Improve Clang Performance

CppAlliance Fellowship 2026

Name: Ezzeldin Ibrahim
Email: ezzibrahimx@gmail.com
GitHub: github.com/AnonMiraj
Discord: AnonMiraj

Contents

---

1. Synopsis

Clang compiles most of the C and C++ code that ships today. Each release adds language features and diagnostics, and each addition risks making compilation slower. A 5% regression in one release compounds with the next. By the time you notice, builds have gotten slower across the board.

I will benchmark Clang across real-world codebases, find where time and memory go, and land targeted fixes upstream.

2. Benefits

Every developer who uses Clang benefits from faster compilation. CI pipelines finish sooner. `clangd` responds faster in IDEs. No one has to change their code. Cutting peak memory helps with large translation units and machines with limited RAM. The benchmarks I build will also make future regressions easier to catch before they ship.

3. Approach

The work has two phases: measure, then fix.

3.1 Benchmarking

I will compile several open-source projects with Clang 18, 19, 20, and trunk. The benchmark set includes LLVM itself, Linux kernel headers, Boost, SQLite, and a Qt or KDE module. These cover template-heavy C++, macro-heavy C, single-file compilation, and real-world application code.

For each project I record total compile time (`hyperfine`), per-phase breakdown from `-ftime-trace` (parsing, Sema, CodeGen, backend), and peak memory (`/usr/bin/time -v` or `valgrind --tool=massif`). Comparing across releases shows which phases got slower and by how much.

3.2 Profiling and Root-Cause Analysis

Once I find a regression or hot spot, I drill down with `perf record` / `perf report` for hot functions, `-ftime-trace` for phase attribution, and `valgrind` for instruction-level cache analysis (`callgrind`) and heap allocation profiling (`massif`). If a regression appeared between releases, `git bisect` pinpoints the commit.

I want to get specific: “this function in Sema allocates N times per template instantiation, and reducing that to 1 saves X%.”

3.3 Targeted Fixes

Each fix will be small and backed by benchmark numbers. A fix might eliminate redundant hash map lookups in name resolution, avoid unnecessary copies of `QualType` during constant evaluation, or cache repeated type trait queries. Shrinking AST node sizes improves cache locality. Deferring work that only matters when a diagnostic fires saves time on the common path. Every patch passes `check-clang` before submission.

4. Timeline

Week	Dates	Deliverable
0	May 4 – 24	Community bonding. Set up build infrastructure for multiple Clang versions. Select benchmark projects.
1–2	May 25 – Jun 7	Build benchmark harness. Automate compilation + measurement for all projects across Clang 18/19/20/trunk.
3–4	Jun 8 – 21	Run full benchmark suite. Collect `-ftime-trace` , wall-clock, and memory data. Produce comparison tables.
5–6	Jun 22 – Jul 5	Analyze results. Identify the top 3-5 regressions or hot spots. Write up findings with data.
7–8	Jul 6 – 19	Profile the first hot spot. Root-cause with `perf` and `git bisect` . Develop and test a fix.
9–10	Jul 20 – Aug 2	Profile and fix the second hot spot. Submit both patches for review.
11–12	Aug 3 – 16	Profile and fix the third hot spot. Address review feedback on earlier patches.
13–14	Aug 17 – 30	Continue fixing identified issues. Re-run benchmarks to validate improvements.
15–16	Aug 31 – Sep 13	Additional fixes or deeper investigation into memory usage. Validate no correctness regressions.
17–20	Sep 14 – Oct 11	Polish. Land remaining patches. Re-run full benchmark suite to measure cumulative improvement.
21–22	Oct 12 – 25	Write final report: regressions found, fixes landed, measured improvements. Final evaluation.

The timeline is flexible. If benchmarking reveals a single high-impact fix early, I will prioritize landing it and move on. If a fix turns out to be too invasive, I will document the finding and pick a different target.

5. Personal Information

Final-year CS student at Zagazig University. **30+ Pull Requests** merged into the LLVM main branch.

LLVM Libc Contributions

My primary upstream work has been in LLVM's libc project, implementing C23 math functions for `float16`:

- **`lgammaf16`:** Log-gamma function (`#192834`). The most complex function I implemented. It required Clenshaw evaluation of Chebyshev polynomials across multiple subintervals, a Stirling approximation for large arguments, and `sinpi`-based reflection for negative inputs.
- **Header-only refactoring:** Led a codebase-wide migration of math function implementations from `.cpp` files to header-only `__support/math/` headers. This touched hundreds of files across CMake, Bazel, and shared test infrastructure.

Automated Refactoring at Scale

Built a Tree-sitter-based refactoring tool that automated the header-only migration described above (`#182922`, `#176531`). The tool parsed C++ source files, extracted function bodies, generated header-only wrappers with correct namespace and include structure, and updated CMakeLists.txt and BUILD.bazel files. Wrote about the approach in Automating LLVM Refactoring. What would have taken three months of manual work was done in under a week. The maintainers were really happy :).

Other Experience

- **GSoC 2024 (Fortran-lang):** Built `fig`, a vector graphics library for Fortran.
- **Side Projects:** `Tanin` (Rust TUI noise generator), `game_of_life` (Conway's Game of Life in C).

Why This Project

I have done alot of measure performance work. A function either got faster or it did not. I already build LLVM from source for my libc work, so I know the build system, the test infrastructure, and the review process. My libc math implementations went through multiple rounds of polynomial degree reduction and precision tuning to hit performance targets, and I enjoyed that process.