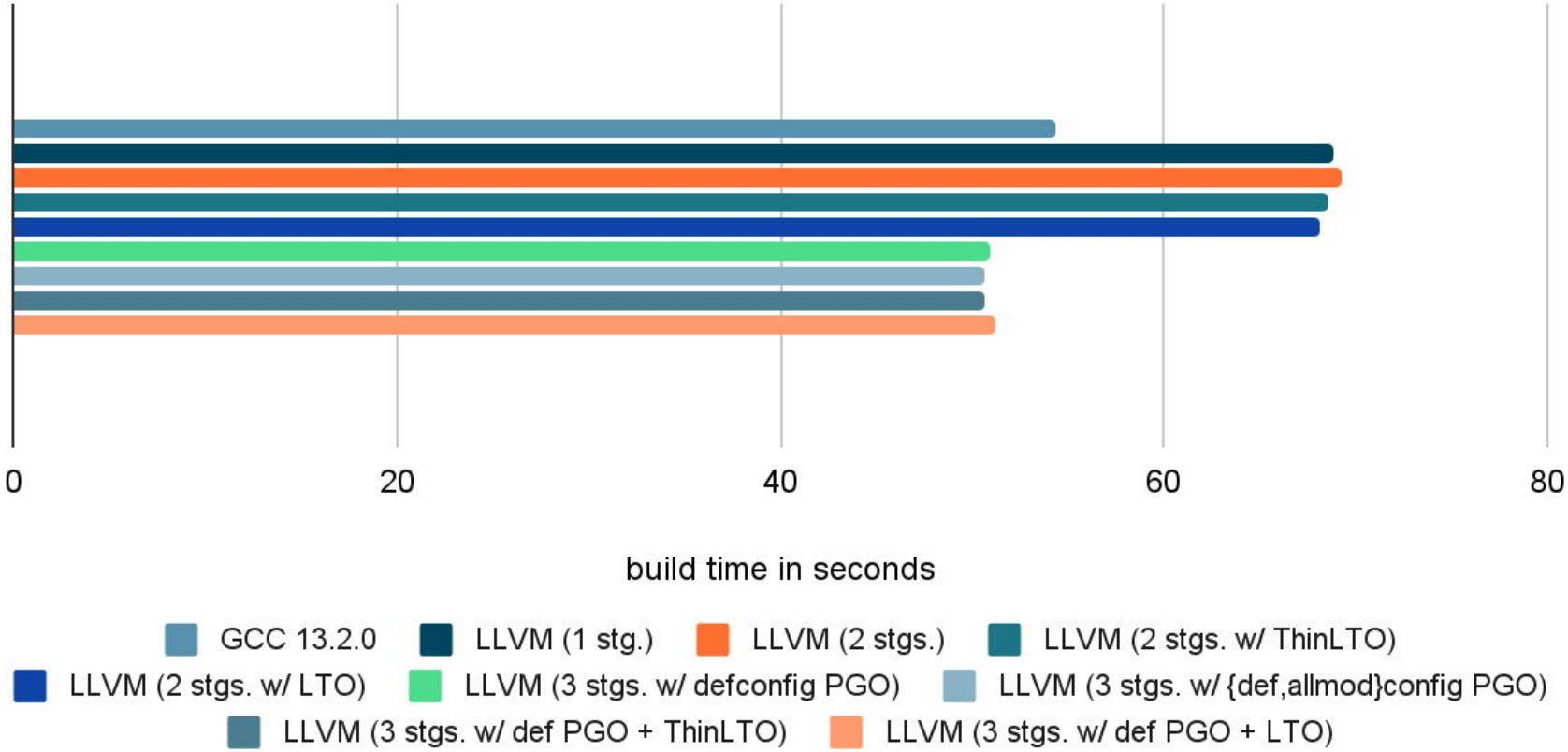
ARCH=arm64 defconfig (x86_64)

Averaged across ten runs



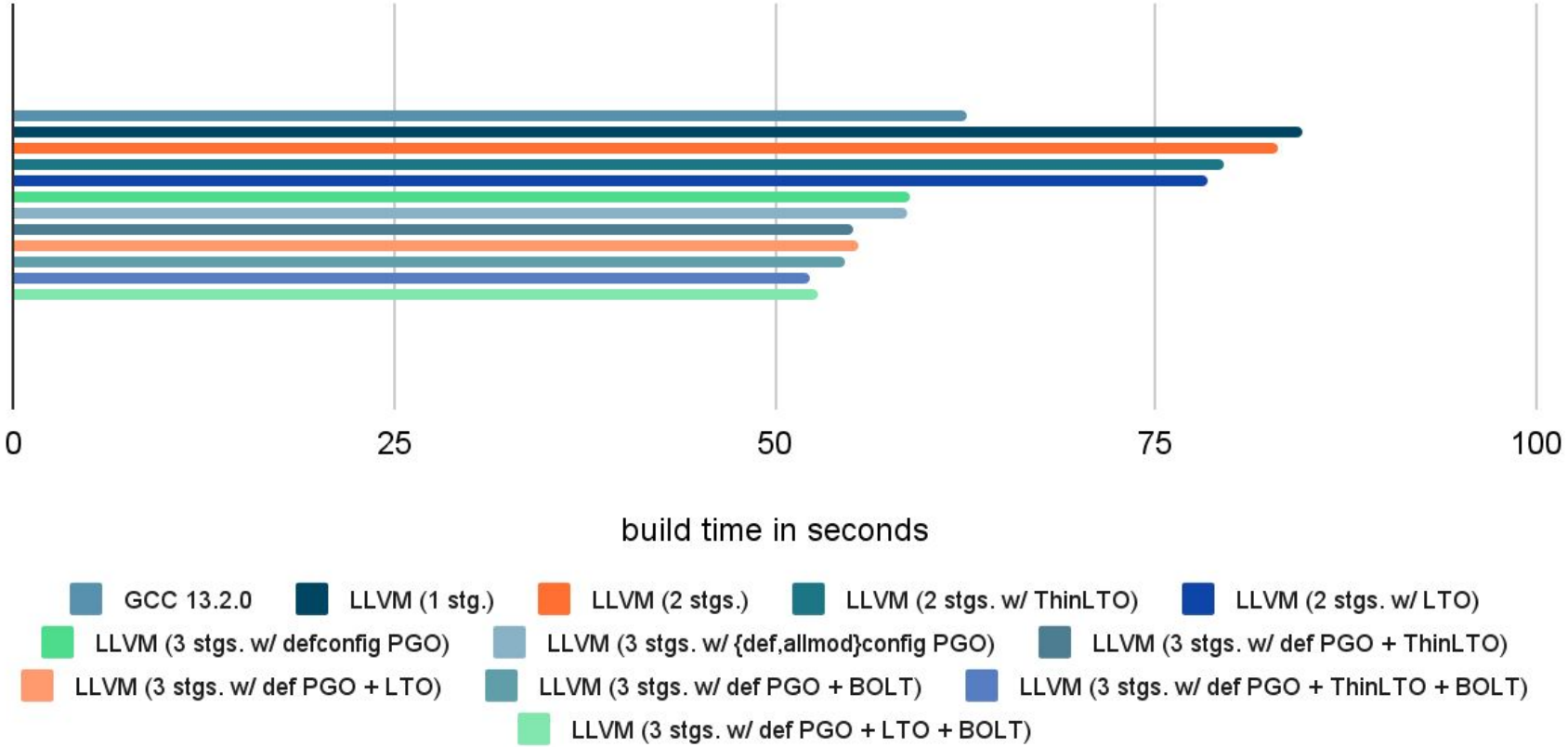
ARCH=x86_64 defconfig (aarch64)

Averaged across ten runs



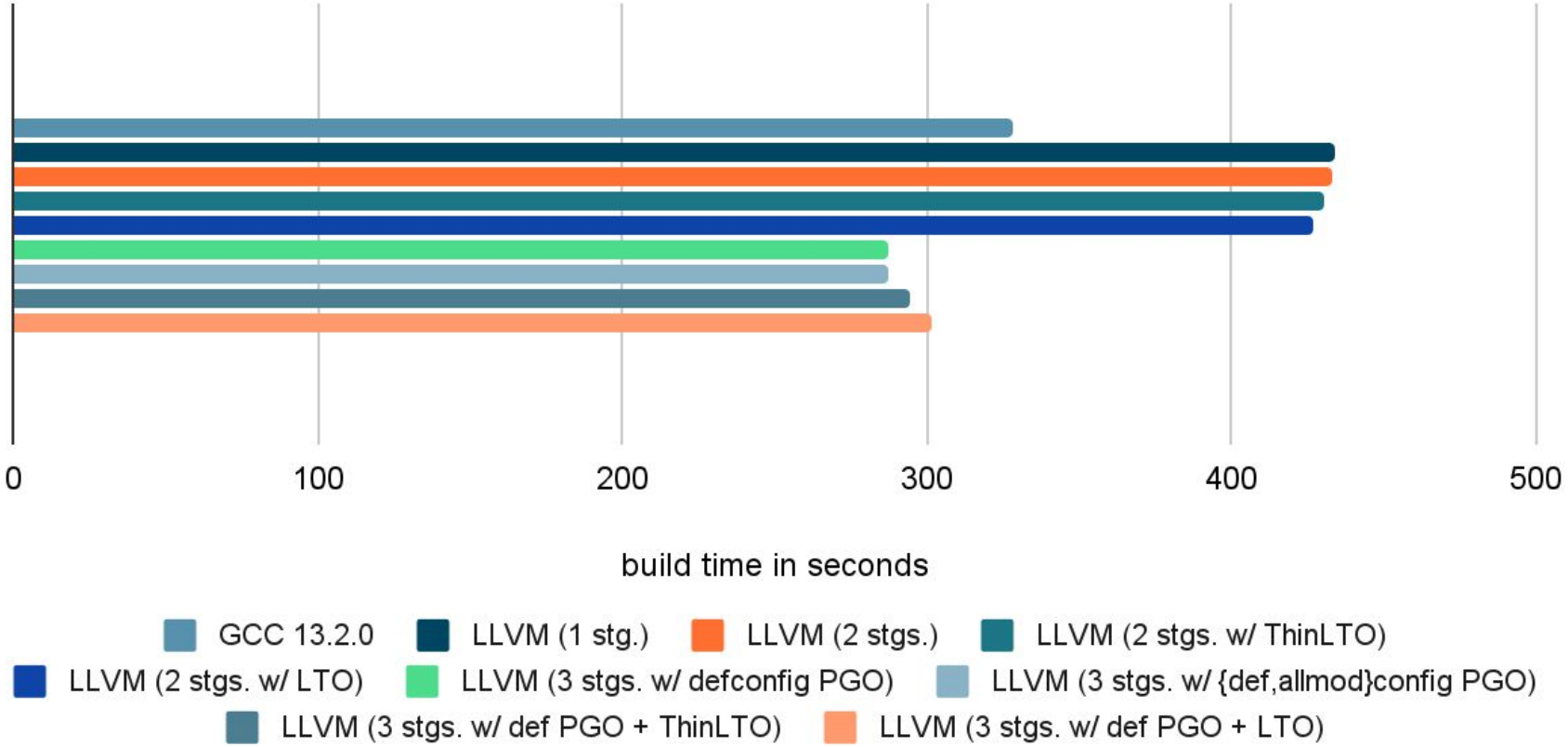
ARCH=x86_64 defconfig (x86_64)

Averaged across ten runs



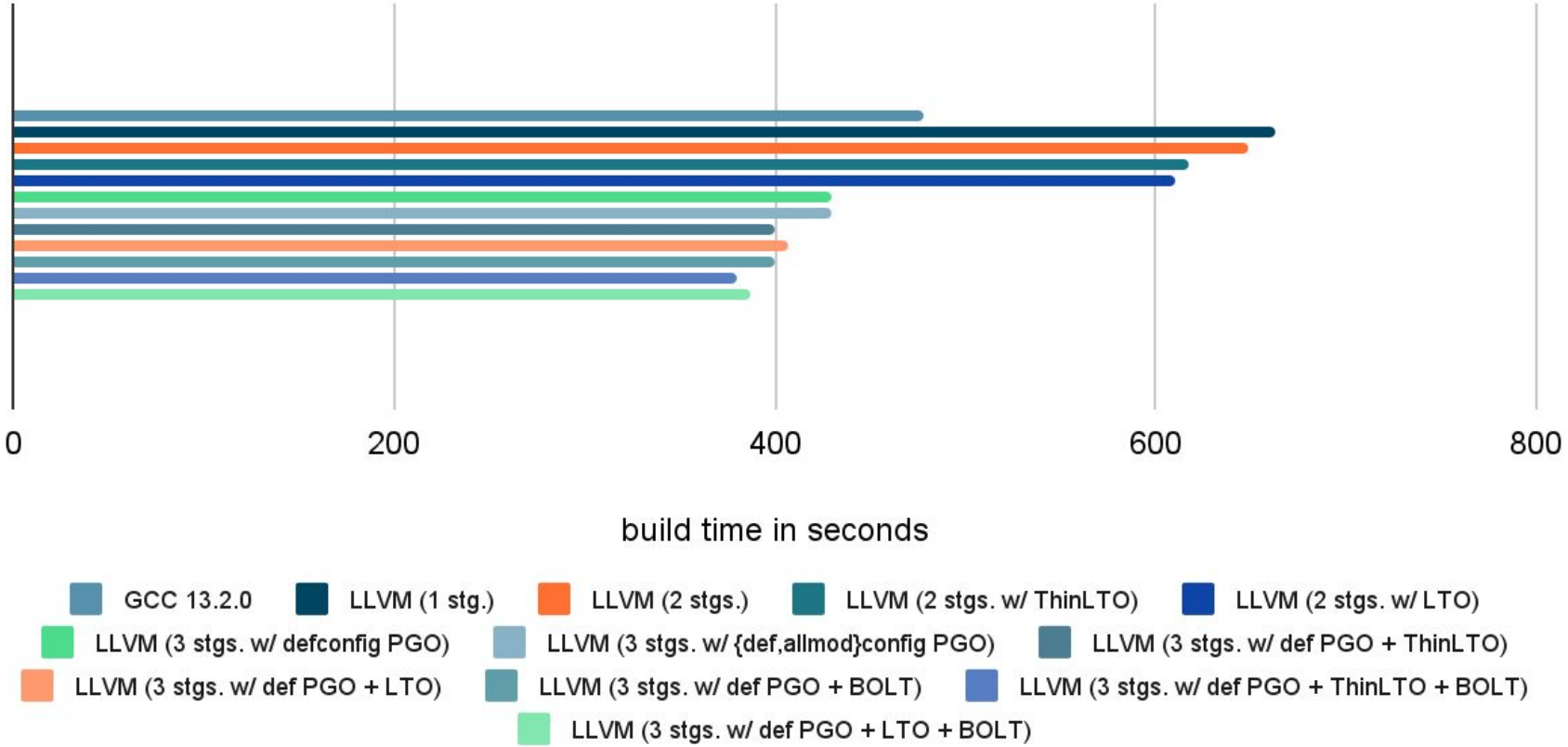
ARCH=arm allmodconfig (aarch64)

Averaged across five runs



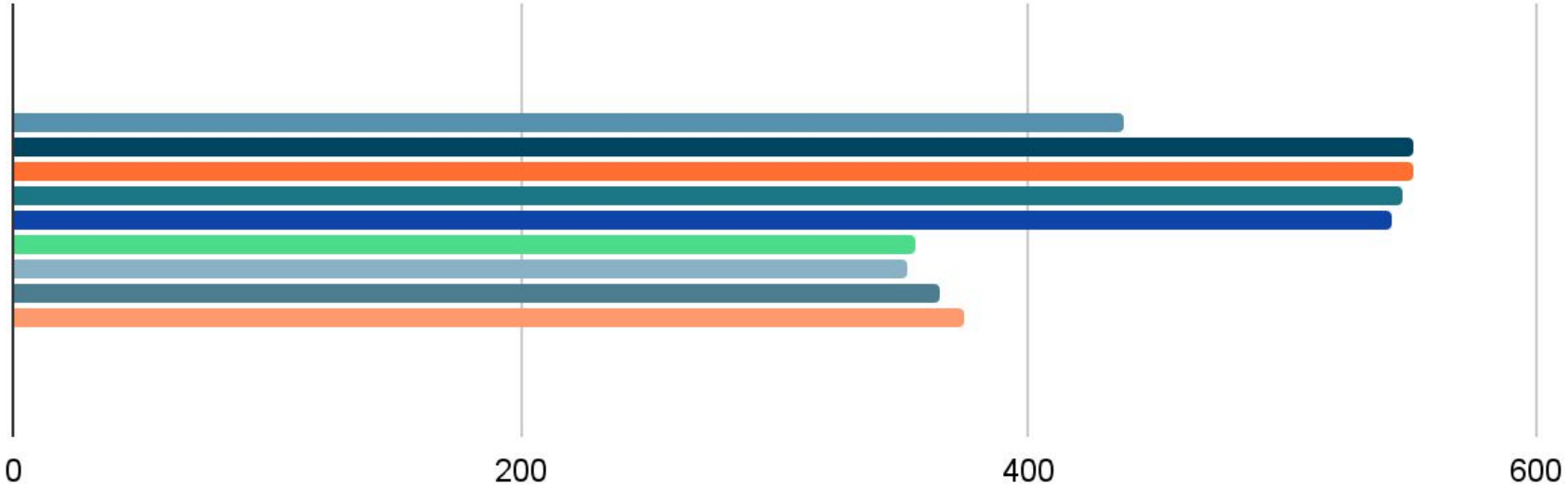
ARCH=arm allmodconfig (x86_64)

Averaged across five runs



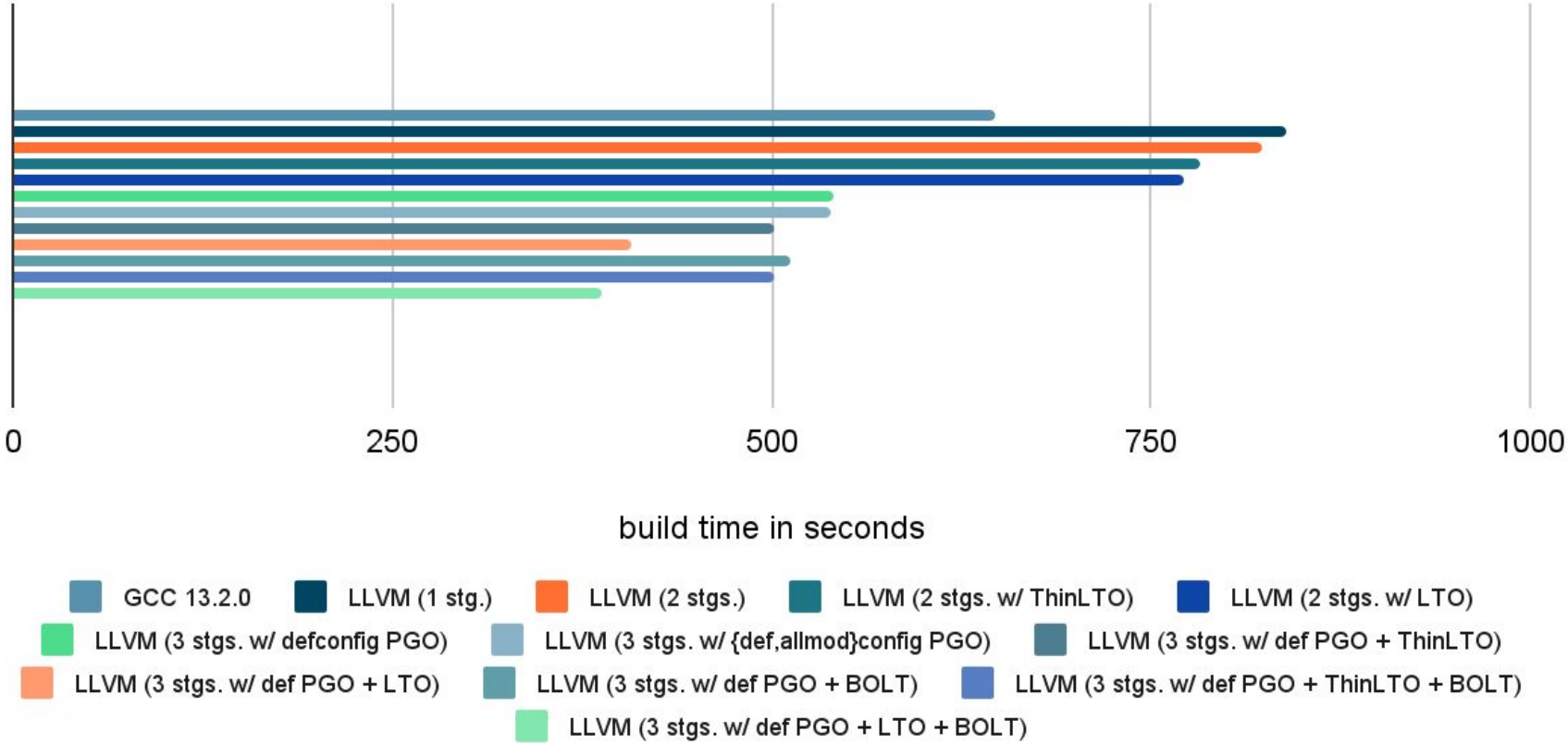
ARCH=arm64 allmodconfig (aarch64)

Averaged across five runs



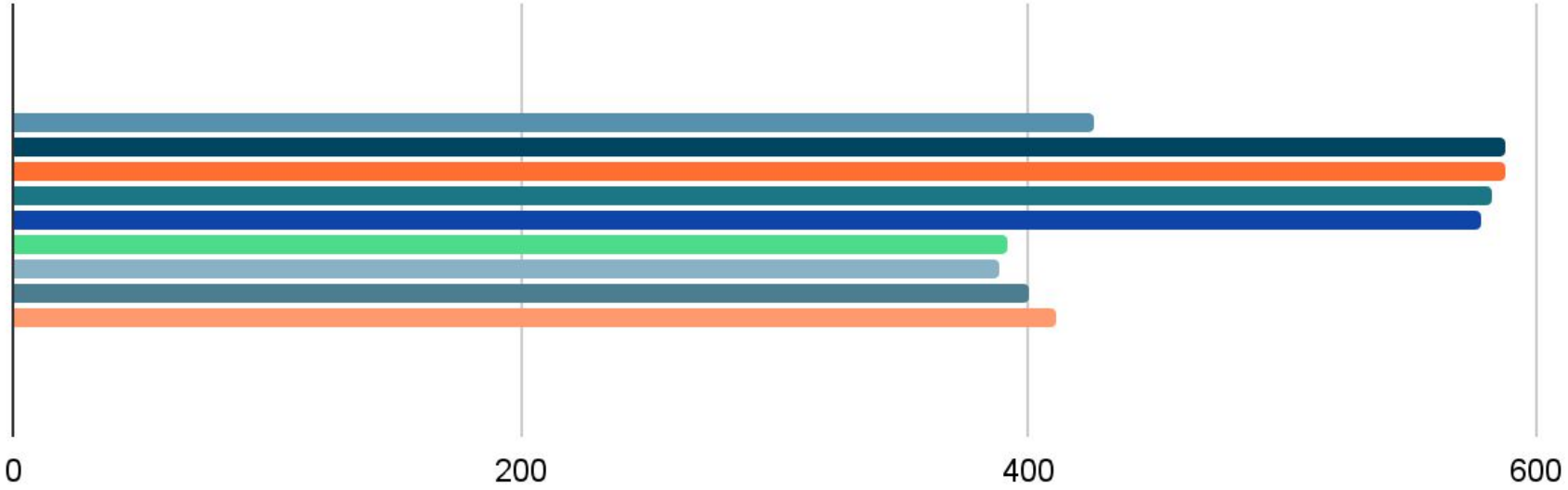
ARCH=arm64 allmodconfig (x86_64)

Averaged across five runs



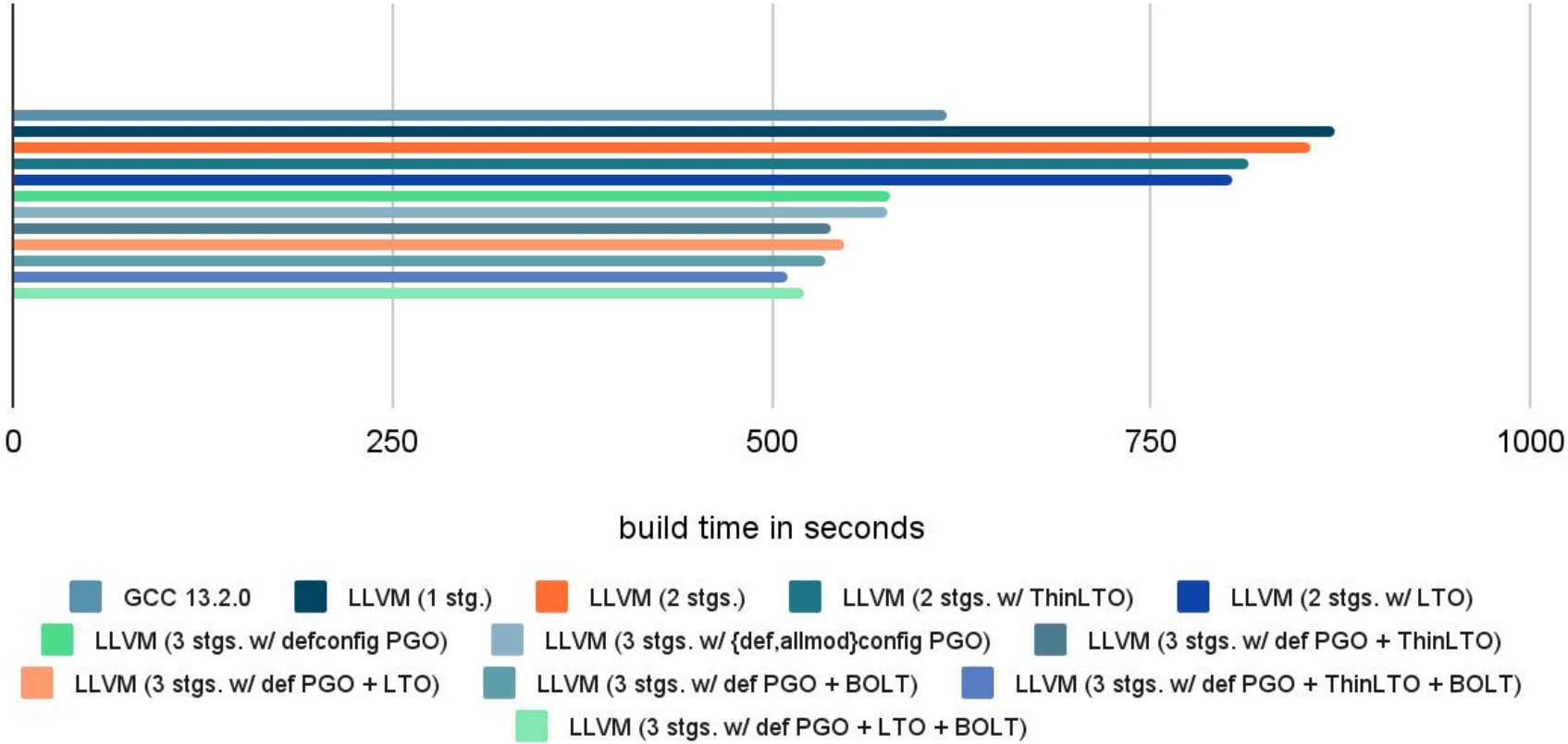
ARCH=x86_64 allmodconfig (aarch64)

Averaged across five runs



ARCH=x86_64 allmodconfig (x86_64)

Averaged across five runs



Observations / Conclusions

- PGO / BOLT big win
- LTO no win
- ThinLTO maybe win?
- Questions?