Making Clang Faster

Six months of improving Clang performance - and how you can improve it even more

CrCon 2026 · Ezzeldin Ibrahim



whoami

- Hello, I am **Ezzeldin Ibrahim**
- A software engineer, particularly interested in compilers and other low-level systems
- The last 6 months I was a **compiler research fellow**, working on improving **Clang performance**
- In this talk I will show how successful I was - and how **you** can improve Clang even more

The scoreboard

Every PR from the fellowship, with its compile-time-tracker improvement:

PR	fix	`-O3` / LTO	`-O0 -g`
#202727	APINotes: early return when no apinotes files are loaded	−0.04%	**−0.13%**
#203710	APINotes: skip per-decl work when no notes are active	−0.03%	**−0.07%**
#208139	compute current instantiation less often in `lookupInBases`	−0.02%	**−0.05%**
#204926	only record lexer check-points when colors are enabled	+0.01%	+0.05% - fixed by #209355
#208862	skip macro source-location check for names that can't warn (−0.7% kernel headers)	≈0%	≈0%
#209355	fix check-point recording under default colors	−0.08%	**−0.22%**
#212213	cache the analysis-based warning policy	−0.06%	**−0.11%**
#218278	smaller AutoTypes map key - **−6.5% peak memory** (CTRE)	-	-
#206363	doc comments out of the AST - **reverted** (+7.6% `-O0 -g`), relanded as #221605	-	-
#219025	IndVarSimplify: batch `forgetValue` - **57×** faster (3:45 → 3.9 s)	-	-
#221605	doc comments only when needed (reland + cache), **−6%** comment-heavy TUs	−0.12%	**−0.31%**
#209694	no typo-correction for disabled diagnostics, **−1.2%** Boost.MPL	−0.13%	**−0.28%**
#228926	hash the main file once (DWARF 5)	−0.02%	**−0.21%**

The scoreboard, continued

Open PRs:

PR	fix	`-O3` / LTO	`-O0 -g`
#213790	quadratic pack indexing - **17×** (11.9 s → 0.69 s), 29× less memory	-	-
#225573	quadratic designated initializers - **127×** (8.87 s → 0.07 s)	-	-
#229550	reuse the PresumedLoc `setLocation` computed	−0.08%	**−0.55%**
#237803	check for a macro before the attribute's presumed file	−0.04%	−0.08%
#237805	check the macro name before its presumed file	−0.06%	−0.13%

Total: **18 compile-time PRs - 13 merged, 5 open.**

+ 4 patches not sent upstream yet: another **≈ −1%** at `-O0 -g`.

tracker = llvm-compile-time-tracker.com, geomean `instructions:u`

How AI was involved

Every slide from here on will talk about how AI was involved in that part. There are a lot of aspects that would have made this project much harder in a pre-LLM era.

Since I was supposed to improve Clang performance in general and not focus on specific parts, most of the areas I was not familiar with before the project:

- **The preprocessor** - macros, the lexer, and the diagnostics around them
- **Sema and the diagnostic engine** - warning policies, typo correction, API Notes
- **Templates** - instantiation, packs, designated initializers
- **The AST and type interning** - how Clang stores and deduplicates types
- **Debug info** - DWARF, and where source locations come from
- **The backend** - optimization passes and the IR they work on

AI helped me explore and understand these areas at a much faster rate.

Why I looked for regressions

Picking an area of Clang and making it faster would have been very hard.

So I went after the **gaps between what the maintainers do** instead. The obvious slowdowns are gone and the big regressions get caught fast, so what is left are the hidden ones - **−0.2% here**, **−0.3% there** - and they accumulate.

20 regressions found, 8 of them fixable. One of them sat in Clang for **two years**.

How I found them

Targets. SQLite first, then **Boost.MPL**. AI set up all ten projects and their dependencies.

The loop. Clang 18 → 23: sample commits, measure, bisect the range down to one commit. AI watched every bisect, kept the notes, and pushed a jump until it found the first bad commit.

The metric. Ir under valgrind - deterministic for every commit

manyclangs. 40 minutes to build a commit, even with ccache; manyclangs serves any commit in seconds

Once a commit was pinned, AI searched the reviews and the issue tracker for known overhead, and checked whether the regression was still live on trunk. Then I read the diff myself and wrote a fix.

For the regressions that were too complicated for me, I threw a couple of agents at them to try multiple different approaches, to see whether they could be truly improved.

What I learnt from finding regressions

Most of the fixes were a few lines, and they all fell into **three categories**:

A. Cache a heavy computation behind a simple boolean

B. Reorder the conditions so the heavy computation stops early

C. Something ran when it shouldn't - a lot of the time some of the diagnostics were computed even where the warning was disabled

None of them is really a clever solution - just a small oversight that maintainers didn't notice. Just the same pattern, with the same fix, and the regression is gone.

But after a while these regressions became hard to find, or too complicated of a change to know whether it is actually a regression or just the cost of a feature or a fix.

So I decided to look for other approaches

Searching for the causes instead

Instead of waiting for a regression to show up, go after the **cause directly**: search the codebase for the same patterns we noticed while fixing regressions.

By hand that is tedious and slow, and it is a perfect task for an agent - point it at a pattern in LLVM and it finds the occurrences.

The patterns are easy to solve, so the agent usually comes back with a working fix, tests it, and measures what it bought. Then it loops: find the next occurrence, fix it, measure it.

The last 8 patches were all found this way, and there are many more still in there.

An example: the same location, twice

#229550 - `setLocation` computes the `PresumedLoc` to get the line and the column, then `getOrCreateFile` computes it **again** for the same location and throws the first copy away.

```
-    CurLocFile = getOrCreateFile(CurLoc);
+    CurLocFile = getOrCreateFile(CurLoc, PCLoc);
```

The fix is one extra parameter: hand over the `PresumedLoc` that was already computed, instead of asking the `SourceManager` for it twice.

−0.55% at `-O0 -g` - tracker geomean.

Two more: check the cheap thing first

#228926 (merged) - the main file was hashed **twice** for the DWARF 5 checksum: once by `CreateCompileUnit`, once by `getOrCreateFile`.

```
+  // Reuse the checksum CreateCompileUnit already computed for the main file.
+  if (!CSInfo && FID == SM.getMainFileID() && TheCU)
+    CSInfo = TheCU->getFile()->getChecksum();
   if (!CSInfo) {
```

−0.21% at `-O0 -g`.

#237803 (open) - `ParseGNUAttributes` asked "is this a builtin file?", which computes a `PresumedLoc`, **before** the cheap file-ID comparison.

```
-    if (!isWrittenInBuiltinFile(getSpellingLoc(AttrTokLoc)) &&
-        FindLocsWithCommonFileID(AttrTokLoc, Loc)) {
+    if (FindLocsWithCommonFileID(AttrTokLoc, Loc) &&
+        !isWrittenInBuiltinFile(getSpellingLoc(AttrTokLoc))) {
```

−0.08% at `-O0 -g`.

Both fixes just make sure that the expensive check only runs if the cheap one passes.

These optimizations are **easy to make and hard to notice** - and they appear all over the codebase.
And most of the process can be automated.

Thank you! **Questions?** 