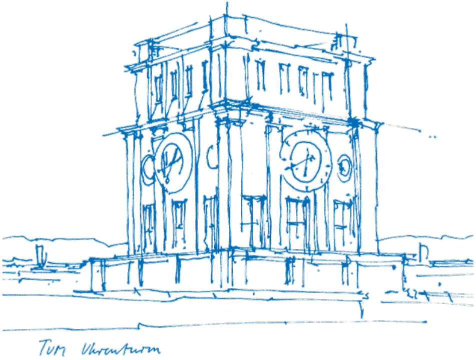
Faster Compilation in LLVM 20 and Beyond

Alexis Engelke
engelke@tum.de

Chair of Data Science and Engineering
Department of Computer Science
Technical University of Munich

EuroLLVM '25, Berlin, DE, 2025-04-16



Why Fast Compilation?

- Fast compilation *is* important, especially at -O0
- JIT compilation: databases, WebAssembly runtimes, ...
  - LLVM often used anyway, as high-quality compiler
  - Separate back-end increases maintenance cost
  - Fast baseline compilation $\Rightarrow$ low startup latency
- Developer experience: faster develop–test roundtrip, CIs
  - (Also needs to consider front-end)
- LLVM 18$\rightarrow$20 Back-end Performance: -18% (x86-64), -13% (AArch64)
- This talk: how we got there + how to be faster

General: Hash Maps

- Hash maps can be rather expensive
  - $\mathcal{O}(1)$ *asymptotic run-time*; but every access has a non-trivial cost

- For pointer maps: pointer dereference is fastest
  - E.g., add field to struct; but limits reusability
  - Example: worklist for `SDNode` (#92900, #94609)
- Dense numbering for keys, then use arrays
  - Example: add numbers for IR blocks (#101052) $\rightsquigarrow$ faster dominator tree
- Avoid redundant lookups
  - Example: reduce number of hash table lookups for symbol creation to one (#95464)
- Prefer `llvm::DenseMap`, `llvm::StringMap` when possible

General: Memory Allocations

- Memory allocations have a cost, esp. when done often
  - Cost depends on allocator, particularly noticeable with glibc's malloc
- Bump allocator can make allocations much cheaper
  - Additional benefit: improved spatial locality
  - Downside: can lead to higher `max-rss`, so no clear cut
  - Example: use for `MCFragment` (#96402), dominator tree nodes (#102516 (unmerged))
- Bump allocation of `MCFragment` contents/fixups would be nice
  - Bump-allocatable `SmallVector`?

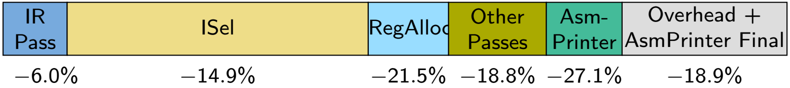
General: Miscellaneous

- Indirect/virtual function calls have some overhead
  - Especially avoidable: virtual functions that do nothing by default
  - Example: should allocate register class (#96296)
- `raw_svector_ostream`: every write goes through slow path (=virt. fn call)
  - Making slow path faster is beneficial (e.g., #97396), but not ideal
  - Ideally, use fast path with `SmallVector` itself as buffer
- Timers are not free even if disabled (global/TLS access)

-O0 Back-End Performance

- LLVM 18, x86-64:

- LLVM 20, x86-64: -18%



- LLVM 18, AArch64, GlobalISel:

- LLVM 20, AArch64, GlobalISel: -13%

Pre-ISel -O0 Back-End Passes

- 15–20 passes to prepare LLVM IR for back-end
- Mostly lowering intrinsics and some complex operations

⇝ For many functions, these do nothing

- Iterating over LLVM-IR is not free ⇝ reduce number of passes
  - Two passes merged into the pre-ISel intrinsic lowering (#97727, #101652)
- Goal (?): merge most of these into a single pre-ISel legalization pass

Machine IR

- Back-end mostly works on SSA-based Machine IR
- Very featureful, can represent machine code for various target architectures

- Fairly expensive to create/modify
  - `addOperand` takes considerable amount of time
  - Managing use–def lists of virtual/physical registers is expensive
- Supports storing extra-information inline and out-of-line
  - Reduces memory utilization, but leads to branch misses

Instruction Selection

- Transform/lower LLVM IR into Machine IR
  - FastISel: handle common cases in single step
  - SelectionDAG: rewrite to graph, match patterns, schedule into MIR
  - GlobalISel: rewrite to generic MIR, rewrite gMIR twice, rewrite to MIR

- Call lowering is not cheap (attributes, ABIs, etc.)
- SelectionDAG fallbacks are expensive
  - Adding more FastISel duplicates functionality – maintainability...

Instruction Selection: GlobalISel I

- Multi-pass: translate gMIR, legalize, select register bank, actual ISel
  - Additionally: combiners between passes; localizer for constants
- Fixed-point iteration often not really beneficial, esp. at `-O0`
  - Opt-in to do single pass of GISel combiners (#94291, #102167)
  - Also changed earlier for InstCombine and SelectionDAG
- Full dead code elimination is not cheap, but not always needed
  - Legalizer already performs DCE, so combiners don't need full DCE again
  - Use observer on combined instruction for sparse DCE (#102163)

Instruction Selection: GlobalISel II

- Generating "bad" IR to clean it up later is simple but expensive
  - Legalizer expands `i1` arithmetic at uses, resulting in unneeded instructions
  - Can use `KnownBits` to avoid such artifacts (`D159140`), always beneficial
- GlobalISel still 47% slower than FastISel
  - Multi-pass approach costly, esp. on already-slow Machine IR
  - Localizer can have quadratic runtime for large basic blocks
  - Add fast path to directly generate target MIR from IRTranslator?

Register Allocator

- Fast paths for common cases are important
  - Example: early exit for x86-typical single-tied-def case (#96284)
- Fast data structures are very important
  - Example: replacing `SparseSet` with a vector (#96323)
- Managing registers is expensive: handle all regunits
  - Regunits stored as difflist $\rightsquigarrow$ iteration has data dependencies
  - Maybe add simplified handling for subregisters to `RegAllocFast`?

x86-Specific Passes

- Many back-end passes are target-specific
- Several of these do nothing on typical input

- `-O0` compilation should not require a dominator tree
  - Example: x86 copy-flags lowering does nothing on typical IR
    $\leadsto$ detect such cases early and compute analysis only if required (#97628)
  - Optional analysis passes hard to model in legacy pass manager

- Passes for specific ISA features should be fast §f feature not used
  - Example: x86 AMX rarely used — track usage during ISel lowering and store in `MachineFunctionInfo`; add early exit to passes (#94358, #94989)
  - Keeping track of used ISA features in LLVM-IR would be better
  - Passes that do nothing still have a small cost

Machine Code Emission (AsmPrinter/MC)

- Lowers Machine IR to MC and writes object files, highly customizable
  - Various formats, hooks for instructions, NaCl bundles, full assembler, ...
  - Most functionality based on virtual function calls
- Not originally designed for performance

- Reduce virtual function calls
  - E.g., move shared functionality to base classes, avoid hooks that do nothing (e.g., #96785)
- Avoid copying data/instructions; vector append is just asymptotically $\mathcal{O}(1)$

- Still optimization potential when focusing on common path

Other Considerations

- LLVM's fundamental performance problem: incremental IR rewriting
  - Great for composability, but IR rewriting is expensive
- Compile-time performance is not the primary concern
  - Quality of generated code, size of generated code, maintainability, memory usage, reusability, libLLVM size, ...

- Front-ends tend to generate "bad" IR
- Front-end time increasingly dominates
  - Clang tends to become slower with more features, due to its architecture

⇝ Separate `-O0` back-end focusing on common case for >10x improvement

Summary

- LLVM back-end performance got substantially better over the last year
- Many small improvements (or inefficiencies) add up
- Optimizing for common path is important
- Fundamental performance of LLVM unlikely to change in near future

Thanks to all contributors and reviewers!