Hunting Clang Compile-Time Regressions

Ezzeldin Ibrahim (AnonMiraj)

Hunting Clang Compile-Time Regressions

Building every Clang, benchmarking code, bisecting the slowdowns.

C++ Alliance Fellowship 2026 · Ezzeldin Ibrahim

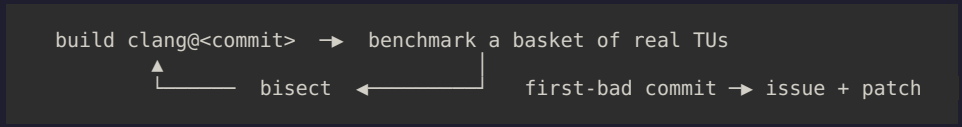
whoami

- My name is **Ezz-eldin**, "Ezz" for short. From Egypt
- Final-year CS student, supposed to graduate soon
- About a year contributing to LLVM, mostly **libc**.
- So I'm excited to work more with **Clang**
- Big Linux nerd . I use **NixOS** btw
- Playing a lot of **Slay the Spire 2** lately. DM me for a game

The project

I keep **building Clang**, every release and every commit, **benchmark codebases**, find **compile-time regressions**, **bisect** each one down to a single commit, and **send a fix upstream.**

The loop runs on my machine 24/7:



So far I've looked into benchmarking **SQLite** (a lot of macro-heavy C) and **Boost.MPL** (template-heavy C++).

Why this matters

We all hate long compile times. Trust me, after this project I don't think anyone hates them more than I do.

- A small win helps everyone. Shorter CI runs, less money and energy spent compiling, less developer frustration, and **nobody has to change their code** to get it back.
- A 5% regression in one release stacks on top of the next, and by the time you notice, everything got slower.
- Big codebases feel it most. Chromium has a mountain of dependencies. A clean build runs **6 to 7 hours on my machine**.

What I actually measure

- I split the cost into **frontend** (parse, Sema, template instantiation) and **codegen**
- I sweep **clang 18 → 19 → 20 → 21 → 22 → trunk** over the same basket of files
- I find out *why* with `-ftime-trace` and `-Xclang -print-stats`

Boost.MPL is the current target. It's header-only, so the `libs/mpl/test/*.cpp` files are the real translation units that instantiate the templates: **95 files** of heavy metaprogramming.

The mistake I made first

On SQLite I used **wall-clock time** (hyperfine). Wrong call.

The bisect builds Clang across all cores, then samples the probe immediately, on a thermally throttling laptop.

- The **same binary** ran **7%** slower when the laptop was hot
- The regression I was chasing was only about **2.5%**
- Verdict: a **false positive** (`46ad7ff4`), retracted

The fix: stop timing, count instructions

I stopped measuring time. I count **retired instructions (Ir)** with **callgrind**.

Wall-clock

- depends on heat, load, CPU frequency
- 7% swing on an *identical* binary
- needs many repeats, still noisy
- false positives

Callgrind Ir

- It is deterministic! So same clang + same file = same number
- immune to heat, load, core count
- one run per commit is enough
- bisect is trustworthy again

Same clang and same file give the exact same number every run, so I can compare two commits and trust the difference.

The workflow today

1. build clang @ commit (~45 min each, even with ccache)
2. callgrind Ir over the basket (deterministic, exact)
3. COARSE-BISECT: 9 commits across the whole range
4. refine the biggest bucket (repeat 3 on a smaller window)
5. git bisect run + Ir gate → exact first-bad commit
6. attribute with -ftime-trace → WHY it got slower

Steps 3 and 4 are very important. So why not just `git bisect` straight away?

Why coarse-bisect first

The MPL regressions are **diffuse**. There's no single bad commit.

- The 19→20 slowdown spreads over about **5 commits**, each only **0.3 to 0.5%**
- A naive bisect on a 20,000-commit range stops at whatever commit noise tips over the gate, often the wrong one
- Coarse-bisect samples **9 points across the range first**. I see the shape, one cliff or a long slope, then I zoom into the commits that actually cost something

I look at the whole range before I start digging into it.

First confirmed regression

`440b1743` · "*[APINotes] Upstream Sema logic to apply API Notes to decls*" (PR #73017)

- I found it with **coarse → refine → git-bisect** on the diffuse **18→19** step
- Clean single-commit jump: **+0.41% Ir** (+123M instructions on the basket)
- **Why:** it calls `ProcessAPINotes` on **every declaration**, even when there are no API notes at all. MPL's mountain of declarations and template instantiations pays that tax on each one

I filed an issue, and a patch is already up to fix it.

Challenges

- **Noise sends you down wrong paths.** Early wall-clock runs had me chasing false positives. Callgrind `Ir` killed most of that.
- **Some regressions are ambiguous.** Before API Notes I hit `8009bbec`, a Sema change that resolves member access earlier. It made everything a tiny bit slower with no single cause. The same PR also fixed a few crashes, so I couldn't even tell if it was a regression or the price of a necessary fix.
- **Some are already fixed on trunk.** A `#embed` token-stream cost from 18→19 got healed later by an unrelated optimization. Chasing those wastes time, so I confirm the slowdown still exists on trunk before I dig in.

The bottleneck: Compile Clang is a chore

Remember how I said I really, really hate compile times?

- My machine runs **24/7** building Clang commits and versions, then benchmarking with them
- Each build takes about **45 minutes**, even with ccache
- It locks up the laptop, so when I need the machine I build on **half the cores** and wait longer

Counting from the logs so far:

> **135+ Clang builds, and counting.**
> **~50** for the SQLite bisect, **76** coarse-bisect commits across **14 passes** on MPL, and the bisect steps.

What is next

- **Finish 18→19.** Still in progress, diffuse, a few contributors left to pin
- **Upstream the confirmed wins**, and chase the ambiguous ones to a verdict
- Need To have another go on SQLite with valgrind
- **Look into more codebases**: LLVM itself, kernel headers, and more

Thank you! Questions?